Dispense del Corso di

Algoritmi e Strutture Dati

Marco Bernardo Edoardo Bontà

Università degli Studi di Urbino "Carlo Bo"
Facoltà di Scienze e Tecnologie
Corso di Laurea in Informatica Applicata

Versione del 30/11/2007

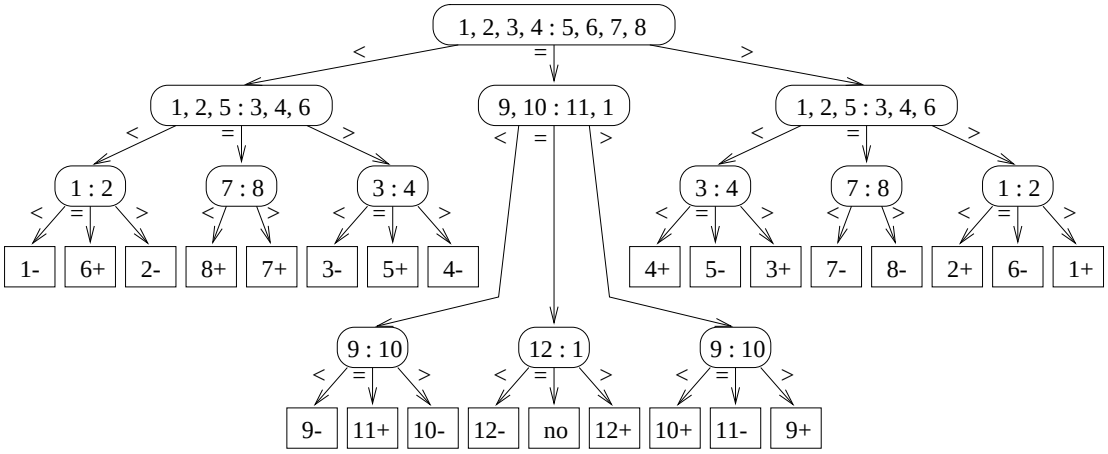
Indice

Capitolo 1

Introduzione agli algoritmi e alle strutture dati

1.1 Algoritmi e loro tipologie

- Il termine "algoritmo" deriva dal nome del matematico persiano Muhammad al-Khwarizmi (IX secolo), autore di uno dei più antichi trattati di algebra in cui descrisse procedure per eseguire calcoli aritmetici.
- Un algoritmo è una sequenza finita di passi interpretabili da un esecutore.
- Sebbene un algoritmo sia composto da un numero finito di passi, la sua esecuzione potrebbe richiedere un tempo non necessariamente finito.
- L'esecutore di un algoritmo non deve necessariamente essere un computer, ma può essere un dispositivo meccanico, un circuito elettronico o un sistema biologico.
- Quand'anche l'esecutore fosse un computer, un algoritmo non deve necessariamente essere espresso in un linguaggio di programmazione, in quanto esso si trova comunque ad un livello di astrazione più alto rispetto ad ogni programma che lo implementa.
- Un algoritmo non deve necessariamente essere espresso attraverso un supporto linguistico. A volte può essere più comodo rappresentare un algoritmo graficamente con un albero di decisione.
- Esempio: dati una bilancia a due piatti e 12 monete dello stesso aspetto tra le quali al più una potrebbe avere peso diverso dalle altre, la moneta diversa può essere individuata, specificando se pesa di più o di meno delle altre, mediante non più di 3 pesate come descritto dal seguente albero di decisione ($12 + 12 + 1 = 25$ diversi esiti non possono essere gestiti con meno di 3 pesate)



- Classificazione degli algoritmi basata sul numero di passi che possono eseguire contemporaneamente:
  - Algoritmi sequenziali: eseguono un solo passo alla volta.
  - Algoritmi paralleli: possono eseguire più passi per volta, avvalendosi di un numero prefissato di esecutori.
- Classificazione degli algoritmi basata sul modo in cui risolvono le scelte:
  - Algoritmi deterministici: ad ogni punto di scelta, intraprendono una sola via determinata in base ad un criterio prefissato (p.e. considerando il valore di un'espressione aritmetico-logica).
  - Algoritmi probabilistici: ad ogni punto di scelta, intraprendono una sola via determinata a caso (p.e. lanciando una moneta o un dado).
  - Algoritmi non deterministici: ad ogni punto di scelta, esplorano tutte le vie contemporaneamente (necessitano di un numero di esecutori generalmente non fissabile a priori).
- In questo corso assumeremo che l'esecutore sia un computer e ci limiteremo a considerare solo gli algoritmi sequenziali e deterministici. Useremo inoltre il linguaggio ANSI C per formalizzare tali algoritmi.

1.2 Correttezza di un algoritmo rispetto ad un problema

- C'è uno stretto legame tra algoritmi e problemi, non solo teorico ma anche pratico. La progettazione di algoritmi a partire da problemi provenienti dal mondo reale prende il nome di problem solving. Questa attività si basa su varie tecniche, tra le quali vedremo divide et impera, programmazione dinamica, tecniche golose e tecniche per tentativi e revoche.
- Fissati un insieme I di istanze dei dati di ingresso e un insieme S di soluzioni, un problema P può essere formalizzato come una relazione che ad ogni istanza associa le relative soluzioni:
$$P \subseteq I \times S$$
- Un problema può essere espresso in tre diverse forme:
  - Problema di decisione: richiede una risposta binaria rappresentante il soddisfacimento di qualche proprietà o l'esistenza di qualche entità, quindi $S = \{0, 1\}$.
  - Problema di ricerca: richiede di trovare una generica soluzione in corrispondenza di ciascuna istanza dei dati di ingresso.
  - Problema di ottimizzazione: richiede di trovare la soluzione ottima rispetto ad un criterio prefissato in corrispondenza di ciascuna istanza dei dati di ingresso.
- Mentre un problema specifica il cosa, cioè quali soluzioni produrre in uscita a partire dai dati ricevuti in ingresso, un algoritmo per quel problema descrive il come, cioè quale procedimento seguire per produrre le soluzioni attese a partire dai dati di ingresso.
- Dati un problema e un algoritmo, l'algoritmo è detto corretto rispetto al problema se, per ogni istanza dei dati di ingresso del problema, l'algoritmo termina e produce la soluzione corrispondente. Si parla di correttezza parziale qualora l'algoritmo non garantisca sempre la sua terminazione.
- Dato un problema, è sempre possibile trovare un algoritmo che lo risolve?

1.3 Complessità di un algoritmo rispetto all'uso di risorse

- Dato un problema, possono esistere più algoritmi che sono corretti rispetto ad esso.
- Questi algoritmi possono essere confrontati rispetto alla loro complessità o efficienza computazionale, cioè rispetto alla quantità di uso che essi fanno delle seguenti risorse durante la loro esecuzione:
  - Tempo di calcolo.
  - Spazio di memoria.
  - Banda trasmissiva.
- Poiché tramite un'opportuna gerarchia di memorie è possibile avere a disposizione una capacità di memoria praticamente illimitata e inoltre gli algoritmi che considereremo non richiedono lo scambio di dati tra computer, in questo corso ci concentreremo sulla complessità temporale degli algoritmi.
- Un motivo più profondo per concentrare l'attenzione sulla complessità temporale è che lo spazio e la banda occupati in un certo momento possono essere riusati in futuro (risorse recuperabili), mentre il passare del tempo è irreversibile (risorsa non recuperabile).
- Dato un problema, è sempre possibile trovare un algoritmo che lo risolve in maniera efficiente?

1.4 Strutture dati e loro tipologie

- C'è uno stretto legame pure tra algoritmi e strutture dati, in quanto le strutture dati costituiscono gli ingredienti di base degli algoritmi.
- Anche l'efficienza di un algoritmo dipende in maniera critica dal modo in cui sono organizzati i dati su cui esso deve operare.
- Una struttura dati è un insieme di dati logicamente correlati e opportunamente memorizzati, per i quali sono definiti degli operatori di costruzione, selezione e manipolazione.
- Le varie strutture dati sono riconducibili a combinazioni di strutture dati appartenenti alle quattro classi fondamentali che studieremo in questo corso: array, liste, alberi e grafi.
- Classificazione delle strutture dati basata sulla loro occupazione di memoria:
  - Strutture dati statiche: la quantità di memoria di cui esse necessitano è determinabile a priori (array).
  - Strutture dati dinamiche: la quantità di memoria di cui esse necessitano varia a tempo d'esecuzione e può essere diversa da esecuzione a esecuzione (liste, alberi, grafi). ■flf_1

Capitolo 2

Classi di problemi

2.1 Problemi decidibili e indecidibili

- Purtroppo non è sempre possibile trovare un algoritmo che risolve un problema dato. Di conseguenza i problemi si dividono in decidibili e indecidibili a seconda che possano essere risolti oppure no.
- Un problema è detto decidibile se esiste un algoritmo che produce la corrispondente soluzione in tempo finito per ogni istanza dei dati di ingresso del problema.
- Un problema è detto indecidibile se non esiste nessun algoritmo che produce la corrispondente soluzione in tempo finito per ogni istanza dei dati di ingresso del problema. In tal caso, sarà possibile calcolare la soluzione in tempo finito solo per alcune delle istanze dei dati di ingresso del problema, mentre per tutte le altre istanze non è possibile decidere quali siano le corrispondenti soluzioni.
- Un classico esempio di problema indecidibile è il problema della terminazione (Turing – 1937): dati un qualsiasi algoritmo e una qualsiasi istanza dei suoi dati di ingresso, stabilire se l'esecuzione di quell'algoritmo su quell'istanza dei suoi dati di ingresso termina oppure no.
- A dimostrazione di ciò, assumiamo che esista un algoritmo H che, preso in ingresso un qualsiasi algoritmo A e una qualsiasi istanza i dei suoi dati di ingresso, restituisce in tempo finito un valore di verità $H(A, i)$ che indica se A termina o meno quando viene eseguito su i. Poiché sia gli algoritmi che le loro istanze dei dati di ingresso sono descritti attraverso sequenze di simboli e queste possono essere univocamente codificate attraverso numeri interi, è lecito considerare un secondo algoritmo H' che prende in ingresso un qualsiasi algoritmo A e termina se e solo se $H(A, A)$ è falso. Ora, $H'(H')$ termina se e solo se $H(H', H')$ è falso, cioè se e solo se H' non termina quando viene eseguito su H'. Poiché ciò è assurdo, l'algoritmo H non può esistere e pertanto il problema della terminazione è indecidibile.
- Questo fondamentale risultato di Turing fu ispirato dal teorema di incompletezza di Gödel, un risultato fondamentale della logica secondo cui esistono teoremi che non possono essere dimostrati in nessun sistema formale che comprende l'aritmetica dei numeri interi. Entrambi i risultati ci dicono che non tutto è computabile.
- Vale la pena di osservare che esistono infiniti problemi decidibili che approssimano il problema della terminazione. Ognuno di essi può essere formulato nel seguente modo: dati un numero naturale n, un qualsiasi algoritmo e una qualsiasi istanza dei suoi dati di ingresso, stabilire se l'esecuzione di quell'algoritmo su quell'istanza dei suoi dati di ingresso termina oppure no entro n passi.
- Approssimazioni finite di questo genere sono ad esempio alla base di protocolli di comunicazione che necessitano di stabilire se un messaggio è giunto a destinazione oppure no. Poiché il mittente non riesce a distinguere tra perdita del messaggio e congestione della rete, in assenza di riscontro il messaggio viene di nuovo inviato dopo aver atteso per una quantità finita di tempo.

2.2 Problemi trattabili e intrattabili

- I problemi decidibili vengono poi classificati in base alla loro trattabilità, cioè alla possibilità di risolverli in maniera efficiente.

- Al fine di confrontare i problemi decidibili in modo equo rispetto alla loro trattabilità, conviene portarli tutti nella loro forma più semplice di problemi di decisione.

- Un problema di decisione decidibile è intrinsecamente intrattabile se non è risolvibile in tempo polinomiale nemmeno da un algoritmo non deterministico.

- Per i problemi di decisione decidibili che non sono intrinsecamente intrattabili si usa adottare la seguente classificazione:

  - $\mathcal{P}$ è l'insieme dei problemi di decisione risolvibili in tempo polinomiale da un algoritmo deterministico.
  - $\mathcal{NP}$ è l'insieme dei problemi di decisione risolvibili in tempo polinomiale da un algoritmo non deterministico. Equivalentemente, esso può essere definito come l'insieme dei problemi di decisione tali che la correttezza di una soluzione corrispondente ad un'istanza dei dati di ingresso può essere verificata in tempo polinomiale da un algoritmo deterministico.

- Ovviamente $\mathcal{P} \subseteq \mathcal{NP}$.

- L'inclusione di $\mathcal{P}$ in $\mathcal{NP}$ è stretta oppure i due insiemi coincidono? In altri termini, è possibile risolvere in maniera efficiente ogni problema non intrinsecamente intrattabile?

2.3 Teorema di Cook

- Il teorema di Cook è un risultato fondamentale nello studio della complessità della risoluzione dei problemi non intrinsecamente intrattabili.

- Esso si basa sul concetto di riducibilità in tempo polinomiale tra problemi e individua nel problema della soddisfacibilità l'archetipo di problema della classe $\mathcal{NP}$.

- Dati due problemi di decisione $P_1 \subseteq I_1 \times \{0,1\}$ e $P_2 \subseteq I_2 \times \{0,1\}$, diciamo che P_1 è riducibile in tempo polinomiale a P_2 se esiste un algoritmo deterministico che calcola in tempo polinomiale una funzione $f : I_1 \longrightarrow I_2$ tale che per ogni $i \in I_1$ ed $s \in \{0,1\}$ risulta:
$$(i,s) \in P_1 \Longleftrightarrow (f(i),s) \in P_2$$

- La riducibilità in tempo polinomiale di P_1 a P_2 implica che la soluzione di ogni istanza di P_1 può essere ottenuta risolvendo la corrispondente istanza di P_2 calcolabile in tempo polinomiale.

- Problema della soddisfacibilità: data un'espressione logica in forma normale congiuntiva, stabilire se esiste un assegnamento di valori di verità alle sue variabili che la rende vera.

- Teorema di Cook (1971): ogni problema della classe $\mathcal{NP}$ è riducibile in tempo polinomiale al problema della soddisfacibilità.

- Corollario: se esiste un algoritmo deterministico che risolve il problema della soddisfacibilità in tempo polinomiale, allora $\mathcal{NP} = \mathcal{P}$.

2.4 $\mathcal{NP}$-completezza

- Il problema di stabilire l'esatta relazione tra $\mathcal{P}$ ed $\mathcal{NP}$ è ancora aperto, quindi non sappiamo dire se tutti i problemi non intrinsecamente intrattabili siano risolvibili in maniera efficiente oppure no.
- Nella pratica i problemi in $\mathcal{NP}$ per i quali non sono noti algoritmi deterministici che li risolvono in tempo polinomiale vengono affrontati tramite algoritmi di approssimazione. Questi sono algoritmi deterministici che producono in tempo polinomiale una soluzione approssimata dei problemi in corrispondenza di ciascuna delle loro istanze dei dati di ingresso.
- Dal punto di vista teorico, sono stati individuati molti altri problemi ai quali si applica il corollario al teorema di Cook perché sono tanto complessi computazionalmente quanto il problema della soddisfacibilità. Questi problemi sono chiamati $\mathcal{NP}$-completi.
- Un problema in $\mathcal{NP}$ è detto $\mathcal{NP}$-completo se il problema della soddisfacibilità è riducibile ad esso in tempo polinomiale.
- Esempi di problemi $\mathcal{NP}$-completi:
  - Problema della soddisfacibilità.
  - Problema dello zaino: dati un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi e due interi positivi c e z, stabilire se esiste un sottoinsieme di A i cui elementi abbiano somma compresa tra c e z.
  - Problema delle scatole: dati un insieme $A = \{a_1, \ldots, a_n\}$ di interi positivi e due interi positivi k ed s, stabilire se esiste una partizione di A in k sottoinsiemi disgiunti $A_1, \ldots, A_k$ tale che la somma degli elementi in ogni A_i non supera s.
  - Problema del sottografo completo: dati un grafo G e un intero positivo k, stabilire se G ha un sottografo completo di k vertici.
  - Problema della colorazione di un grafo: dati un grafo G e un intero positivo k, stabilire se i vertici di G possono essere colorati con k colori in modo che non vi siano due vertici adiacenti dello stesso colore.
  - Problema del commesso viaggiatore: dati un grafo G in cui ogni arco ha un costo intero positivo e un intero positivo k, stabilire se G contiene un ciclo che attraversa ciascuno dei suoi vertici una sola volta in cui la somma dei costi degli archi non supera k. ■fff_2

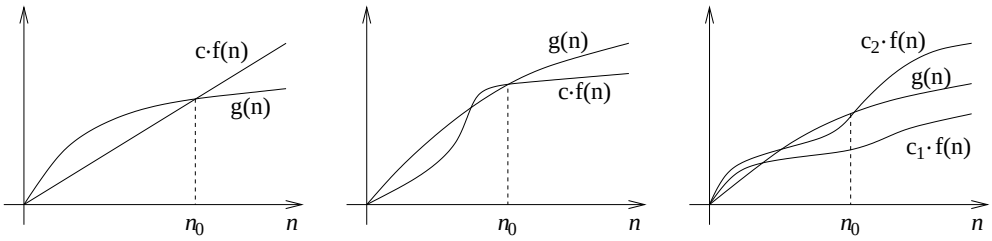
Capitolo 3

Complessità degli algoritmi

3.1 Notazioni per esprimere la complessità asintotica

- Gli algoritmi che risolvono lo stesso problema decidibile vengono confrontati sulla base della loro efficienza, misurata attraverso il loro tempo d'esecuzione.
- Il tempo d'esecuzione di un algoritmo viene espresso come una funzione della dimensione dei dati di ingresso, di solito denotata con $T(n)$.
- La caratterizzazione della dimensione n dei dati di ingresso dipende dallo specifico problema. Può essere il numero di dati di ingresso oppure la quantità di memoria necessaria per contenere tali dati.
- Al fine di catturare l'intrinseca complessità di un algoritmo, il tempo d'esecuzione deve essere indipendente dalla tecnologia dell'esecutore, quindi non può essere misurato attraverso il tempo di CPU impiegato su un certo computer dal programma che implementa l'algoritmo.
- Il tempo d'esecuzione di un algoritmo è dato dal numero di passi base compiuti durante l'esecuzione dell'algoritmo. Volendo formalizzare gli algoritmi nel linguaggio ANSI C, un passo base è costituito dall'esecuzione di un'istruzione di assegnamento (priva di chiamate di funzioni) o dalla valutazione di un'espressione (priva di chiamate di funzioni) contenuta in un'istruzione di selezione o ripetizione.
- Il tempo d'esecuzione di un algoritmo può essere calcolato in tre diversi casi:
  - Caso pessimo: è determinato dall'istanza dei dati di ingresso che massimizza il tempo d'esecuzione, quindi fornisce un limite superiore alla quantità di risorse computazionali necessarie all'algoritmo.
  - Caso ottimo: è determinato dall'istanza dei dati di ingresso che minimizza il tempo d'esecuzione, quindi fornisce un limite inferiore alla quantità di risorse computazionali necessarie all'algoritmo.
  - Caso medio: è determinato dalla somma dei tempi d'esecuzione di tutte le istanze dei dati di ingresso, con ogni addendo moltiplicato per la probabilità di occorrenza della relativa istanza dei dati di ingresso.
- Poiché il confronto degli algoritmi che risolvono lo stesso problema decidibile si riduce a confrontare delle funzioni – le quali possono relazionarsi in modi diversi per istanze diverse dei dati di ingresso – di solito si ragiona in termini di complessità computazionale asintotica.
- Le funzioni tempo d'esecuzione dei vari algoritmi che risolvono lo stesso problema decidibile vengono dunque confrontate considerando il loro andamento al crescere della dimensione n dei dati di ingresso. Ciò significa che, all'interno delle funzioni, si possono ignorare le costanti moltiplicative e i termini non dominanti al crescere di n.

- Notazioni per esprimere relazioni asintotiche tra funzioni intere di variabili intere:
  - Limite asintotico superiore (intuitivamente, $g(n)$ cresce al più come $f(n)$ per $n \to \infty$):
    $g(n) = O(f(n)) \iff \exists c, n_0 > 0.\ \forall n \geq n_0.\ g(n) \leq c \cdot f(n)$
  - Limite asintotico inferiore (intuitivamente, $g(n)$ cresce almeno come $f(n)$ per $n \to \infty$):
    $g(n) = \Omega(f(n)) \iff \exists c, n_0 > 0.\ \forall n \geq n_0.\ g(n) \geq c \cdot f(n)$
  - Limite asintotico stretto (intuitivamente, $g(n)$ cresce come $f(n)$ per $n \to \infty$):
    $g(n) = \Theta(f(n)) \iff \exists c_1, c_2, n_0 > 0.\ \forall n \geq n_0.\ c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)$



- Classi di complessità asintotica degli algoritmi:
  - $T(n) = O(1)$: complessità costante (cioè $T(n)$ non dipende dalla dimensione n dei dati di ingresso).
  - $T(n) = O(\log n)$: complessità logaritmica.
  - $T(n) = O(n)$: complessità lineare.
  - $T(n) = O(n \cdot \log n)$: complessità pseudolineare (così detta da $n \cdot \log n = O(n^{1+\epsilon})$ per ogni $\epsilon > 0$).
  - $T(n) = O(n^2)$: complessità quadratica.
  - $T(n) = O(n^3)$: complessità cubica.
  - $T(n) = O(n^k), k > 0$: complessità polinomiale.
  - $T(n) = O(\alpha^n), \alpha > 1$: complessità esponenziale.

3.2 Calcolo della complessità di algoritmi non ricorsivi

- Il tempo d'esecuzione di un algoritmo non ricorsivo può essere calcolato utilizzando le seguenti regole definite per induzione sulla struttura dell'algoritmo:
  - Il tempo d'esecuzione (risp. di valutazione) di un'istruzione di assegnamento `x = e;` (risp. di un'espressione `e`) è:
    $$T(n) = \begin{cases} 1 = O(1) & \text{se priva di chiamate di funzioni} \\ T'(n) + 1 = O(f'(n)) & \text{se contiene chiamate di funzioni eseguibili in } T'(n) = O(f'(n)) \end{cases}$$
  - Il tempo d'esecuzione di una sequenza $S_1 S_2 \ldots S_s$ di $s \geq 2$ istruzioni, ciascuna avente tempo d'esecuzione $T_i(n) = O(f_i(n))$ per $1 \leq i \leq s$, è:
    $$T(n) = \sum_{i=1}^{s} T_i(n) = O(\max\{f_i(n) \mid 1 \leq i \leq s\})$$
  - Il tempo d'esecuzione di un'istruzione di selezione `if` (β) S_1 `else` S_2 in cui:
    - il tempo di valutazione dell'espressione β è $T_0(n) = O(f_0(n))$;
    - il tempo d'esecuzione dell'istruzione S_1 è $T_1(n) = O(f_1(n))$;
    - il tempo d'esecuzione dell'istruzione S_2 è $T_2(n) = O(f_2(n))$;

    è:
    $$T(n) = T_0(n) + op(T_1(n), T_2(n)) = O(\max\{f_0(n), op(f_1(n), f_2(n))\})$$
    dove op è max nel caso pessimo e min nel caso ottimo.

– Il tempo d'esecuzione di un'istruzione di ripetizione `while` (β) S in cui:

∗ il numero di iterazioni è $i(n) = O(f(n))$;

∗ il tempo di valutazione dell'espressione β è $T_1(n) = O(f_1(n))$;

∗ il tempo d'esecuzione dell'istruzione S è $T_2(n) = O(f_2(n))$;

è:

$$T(n) = i(n) \cdot (T_1(n) + T_2(n)) + T_1(n) = O(f(n) \cdot \max\{f_1(n), f_2(n)\})$$

Se $T_1(n)$ o $T_2(n)$ dipende anche dalla specifica iterazione j dove $1 \leq j \leq i(n)$ – nel qual caso usiamo la notatione $T_1(n, j)$ e $T_2(n, j)$ – allora diventa:

$$T(n) = \sum_{j=1}^{i(n)} (T_1(n, j) + T_2(n, j)) + T_1(n, i(n) + 1)$$

- Esempi:

– Il seguente algoritmo non ricorsivo per calcolare il fattoriale di $n \geq 1$:

```
int calcola_fatt(int n)
{
  int fatt,
      i;

  for (fatt = 1, i = 2;
       (i <= n);
       i++)
    fatt *= i;
  return(fatt);
}
```

ha complessità:

$$T(n) = 1 + (n - 1) \cdot (1 + 1 + 1) + 1 = 3 \cdot n - 1 = O(n)$$

– Il seguente algoritmo non ricorsivo per calcolare l'n-esimo numero di Fibonacci ($n \geq 1$):

```
int calcola_fib(int n)
{
  int fib,
      ultimo,
      penultimo,
      i;

  if ((n == 1) || (n == 2))
    fib = 1;
  else
    for (ultimo = penultimo = 1, i = 3;
         (i <= n);
         i++)
    {
      fib = ultimo + penultimo;
      penultimo = ultimo;
      ultimo = fib;
    }
  return(fib);
}
```

nel caso pessimo ($n \geq 3$) ha complessità:

$$T(n) = 1 + 1 + (n-2) \cdot (1+1+1+1+1) + 1 = 5 \cdot n - 7 = O(n)$$

mentre nel caso ottimo ($n \leq 2$) ha complessità:

$$T(n) = 1 + 1 = 2 = O(1)$$

– Il seguente algoritmo non ricorsivo per calcolare il massimo di un insieme di $n \geq 1$ interi:

```
int calcola_max(int a[],
                int n)
{
  int max,
      i;

  for (max = a[0], i = 1;
       (i < n);
       i++)
    if (a[i] > max)
      max = a[i];
  return(max);
}
```

nel caso pessimo (array ordinato rispetto a $<$) ha complessità:

$$T(n) = 1 + (n-1) \cdot (1+1+1+1) + 1 = 4 \cdot n - 2 = O(n)$$

mentre nel caso ottimo (massimo contenuto in `a[0]`) ha complessità:

$$T(n) = 1 + (n-1) \cdot (1+1+1) + 1 = 3 \cdot n - 1 = O(n)$$

Per il problema dato non è possibile trovare un algoritmo migliore di quello mostrato, in quanto si può dimostrare che il problema in questione non può essere risolto con un algoritmo di complessità asintotica inferiore a quella lineare. ■fff_3

3.3 Calcolo della complessità di algoritmi ricorsivi

- Un algoritmo ricorsivo ha la seguente struttura:

```
soluzione risolvi(problema p)
{
  soluzione s, s1, s2, ..., sn;
  if (◁p semplice▷)
    ◁calcola direttamente la soluzione s di p▷;
  else
  {
    ◁dividi p in p1, p2, ..., pn della stessa natura di p▷;
    s1 = risolvi(p1); s2 = risolvi(p2); ...; sn = risolvi(pn);
    ◁combina s1, s2, ..., sn per ottenere la soluzione s di p▷;
  }
  return(s);
}
```

- Il tempo d'esecuzione di un algoritmo ricorsivo non può essere definito in forma chiusa attraverso le regole viste in precedenza, ma in modo induttivo attraverso una relazione di ricorrenza le cui incognite costituiscono una successione di numeri interi positivi corrispondenti ai valori di $T(n)$. Ci concentriamo su due tipi di relazioni di ricorrenza lineari: quelle di ordine costante e quelle di ordine non costante.

- Una relazione di ricorrenza lineare, di ordine costante k, a coefficienti costanti e omogenea ha forma:

$$\begin{cases} x_n = a_1 \cdot x_{n-1} + a_2 \cdot x_{n-2} + \ldots + a_k \cdot x_{n-k} & \text{per } n \geq k \\ x_i = d_i & \text{per } 0 \leq i \leq k-1 \end{cases}$$

cioè l'n-esima incognita x_n è espressa come combinazione lineare (senza termine noto) delle k incognite che la precedono, a meno che non sia uno dei primi k elementi della successione. Ci sono due casi:

- Se $k = 1$, la soluzione in forma chiusa è:
$$x_n = \begin{cases} d_0 & = & O(1) & \text{se } a_1 = 1 \\ d_0 \cdot {a_1}^n & = & O({a_1}^n) & \text{se } a_1 > 1 \end{cases}$$
- Se $k > 1$, la relazione può essere risolta nel seguente modo. Assumiamo che esistano delle soluzioni in forma chiusa del tipo $x_n = c \cdot z^n$ con $c \neq 0 \neq z$. Allora sostituendole nella prima equazione si ha $c \cdot z^{n-k} \cdot (z^k - \sum_{i=1}^{k} a_i \cdot z^{k-i}) = 0$ e quindi $z^k - \sum_{i=1}^{k} a_i \cdot z^{k-i} = 0$ essendo $c \neq 0 \neq z$. Se le radici $z_1, z_2, \ldots, z_k$ del polinomio associato alla relazione sono tutte distinte, per la linearità e l'omogeneità della relazione la soluzione in forma chiusa più generale è:
$$x_n = \sum_{j=1}^{k} c_j \cdot {z_j}^n = O(\max\{{z_j}^n \mid 1 \leq j \leq k\})$$
dove i c_j vengono ricavati per sostituzione nelle ultime k equazioni della relazione.

- Una relazione di ricorrenza lineare, di ordine costante k, a coefficienti costanti e non omogenea ha forma:
$$\begin{cases} x_n & = & a_1 \cdot x_{n-1} + a_2 \cdot x_{n-2} + \ldots + a_k \cdot x_{n-k} + h & \text{per } n \geq k \\ x_i & = & d_i & \text{per } 0 \leq i \leq k-1 \end{cases}$$
Se $\sum_{j=1}^{k} a_j \neq 1$ ci si riconduce al caso precedente ponendo $y_n = x_n + h/(\sum_{j=1}^{k} a_j - 1)$, da cui segue che:
$$x_n = O(y_n)$$

- Una relazione di ricorrenza lineare, di ordine non costante (cioè in cui l'n-esima incognita x_n non è espressa come combinazione a passo costante delle incognite che la precedono) e con lavoro di combinazione:
  - costante, ha forma:
  $$\begin{cases} x_n & = & a \cdot x_{n/b} + c & \text{per } n > 1 \\ x_1 & = & d \end{cases}$$
  e soluzione:
  $$x_n = \begin{cases} O(\log_b n) & \text{se } a = 1 \\ O(n^{\log_b a}) & \text{se } a > 1 \end{cases}$$
  - lineare, ha forma:
  $$\begin{cases} x_n & = & a \cdot x_{n/b} + (c_1 \cdot n + c_2) & \text{per } n > 1 \\ x_1 & = & d \end{cases}$$
  e soluzione:
  $$x_n = \begin{cases} O(n) & \text{se } a < b \\ O(n \cdot \log_b n) & \text{se } a = b \\ O(n^{\log_b a}) & \text{se } a > b \end{cases}$$
  - polinomiale, ha forma:
  $$\begin{cases} x_n & = & a \cdot x_{n/b} + (c_p \cdot n^p + c_{p-1} \cdot n^{p-1} + \ldots + c_1 \cdot n + c_0) & \text{per } n > 1 \\ x_1 & = & d \end{cases}$$
  e soluzione:
  $$x_n = O(n^{\log_b a}) \quad \text{se } a > b^p$$

- Qualora il tempo d'esecuzione di un algoritmo ricorsivo sia definito tramite una relazione di ricorrenza di tipo diverso da quelli trattati sopra, la soluzione in forma chiusa può essere ottenuta mediante il metodo delle sostituzioni. Esso consiste nell'ipotizzare una soluzione per la relazione di ricorrenza e nel verificarla per induzione sulla dimensione n dei dati di ingresso andando a sostituirla alle incognite.

- Esempi:
  - Il seguente algoritmo ricorsivo per calcolare l'n-esimo numero di Fibonacci ($n \geq 1$):

```
int calcola_fib_ric(int n)
{
  int fib;

  if ((n == 1) || (n == 2))
    fib = 1;
  else
    fib = calcola_fib_ric(n - 1) + calcola_fib_ric(n - 2);
  return(fib);
}
```

    ha tempo d'esecuzione definito da:

$$\begin{cases} T(n) = 1 + T(n-1) + T(n-2) + 1 = T(n-1) + T(n-2) + 2 & \text{per } n \geq 3 \\ T(1) = 1 + 1 = 2 \\ T(2) = 1 + 1 = 2 \end{cases}$$

    Questa è una relazione di ricorrenza lineare, di ordine costante 2, a coefficienti costanti e non omogenea. Poiché la somma dei coefficienti nel lato destro della prima equazione è diversa da 1, si può passare alla corrispondente relazione omogenea, il cui polinomio associato è $z^2 - z - 1 = 0$. Poiché tale polinomio ha radici $(1 \pm \sqrt{5})/2$, risulta:

$$T(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

  - Il seguente algoritmo ricorsivo per calcolare il massimo e il submassimo di un insieme di $n \geq 2$ interi (che restituisce un risultato di tipo `coppia`, il quale è una struttura composta da due campi di tipo `int` chiamati `max` e `submax`):

```
coppia calcola_max_submax_ric(int a[],
                              int sx,
                              int dx)
{
  coppia ms,
         ms1,
         ms2;

  if (dx - sx + 1 == 2)
  {
    ms.max = (a[sx] >= a[dx])?
              a[sx]:
              a[dx];
    ms.submax = (a[sx] >= a[dx])?
                 a[dx]:
                 a[sx];
  }
  else
  {
    ms1 = calcola_max_submax_ric(a,
                                 sx,
                                 (sx + dx) / 2);
    ms2 = calcola_max_submax_ric(a,
                                 (sx + dx) / 2 + 1,
                                 dx);
```

```
    ms.max = (ms1.max >= ms2.max)?
               ms1.max:
               ms2.max;
    ms.submax = (ms1.max >= ms2.max)?
                  ((ms2.max >= ms1.submax)?
                     ms2.max:
                     ms1.submax):
                  ((ms1.max >= ms2.submax)?
                     ms1.max:
                     ms2.submax);
  }
  return(ms);
}
```

ha tempo d'esecuzione definito da:

$$\begin{cases} T(n) &= 1 + T(n/2) + 1 + T(n/2) + 1 + 1 + 1 = 2 \cdot T(n/2) + 5 \quad \text{per } n > 2 \\ T(2) &= 1 + 1 + 1 = 3 \end{cases}$$

Questa è una relazione di ricorrenza lineare, di ordine non costante e con lavoro di combinazione costante, la cui soluzione è:

$$T(n) = O(n^{\log_2 2}) = O(n)$$

– Il seguente algoritmo ricorsivo per calcolare il fattoriale di $n \geq 1$:

```
int calcola_fatt_ric(int n)
{
  int fatt;

  if (n == 1)
    fatt = 1;
  else
    fatt = n * calcola_fatt_ric(n - 1);
  return(fatt);
}
```

ha tempo d'esecuzione definito da:

$$\begin{cases} T(n) &= 1 + T(n-1) + 1 = T(n-1) + 2 \quad \text{per } n \geq 2 \\ T(1) &= 1 + 1 = 2 \end{cases}$$

Questa è una relazione di ricorrenza lineare, di ordine costante 1, a coefficienti costanti e non omogenea. Poiché la somma dei coefficienti nel lato destro della prima equazione è 1, non si può passare alla corrispondente relazione omogenea, quindi bisogna applicare il metodo delle sostituzioni. Proviamo che $T(n) = 2 \cdot n$ procedendo per induzione su $n \geq 1$:

* Sia $n = 1$. Risulta $T(1) = 2$ e $2 \cdot 1 = 2$, da cui l'asserto è vero per $n = 1$.
* Supponiamo $T(n) = 2 \cdot n$ per un certo $n \geq 1$. Risulta $T(n+1) = T(n) + 2 = 2 \cdot n + 2$ per ipotesi induttiva. Poiché $2 \cdot n + 2 = 2 \cdot (n+1)$, l'asserto è vero per $n + 1$.

Di conseguenza:

$$T(n) = O(n)$$

- In generale, gli algoritmi ricorsivi il cui tempo d'esecuzione è espresso tramite una relazione di ricorrenza lineare di ordine non costante hanno una complessità asintotica polinomiale, mentre quelli il cui tempo d'esecuzione è espresso tramite una relazione di ricorrenza lineare di ordine costante maggiore di 1 hanno una complessità asintotica esponenziale. Il motivo per cui questi ultimi sono meno efficienti è dato dal modo in cui questi algoritmi operano sui loro dati. Poiché tali algoritmi non dividono i loro dati in sottoinsiemi disgiunti, finiscono per ripetere gli stessi calcoli più volte.

- Esempio: l'esecuzione di `calcola_fib_ric(4)` invoca `calcola_fib_ric(3)` e `calcola_fib_ric(2)`, con `calcola_fib_ric(3)` che invoca di nuovo a sua volta `calcola_fib_ric(2)`.
- A parità di complessità temporale, un algoritmo ricorsivo e un algoritmo non ricorsivo che risolvono lo stesso problema possono avere complessità spaziali diverse. In particolare, va ricordato che per la sua implementazione un algoritmo ricorsivo necessita dell'allocazione e della manipolazione di una pila per far sì che le varie chiamate ricorsive non interferiscano tra loro, così da evitare perdite d'informazione e quindi la produzione di risultati errati. ■fif_4

Capitolo 4

Algoritmi per array

4.1 Array: definizioni di base e problemi classici

- Un array è una struttura dati statica e omogenea, cioè i cui elementi non variano di numero a tempo d'esecuzione e sono tutti dello stesso tipo.
- Gli elementi di un array sono memorizzati consecutivamente e individuati attraverso la loro posizione. L'indirizzo di ciascuno di essi è determinato dalla somma dell'indirizzo del primo elemento dell'array e dell'indice che individua l'elemento in questione all'interno dell'array.
- Poiché l'accesso a ciascun elemento di un array è diretto, l'operazione di lettura o modifica del valore di un elemento di un array ha complessità asintotica $O(1)$ rispetto al numero di elementi dell'array.
- Problema della visita: dato un array, attraversare tutti i suoi elementi esattamente una volta.
- Problema della ricerca: dati un array e un valore, stabilire se il valore è contenuto in un elemento dell'array, riportando in caso affermativo l'indice di tale elemento.
- Problema dell'ordinamento: dato un array i cui elementi contengono tutti una chiave d'ordinamento e data una relazione d'ordine totale sul dominio delle chiavi, determinare una permutazione dei valori contenuti negli elementi dell'array tale che la nuova disposizione delle chiavi soddisfa la relazione.

4.2 Algoritmo di visita per array

- Gli elementi dell'array vengono attraversati uno dopo l'altro nell'ordine in cui sono memorizzati e i loro valori vengono sottoposti a qualche forma di elaborazione:

```
void visita_array(int a[],
                  int n)
{
  int i;

  for (i = 0;
       (i < n);
       i++)
    elabora(a[i]);
}
```

- Se l'elaborazione del valore di ogni elemento si svolge in al più d passi, in ogni caso la complessità è:

$$T(n) = 1 + n \cdot (1 + d + 1) + 1 = (d + 2) \cdot n + 2 = O(n)$$

4.3 Algoritmo di ricerca lineare per array

- Gli elementi dell'array vengono attraversati uno dopo l'altro nell'ordine in cui sono memorizzati finché il valore cercato non viene trovato e la fine dell'array non viene raggiunta:

```
int ricerca_lineare_array(int a[],
                          int n,
                          int valore)
{
  int i;

  for (i = 0;
       ((i < n) && (a[i] != valore));
       i++);
  return((i < n)?
           i:
           -1);
}
```

- Nel caso pessimo (valore non presente nell'array) la complessità è:
$$T(n) = 1 + n \cdot (1 + 1) + 1 = 2 \cdot n + 2 = O(n)$$
mentre nel caso ottimo (valore presente nel primo elemento dell'array) la complessità è:
$$T(n) = 1 + 1 = 2 = O(1)$$

4.4 Algoritmo di ricerca binaria per array ordinati

- Se l'array è ordinato, si può procedere dimezzando lo spazio di ricerca ogni volta. L'idea è di confrontare il valore cercato col valore dell'elemento che sta nella posizione di mezzo dell'array. Se i due valori sono diversi, si continua la ricerca solo nella metà dell'array che sta a sinistra (risp. destra) dell'elemento considerato se il suo valore è maggiore (risp. minore) di quello cercato:

```
int ricerca_binaria_array(int a[],
                          int n,
                          int valore)
{
  int sx,
      dx,
      mx;

  for (sx = 0, dx = n - 1, mx = (sx + dx) / 2;
       ((sx <= dx) && (a[mx] != valore));
       mx = (sx + dx) / 2)
    if (a[mx] > valore)
      dx = mx - 1;
    else
      sx = mx + 1;
  return((sx <= dx)?
           mx:
           -1);
}
```

- Il caso ottimo si verifica quando il valore cercato è presente nell'elemento che sta nella posizione di mezzo dell'array. In questo caso la complessità asintotica è $O(1)$ come per il caso ottimo dell'algoritmo di ricerca lineare.

- Nel caso pessimo lo spazio di ricerca viene ripetutamente diviso a metà fino a restare con un unico elemento da confrontare con il valore cercato. Denotato con k il numero di iterazioni, il quale coincide con il numero di dimezzamenti dello spazio di ricerca, nel caso pessimo $n/2^k = 1$ da cui $k = \log_2 n$. Di conseguenza la complessità è:
$$T(n) = 1 + k \cdot (1 + 1 + 1 + 1) + 1 = 4 \cdot k + 2 = 4 \cdot \log_2 n + 2 = O(\log n)$$

- L'algoritmo di ricerca binario è quindi più efficiente di quello di ricerca lineare. Tuttavia, esso è meno generale, in quanto può essere applicato solo ad array ordinati.

4.5 Criteri di confronto per algoritmi di ordinamento per array

- Gli algoritmi di ordinamento consentono di solito una più rapida ricerca di valori all'interno di un array attraverso le chiavi degli elementi. In generale, un elemento può contenere una chiave primaria e più chiavi secondarie, ciascuna delle quali viene considerata durante la ricerca quando la chiave primaria e le precedenti chiavi secondarie dell'elemento coincidono con quelle del valore cercato.

- Per noi le chiavi saranno numeri interi e la relazione d'ordine totale sarà $\leq$.

- Oltre che per il loro principio di funzionamento e per la loro efficienza, gli algoritmi di ordinamento possono essere confrontati in base ai seguenti criteri:

  – Stabilità: un algoritmo di ordinamento è stabile se non altera l'ordine relativo di elementi dell'array aventi la stessa chiave. Algoritmi di questo tipo evitano interferenze con ordinamenti pregressi dello stesso array basati su chiavi secondarie.

  – Sul posto: un algoritmo di ordinamento opera sul posto se la dimensione delle strutture ausiliarie di cui necessita è indipendente dal numero di elementi dell'array da ordinare. Algoritmi di questo tipo fanno un uso parsimonioso della memoria.

- È possibile dimostrare che il problema dell'ordinamento non può essere risolto con un algoritmo di complessità asintotica inferiore a quella pseudolineare. Formalmente, per ogni algoritmo che ordina un array di n elementi, il tempo d'esecuzione soddisfa $T(n) = \Omega(n \cdot \log n)$.

4.6 Insertsort

- Insertsort è un algoritmo di ordinamento iterativo che al generico passo i vede l'array diviso in una sequenza di destinazione $a[0], \ldots, a[i-1]$ già ordinata e una sequenza di origine $a[i], \ldots, a[n-1]$ ancora da ordinare. L'obiettivo è di inserire il valore contenuto in $a[i]$ al posto giusto nella sequenza di destinazione facendolo scivolare a ritroso, in modo da ridurre la sequenza di origine di un elemento:

```
void insertsort(int a[],
                int n)
{
  int valore_ins,
      i,
      j;

  for (i = 1;
       (i < n);
       i++)
  {
    for (valore_ins = a[i], j = i - 1;
         ((j >= 0) && (a[j] > valore_ins));
         j--)
      a[j + 1] = a[j];
```

```
    if (j + 1 != i)
      a[j + 1] = valore_ins;
  }
}
```

- Insertsort è stabile in quanto nel confronto tra `a[j]` e `valore_ins` viene usato `>` anziché `>=`.
- Insertsort opera sul posto in quanto usa soltanto una variabile aggiuntiva (`valore_ins`).
- Il caso ottimo si verifica quando l'array è già ordinato. In questo caso la complessità è:
$$T(n) = 1 + (n-1) \cdot (1 + 1 + 0 \cdot (1 + 1 + 1) + 1 + 1 + 1) + 1 = 5 \cdot n - 3 = O(n)$$
- Il caso pessimo si verifica quando l'array è inversamente ordinato. In questo caso il numero di iterazioni nell'istruzione `for` interna è proporzionale ad `i`, quindi la complessità è:
$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n-1} (1 + 1 + i \cdot (1 + 1 + 1) + 1 + 1 + 1 + 1) + 1 = 2 + (n-1) \cdot 6 + 3 \cdot \sum_{i=1}^{n-1} i \\ &= 6 \cdot n - 4 + 3 \cdot \tfrac{(n-1) \cdot n}{2} = 1.5 \cdot n^2 + 4.5 \cdot n - 4 = O(n^2) \end{aligned}$$
- Esempio: l'array

  42 38 11 75 99 23 84 67

  viene ordinato da insertsort attraverso le seguenti iterazioni (il valore sottolineato è quello da inserire al posto giusto nella sequenza di destinazione)

42	<u>38</u>	11	75	99	23	84	67	(1 scambio)
38	42	<u>11</u>	75	99	23	84	67	(2 scambi)
11	38	42	<u>75</u>	99	23	84	67	(0 scambi)
11	38	42	75	<u>99</u>	23	84	67	(0 scambi)
11	38	42	75	99	<u>23</u>	84	67	(4 scambi)
11	23	38	42	75	99	<u>84</u>	67	(1 scambio)
11	23	38	42	75	84	99	<u>67</u>	(3 scambi)
11	23	38	42	67	75	84	99	■fff_5

4.7 Selectsort

- Selectsort è un algoritmo di ordinamento iterativo che, come insertsort, al generico passo i vede l'array diviso in una sequenza di destinazione $a[0], \ldots, a[i-1]$ già ordinata e una sequenza di origine $a[i], \ldots, a[n-1]$ ancora da ordinare. L'obiettivo è di selezionare l'elemento della sequenza di origine che contiene il valore minimo e di scambiare tale valore con il valore contenuto in $a[i]$, in modo da ridurre la sequenza di origine di un elemento:

```
void selectsort(int a[],
                int n)
{
  int valore_min,
      indice_valore_min,
      i,
      j;

  for (i = 0;
       (i < n - 1);
       i++)
  {
    for (valore_min = a[i], indice_valore_min = i, j = i + 1;
         (j < n);
         j++)
```

```
      if (a[j] < valore_min)
      {
        valore_min = a[j];
        indice_valore_min = j;
      }
    if (indice_valore_min != i)
    {
      a[indice_valore_min] = a[i];
      a[i] = valore_min;
    }
  }
}
```

- Selectsort è stabile in quanto nel confronto tra `a[j]` e `valore_min` viene usato `<` anziché `<=`.
- Selectsort opera sul posto in quanto usa soltanto due variabili aggiuntive (`valore_min` e `indice_valore_min`).
- La complessità asintotica non dipende dalla disposizione iniziale dei valori negli elementi dell'array. Denotiamo con $h \in \{1, 3\}$ il tempo d'esecuzione di una delle due istruzioni `if`. Poiché il numero di iterazioni nell'istruzione `for` interna è proporzionale ad `n - i`, la complessità è:

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n-1} (1 + 1 + (n-i) \cdot (1 + h + 1) + 1 + h + 1) + 1 \\ &= 2 + (n-1) \cdot (h+4) + (n-1) \cdot (h+2) \cdot n - (h+2) \cdot \sum_{i=1}^{n-1} i \\ &= (h+2) \cdot n^2 + 2 \cdot n - (h+2) - (h+2) \cdot \tfrac{(n-1) \cdot n}{2} \\ &= (0.5 \cdot h + 1) \cdot n^2 + (0.5 \cdot h + 3) \cdot n - (h+2) = O(n^2) \end{aligned}$$

- Esempio: il solito array viene ordinato da selectsort attraverso le seguenti iterazioni (il valore sottolineato è quello da scambiare con il valore minimo della sequenza di origine)

<u>42</u>	38	11	75	99	23	84	67	(1 scambio)
11	<u>38</u>	42	75	99	23	84	67	(1 scambio)
11	23	<u>42</u>	75	99	38	84	67	(1 scambio)
11	23	38	<u>75</u>	99	42	84	67	(1 scambio)
11	23	38	42	<u>99</u>	75	84	67	(1 scambio)
11	23	38	42	67	<u>75</u>	84	99	(0 scambi)
11	23	38	42	67	75	<u>84</u>	99	(0 scambi)
11	23	38	42	67	75	84	99	

4.8 Bubblesort

- Bubblesort è un algoritmo di ordinamento iterativo che, come i due precedenti algoritmi, al generico passo i vede l'array diviso in una sequenza di destinazione $a[0], \ldots, a[i-1]$ già ordinata e una sequenza di origine $a[i], \ldots, a[n-1]$ ancora da ordinare. L'obiettivo è di far emergere (come se fosse una bollicina) il valore minimo della sequenza di origine confrontando e scambiando sistematicamente i valori di elementi adiacenti a partire dalla fine dell'array, in modo da ridurre la sequenza di origine di un elemento:

```
void bubblesort(int a[],
                int n)
{
  int tmp,
      i,
      j;
```

```
  for (i = 1;
       (i < n);
       i++)
    for (j = n - 1;
         (j >= i);
         j--)
      if (a[j] < a[j - 1])
      {
        tmp = a[j - 1];
        a[j - 1] = a[j];
        a[j] = tmp;
      }
}
```

- Bubblesort è stabile in quanto nel confronto tra `a[j]` e `a[j - 1]` viene usato `<` anziché `<=`.
- Bubblesort opera sul posto in quanto usa soltanto una variabile aggiuntiva (`tmp`).
- La complessità asintotica non dipende dalla disposizione iniziale dei valori negli elementi dell'array. Denotiamo con $h \in \{1, 4\}$ il tempo d'esecuzione dell'istruzione `if`. Poiché il numero di iterazioni nell'istruzione `for` interna è proporzionale ad `n - i`, la complessità è:

$$\begin{aligned} T(n) &= 1 + \sum_{i=1}^{n-1} (1 + 1 + (n-i) \cdot (1 + h + 1) + 1 + 1) + 1 \\ &= 2 + (n-1) \cdot 4 + (n-1) \cdot (h+2) \cdot n - (h+2) \cdot \sum_{i=1}^{n-1} i \\ &= (h+2) \cdot n^2 + (2-h) \cdot n - 2 - (h+2) \cdot \tfrac{(n-1) \cdot n}{2} \\ &= (0.5 \cdot h + 1) \cdot n^2 + (3 - 0.5 \cdot h) \cdot n - 2 = O(n^2) \end{aligned}$$

- Esempio: il solito array viene ordinato da bubblesort attraverso le seguenti iterazioni (il valore sottolineato è quello che emerge a seguito degli scambi)

42	38	11	75	99	23	84	67	(5 scambi)
<u>11</u>	42	38	23	75	99	67	84	(4 scambi)
11	<u>23</u>	42	38	67	75	99	84	(2 scambi)
11	23	<u>38</u>	42	67	75	84	99	(0 scambi)
11	23	38	<u>42</u>	67	75	84	99	(0 scambi)
11	23	38	42	<u>67</u>	75	84	99	(0 scambi)
11	23	38	42	67	<u>75</u>	84	99	(0 scambi)
11	23	38	42	67	75	<u>84</u>	99	

- A causa dell'elevato numero di scambi di valori vicini tra loro, bubblesort è il peggiore degli algoritmi di ordinamento visti finora. Le sue prestazioni possono essere migliorate evitando di procedere con le iterazioni rimanenti nel caso l'ultima non abbia dato luogo a nessuno scambio.

4.9 Mergesort

- Mergesort è un algoritmo di ordinamento ricorsivo proposto da Von Neumann nel 1945. Data una porzione di un array, mergesort la divide in due parti della stessa dimensione a cui applicare l'algoritmo stesso, poi fonde ordinatamente le due parti:

```
void mergesort(int a[],
               int sx,
               int dx)
{
  int mx;
```

```
  if (sx < dx)
  {
    mx = (sx + dx) / 2;
    mergesort(a,
              sx,
              mx);
    mergesort(a,
              mx + 1,
              dx);
    fondi(a,
          sx,
          mx,
          dx);
  }
}

void fondi(int a[],
           int sx,
           int mx,
           int dx)
{
  int *b,   /* array di appoggio */
      i,    /* indice per la parte sinistra di a (da sx ad m) */
      j,    /* indice per la parte destra di a (da m + 1 a dx) */
      k;    /* indice per la porzione di b da sx a dx */

  /* fondi ordinatamente le due parti finche' sono entrambe non vuote */
  b = (int *)calloc(dx + 1,
                    sizeof(int));
  for (i = sx, j = mx + 1, k = sx;
       ((i <= mx) && (j <= dx));
       k++)
    if (a[i] <= a[j])
    {
      b[k] = a[i];
      i++;
    }
    else
    {
      b[k] = a[j];
      j++;
    }

  /* copia i valori rimasti nella parte sinistra in fondo alla porzione di a */
  if (i <= mx)
    copia(a,
          i,
          mx,
          a,
          k,
          dx);
```

```
  /* copia i valori messi negli elementi di b nei corrispondenti elementi di a */
  copia(b,
        sx,
        k - 1,
        a,
        sx,
        k - 1);
  free(b);
}

void copia(int a1[],
           int sx1,
           int dx1,
           int a2[],
           int sx2,
           int dx2)
{
  int i1,
      i2;

  for (i1 = sx1, i2 = sx2;
       (i1 <= dx1);
       i1++, i2++)
    a2[i2] = a1[i1];
}
```

- Mergesort è stabile in quanto nel confronto tra `a[i]` e `a[j]` effettuato nella fusione ordinata viene usato `<=` anziché `<`.

- Diversamente dai tre algoritmi di ordinamento visti sinora, mergesort non opera sul posto in quanto nella fusione ordinata viene usato un array di appoggio (`b`) il cui numero di elementi è proporzionale al numero di elementi dell'array da ordinare.

- La complessità asintotica non dipende dalla disposizione iniziale dei valori negli elementi dell'array. Denotiamo con $n = dx - sx + 1$ la dimensione della porzione di array in esame. Poiché il tempo d'esecuzione dell'operazione di fusione ordinata è proporzionale ad n, il tempo d'esecuzione di mergesort è dato dalla relazione di ricorrenza lineare, di ordine non costante e con lavoro di combinazione lineare:

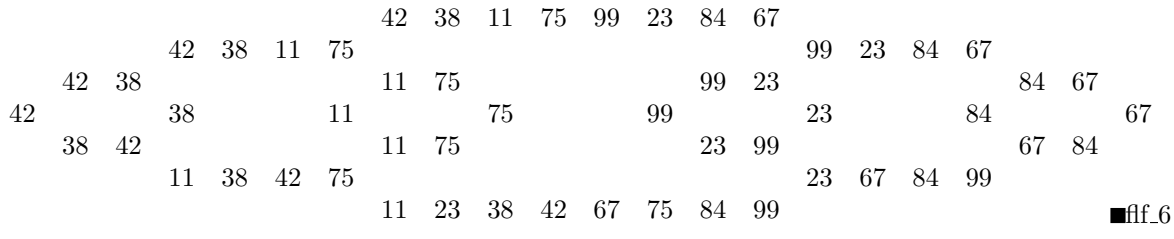
$$\begin{cases} T(n) &= 2 \cdot T(n/2) + (c_1 \cdot n + c_2) \quad \text{per } n > 1 \\ T(1) &= 1 \end{cases}$$

  la cui soluzione è:

$$T(n) = O(n \cdot \log n)$$

- Mergesort è quindi più efficiente dal punto di vista temporale dei precedenti tre algoritmi di ordinamento.

- Esempio: il solito array viene ordinato da mergesort nel seguente modo



■fff_6

4.10 Quicksort

- Quicksort è un algoritmo di ordinamento ricorsivo proposto da Hoare nel 1962. Data una porzione di un array e scelto un valore v detto pivot contenuto in quella porzione, quicksort divide la porzione in tre parti – la prima composta da elementi contenenti valori $\leq v$, la seconda (eventualmente vuota) composta da elementi contenenti valori $= v$, la terza composta da elementi contenenti valori $\geq v$ – poi applica l'algoritmo stesso alla prima e alla terza parte. Quicksort determina la tripartizione effettuando degli scambi di valori $\geq v$ incontrati a partire dall'inizio della porzione di array con valori $\leq v$ incontrati a partire dalla fine della porzione di array, in modo tale da spostare i primi verso la fine della porzione di array e gli ultimi verso l'inizio della porzione di array:

```
void quicksort(int a[],
               int sx,
               int dx)
{
  int pivot,
      tmp,
      i,
      j;

  /* crea la tripartizione */
  for (pivot = a[(sx + dx) / 2], i = sx, j = dx;
       (i <= j);
       )
  {
    while (a[i] < pivot)
      i++;
    while (a[j] > pivot)
      j--;
    if (i <= j)
    {
      if (i < j)
      {
        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
      }
      i++;
      j--;
    }
  }

  /* ordina la prima e la terza parte se contenenti piu' di un elemento */
  if (sx < j)
    quicksort(a,
              sx,
              j);
  if (i < dx)
    quicksort(a,
              i,
              dx);
}
```

- Diversamente dai quattro algoritmi di ordinamento visti sinora, quicksort non è stabile in quanto nel confronto tra `a[i]` e `pivot` viene usato `<` anziché `<=` e nel confronto tra `a[j]` e `pivot` viene usato `>` anziché `>=`. Questo è inevitabile al fine di ottenere una corretta tripartizione.

- Diversamente da mergesort, quicksort opera sul posto in quanto usa soltanto due variabili aggiuntive (`pivot` e `tmp`).

- Denotiamo con $n = dx - sx + 1$ la dimensione della porzione di array in esame. Poiché il tempo per la creazione della tripartizione è proporzionale ad n, il tempo d'esecuzione di quicksort è dato dalla relazione di ricorrenza lineare, di ordine non costante e con lavoro di combinazione lineare:
$$\begin{cases} T(n) &= T(n_1) + T(n_3) + (c_1 \cdot n + c_2) \quad \text{per } n > 1 \\ T(1) &= 11 \end{cases}$$
dove n_1 è il numero di elementi nella prima parte ed n_3 è il numero di elementi nella terza parte.

- Nel caso ottimo, come pivot viene sempre scelto il valore mediano della porzione di array, cosicché sia la prima parte che la terza parte contengono ciascuna un numero di elementi proporzionale ad $n/2$. Di conseguenza, in questo caso la relazione di ricorrenza diventa:
$$\begin{cases} T(n) &= 2 \cdot T(n/2) + (c_1 \cdot n + c_2) \quad \text{per } n > 1 \\ T(1) &= 11 \end{cases}$$
la cui soluzione è:
$$T(n) = O(n \cdot \log n)$$

- Nel caso pessimo, come pivot viene sempre scelto il valore minimo o massimo della porzione di array, cosicché una tra la prima parte e la terza parte contiene un elemento mentre l'altra contiene $n - 1$ elementi. Di conseguenza, in questo caso la relazione di ricorrenza diventa:
$$\begin{cases} T(n) &= \max\{T(q) + T(n-q) \mid 1 \le q \le n-1\} + (c_1 \cdot n + c_2) \quad \text{per } n > 1 \\ T(1) &= 11 \end{cases}$$
Applichiamo il metodo delle sostituzioni e proviamo che $T(n) \le c \cdot n^2$ per un opportuno valore di $c > 0$ procedendo per induzione su $n \ge 1$:

  - Sia $n = 1$. Risulta $T(1) = 11$ e $c \cdot 1^2 = c$, da cui l'asserto è vero per $n = 1$ scegliendo $c \ge 11$.
  - Consideriamo $n > 1$ e supponiamo $T(m) \le c \cdot m^2$ per ogni m tale che $1 \le m \le n-1$. Risulta $T(n) = \max\{T(q) + T(n-q) \mid 1 \le q \le n-1\} + (c_1 \cdot n + c_2) \le \max\{c \cdot q^2 + c \cdot (n-q)^2 \mid 1 \le q \le n-1\} + (c_1 \cdot n + c_2) = c \cdot \max\{q^2 + (n-q)^2 \mid 1 \le q \le n-1\} + (c_1 \cdot n + c_2) = c \cdot \max\{2 \cdot q^2 - 2 \cdot n \cdot q + n^2 \mid 1 \le q \le n-1\} + (c_1 \cdot n + c_2)$ per ipotesi induttiva. Poiché $2 \cdot q^2 - 2 \cdot n \cdot q + n^2$ è una funzione quadratica di q con derivata seconda positiva, graficamente essa è una parabola rivolta verso l'alto, quindi nell'intervallo $1 \le q \le n-1$ essa assume il valore massimo agli estremi dell'intervallo stesso. Pertanto $T(n) \le c \cdot n^2 - 2 \cdot c \cdot n + 2 \cdot c + (c_1 \cdot n + c_2) = c \cdot n^2 + (c_1 - 2 \cdot c) \cdot n + (c_2 + 2 \cdot c) \le c \cdot n^2$ per $n > 1$ scegliendo $c \ge (c_1 + c_2)/(2 \cdot n - 2)$.

  Il risultato è quindi verificato per $c \ge \max\{11, c_1 + c_2\}$ e di conseguenza nel caso pessimo si ha:
$$T(n) = O(n^2)$$

- Si può dimostrare che nel caso medio la complessità asintotica di quicksort è:
$$T(n) = O(n \cdot \log n)$$
Intuitivamente, se anche supponiamo che a seguito di ogni tripartizionamento la maggior parte degli elementi, diciamo il 90%, finiscano nella stessa parte, risulta comunque che il numero k di chiamate ricorsive cresce solo logaritmicamente con il numero n di elementi della porzione di array da ordinare. Infatti k soddisfa $n \cdot 0.90^k = 1$ da cui $k = \log_{10/9} n$. Quindi si ha un numero di chiamate ricorsive che cresce linearmente nel caso pessimo ma non nei casi che gli si avvicinano. Pertanto la probabilità di avere un numero di chiamate ricorsive che cresce linearmente è estremamente bassa.

- Quicksort è dunque più efficiente in media di insertsort, selectsort e bubblesort e ha prestazioni peggiori di mergesort solo nel caso pessimo. Da verifiche sperimentali risulta che in pratica quicksort è il più efficiente algoritmo di ordinamento (per array non troppo piccoli).

- Esempio: il solito array viene ordinato da quicksort nel seguente modo (il valore sottolineato è il pivot della porzione di array)

							42	38	11	<u>75</u>	99	23	84	67						
	42	38	<u>11</u>	67	23											99	<u>84</u>	75		
11						38	<u>42</u>	67	23						75		84		99	
11				<u>38</u>	23					<u>67</u>	42				75		84		99	
11			23			38			42			67			75		84		99	

4.11 Heapsort

- Heapsort è un algoritmo di ordinamento iterativo proposto da Williams nel 1964, il quale si basa su una struttura dati chiamata heap.
- Una sequenza di valori $h_1, \ldots, h_n$ costituisce uno heap se e solo se per ogni $i = 1, \ldots, n/2$ risulta $h_i \geq h_{2 \cdot i}$ e $h_i \geq h_{2 \cdot i+1}$.
- Se immaginiamo i valori di uno heap come disposti in un albero binario (v. Sez. 6.1) nel quale il nodo i contiene h_i, suo figlio sinistro contiene $h_{2 \cdot i}$ e suo figlio destro contiene $h_{2 \cdot i+1}$, allora la radice di ogni sottoalbero contiene il valore massimo presente nel sottoalbero stesso. In particolare, la radice dell'intero albero contiene il valore massimo dell'intero heap.
- Heapsort opera in due fasi:
  - Nella prima fase, heapsort permuta i valori contenuti negli elementi dell'array in modo tale che la nuova disposizione delle chiavi costituisca uno heap. Ciò viene ottenuto facendo scivolare al posto giusto i valori contenuti negli elementi $a[n/2], \ldots, a[1]$ considerati in questo ordine:

```
void heapsort(int a[],   /* a[0] non viene usato */
              int n)
{
  int tmp,
      sx,
      dx;

  /* prima fase: trasformare l'array in uno heap */
  for (sx = n / 2;
       (sx >= 1);
       sx--)
    setaccia_heap(a,
                  sx,
                  n);
```

  - Nel generico passo i della seconda fase, heapsort vede l'array diviso in una sequenza di origine $a[1], \ldots, a[i-1]$ organizzata ad heap e una sequenza di destinazione $a[i], \ldots, a[n]$ già ordinata. Poiché $a[1]$ contiene il valore massimo presente nella sequenza di origine, il valore di $a[1]$ viene scambiato con il valore di $a[i-1]$ in modo da metterlo subito prima dell'inizio della sequenza di destinazione. Facendo poi scivolare al posto giusto il nuovo valore di $a[1]$, si garantisce che le chiavi dei valori contenuti negli elementi della sequenza di origine ridotta di un elemento costituiscano ancora uno heap:

```
  /* seconda fase: ordinare l'array mantenendo l'organizzazione ad heap */
  for (dx = n;
       (dx > 1);
       dx--)
```

```
    {
      tmp = a[1];
      a[1] = a[dx];
      a[dx] = tmp;
      setaccia_heap(a,
                    1,
                    dx - 1);
    }
  }
```

- L'algoritmo per far scivolare un nuovo valore al posto giusto di uno heap, proposto da Floyd nel 1964, pone il nuovo valore nella radice dello heap e poi lo fa scendere finché necessario scambiandolo ad ogni passo con il valore massimo tra quelli contenuti nei due nodi figli se tale valore massimo è maggiore del nuovo valore:

```
void setaccia_heap(int a[],
                   int sx,
                   int dx)
{
  int nuovo_valore,
      i,
      j;

  for (nuovo_valore = a[sx], i = sx, j = 2 * i;
       (j <= dx);
       )
  {
    if ((j < dx) && (a[j + 1] > a[j]))
      j++;
    if (nuovo_valore < a[j])
    {
      a[i] = a[j];
      i = j;
      j = 2 * i;
    }
    else
      j = dx + 1;
  }
  if (i != sx)
    a[i] = nuovo_valore;
}
```

- Come quicksort, heapsort non è stabile in quanto i ripetuti setacciamenti possono alterare in modo impredicibile l'ordine relativo di elementi dell'array aventi la stessa chiave.

- Come tutti i precedenti algoritmi eccetto mergesort, heapsort opera sul posto in quanto usa soltanto due variabili aggiuntive (`tmp` e `nuovo_valore`).

- La complessità asintotica non dipende dalla disposizione iniziale dei valori negli elementi dell'array. Poiché il tempo d'esecuzione dell'operazione di setacciamento è proporzionale all'altezza dell'albero binario associato allo heap, e questa altezza è proporzionale a $\log_2 n$ essendo l'albero bilanciato (v. Sez. 6.4), la complessità è:

$$\begin{aligned} T(n) &\leq 1 + \tfrac{n}{2} \cdot (1 + \log_2 n + 1) + 1 + 1 + (n-1) \cdot (1 + 1 + 1 + 1 + \log_2 n + 1) + 1 \\ &= 1.5 \cdot n \cdot \log_2 n + 6 \cdot n - \log_2 n - 1 = O(n \cdot \log n) \end{aligned}$$

- Heapsort ha dunque le stesse proprietà di quicksort, ma ne evita il degradamento quadratico.
- Esempio: il solito l'array viene ordinato da heapsort attraverso le seguenti iterazioni (il valore sottolineato è quello da far scivolare al posto giusto nello heap)

42	38	11	<u>75</u>	99	23	84	67	(prima fase)
42	38	<u>11</u>	75	99	23	84	67	(prima fase)
42	<u>38</u>	84	75	99	23	11	67	(prima fase)
<u>42</u>	99	84	75	38	23	11	67	(prima fase)
99	75	84	67	38	23	11	<u>42</u>	(seconda fase)
84	75	42	67	38	23	<u>11</u>	99	(seconda fase)
75	67	42	11	38	<u>23</u>	84	99	(seconda fase)
67	38	42	11	<u>23</u>	75	84	99	(seconda fase)
42	38	23	<u>11</u>	67	75	84	99	(seconda fase)
38	11	<u>23</u>	42	67	75	84	99	(seconda fase)
23	<u>11</u>	38	42	67	75	84	99	(seconda fase)
11	23	38	42	67	75	84	99	

■fff_7

Capitolo 5

Algoritmi per liste

5.1 Liste: definizioni di base e problemi classici

- Si dice lista una tripla $L = (E, t, \mathcal{S})$ dove E è un insieme di elementi, $t \in E$ è detto testa ed $\mathcal{S}$ è una relazione binaria su E – cioè $\mathcal{S} \subseteq E \times E$ – che soddisfa le seguenti proprietà:
  - Per ogni $e \in E$, $(e, t) \notin \mathcal{S}$.
  - Per ogni $e \in E$, se $e \neq t$ allora esiste uno ed un solo $e' \in E$ tale che $(e', e) \in \mathcal{S}$.
  - Per ogni $e \in E$, esiste al più un $e' \in E$ tale che $(e, e') \in \mathcal{S}$.
  - Per ogni $e \in E$, se $e \neq t$ allora e è raggiungibile da t, cioè esistono $e'_1, \ldots, e'_k \in E$ con $k \geq 2$ tali che $e'_1 = t$, $(e'_i, e'_{i+1}) \in \mathcal{S}$ per ogni $1 \leq i \leq k-1$, ed $e'_k = e$.
- Una lista $L = (E, t, \mathcal{S})$ è detta ordinata se le chiavi contenute nei suoi elementi sono disposte in modo tale da soddisfare una relazione d'ordine totale: per ogni $e_1, e_2 \in E$, se $(e_1, e_2) \in \mathcal{S}$ allora la chiave di e_1 precede la chiave di e_2 nella relazione d'ordine totale.
- Una lista viene rappresentata come una struttura dati dinamica lineare, in cui ogni elemento contiene solo l'indirizzo dell'elemento successivo (lista singolarmente collegata) oppure anche l'indirizzo dell'elemento precedente (lista doppiamente collegata):

```
typedef struct elem_lista
{
  int               valore;
  struct elem_lista *succ_p;
} elem_lista_t;
```

```
typedef struct elem_lista_dc
{
  int                  valore;
  struct elem_lista_dc *succ_p, *prec_p;
} elem_lista_dc_t;
```

- L'indirizzo dell'elemento successivo contenuto nell'ultimo elemento di una lista è indefinito, così come l'indirizzo dell'elemento precedente contenuto nel primo elemento di una lista doppiamente collegata. Fa eccezione il caso dell'implementazione circolare di una lista, nella quale l'ultimo elemento è collegato al primo elemento.
- Gli elementi di una lista non sono necessariamente memorizzati in modo consecutivo, quindi l'accesso ad un qualsiasi elemento avviene scorrendo tutti gli elementi che lo precedono. Questo accesso indiretto necessita dell'indirizzo del primo elemento della lista, detto testa, il quale è indefinito se e solo se la lista è vuota.
- Problema della visita: data una lista, attraversare tutti i suoi elementi esattamente una volta.
- Problema della ricerca: dati una lista e un valore, stabilire se il valore è contenuto in un elemento della lista, riportando in caso affermativo l'indirizzo di tale elemento.

- Problema dell'inserimento: dati una lista e un valore, inserire (se possibile) nella posizione appropriata della lista un nuovo elemento in cui memorizzare il valore.
- Problema della rimozione: dati una lista e un valore, rimuovere (se esiste) l'elemento appropriato della lista che contiene il valore.

5.2 Algoritmi di visita, ricerca, inserimento e rimozione per liste

- Algoritmo di visita di una lista:

```
void visita_lista(elem_lista_t *testa_p)
{
  elem_lista_t *elem_p;

  for (elem_p = testa_p;
       (elem_p != NULL);
       elem_p = elem_p->succ_p)
    elabora(elem_p->valore);
}
```

  Se la lista ha n elementi e l'elaborazione del valore di ogni elemento si svolge in al più d passi, in ogni caso la complessità è:
  $$T(n) = 1 + n \cdot (1 + d + 1) + 1 = (d + 2) \cdot n + 2 = O(n)$$

- Algoritmo di ricerca di un valore in una lista (restituisce l'indirizzo dell'elemento che contiene il valore se presente, `NULL` se il valore è assente):

```
elem_lista_t *cerca_in_lista(elem_lista_t *testa_p,
                             int          valore)
{
  elem_lista_t *elem_p;

  for (elem_p = testa_p;
       ((elem_p != NULL) && (elem_p->valore != valore));
       elem_p = elem_p->succ_p);
  return(elem_p);
}
```

  Se la lista ha n elementi, nel caso pessimo (valore non presente nella lista) la complessità è:
  $$T(n) = 1 + n \cdot (1 + 1) + 1 = 2 \cdot n + 2 = O(n)$$
  mentre nel caso ottimo (valore presente nel primo elemento della lista) la complessità è:
  $$T(n) = 1 + 1 = 2 = O(1)$$

- L'algoritmo di inserimento di un valore in una lista varia a seconda della posizione in cui l'operazione deve essere effettuata. Gli inserimenti nelle estremità della lista sono trattati nelle Sezz. 5.3 e 5.4. Qui consideriamo l'algoritmo di inserimento in una lista ordinata contenente valori distinti (restituisce 1 se l'inserimento può avere luogo, 0 altrimenti):

```
int inserisci_in_lista_ordinata(elem_lista_t **testa_p,
                                int          valore)
{
  int          inserito;
  elem_lista_t *corr_p,
               *prec_p,
               *nuovo_p;
```

```
  for (corr_p = prec_p = *testa_p;
       ((corr_p != NULL) && (corr_p->valore < valore));
       prec_p = corr_p, corr_p = corr_p->succ_p);
  if ((corr_p != NULL) && (corr_p->valore == valore))
    inserito = 0;
  else
  {
    inserito = 1;
    nuovo_p = (elem_lista_t *)malloc(sizeof(elem_lista_t));
    nuovo_p->valore = valore;
    nuovo_p->succ_p = corr_p;
    if (corr_p == *testa_p)
      *testa_p = nuovo_p;
    else
      prec_p->succ_p = nuovo_p;
  }
  return(inserito);
}
```

Se la lista ha n elementi, nel caso pessimo (inserimento effettuato alla fine della lista) la complessità è:

$$T(n) = 1 + n \cdot (1 + 1) + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 2 \cdot n + 9 = O(n)$$

mentre nel caso ottimo (valore presente nel primo elemento della lista) la complessità è:

$$T(n) = 1 + 1 + 1 + 1 = 4 = O(1)$$

- Anche l'algoritmo di rimozione di un valore da una lista varia a seconda della posizione in cui l'operazione deve essere effettuata. Le rimozioni dalle estremità della lista sono trattate nelle Sezz. 5.3 e 5.4. Qui consideriamo l'algoritmo di rimozione da una lista ordinata contenente valori distinti (restituisce 1 se la rimozione può avere luogo, 0 altrimenti):

```
int rimuovi_da_lista_ordinata(elem_lista_t **testa_p,
                              int          valore)
{
  int          rimosso;
  elem_lista_t *corr_p,
               *prec_p;

  for (corr_p = prec_p = *testa_p;
       ((corr_p != NULL) && (corr_p->valore < valore));
       prec_p = corr_p, corr_p = corr_p->succ_p);
  if ((corr_p == NULL) || (corr_p->valore > valore))
    rimosso = 0;
  else
  {
    rimosso = 1;
    if (corr_p == *testa_p)
      *testa_p = corr_p->succ_p;
    else
      prec_p->succ_p = corr_p->succ_p;
    free(corr_p);
  }
  return(rimosso);
}
```

Se la lista ha n elementi, nel caso pessimo (rimozione effettuata alla fine della lista) la complessità è:

$$T(n) = 1 + n \cdot (1 + 1) + 1 + 1 + 1 + 1 + 1 + 1 = 2 \cdot n + 7 = O(n)$$

mentre nel caso ottimo (lista vuota oppure valore maggiore presente nel primo elemento della lista) la complessità è:

$$T(n) = 1 + 1 + 1 + 1 = 4 = O(1)$$

5.3 Algoritmi di inserimento e rimozione per code

- Una coda è una lista governata in base al principio FIFO (first in, first out): gli inserimenti hanno tutti luogo presso la stessa estremità della lista, mentre le rimozioni hanno tutte luogo presso l'estremità opposta della lista.
- Per agevolare le operazioni di inserimento e rimozione, una coda è individuata tramite l'indirizzo del suo primo elemento (detto uscita) e l'indirizzo del suo ultimo elemento (detto ingresso), i quali sono indefiniti se e solo se la coda è vuota.
- Algoritmo di inserimento di un valore in una coda:

```
void metti_in_coda(elem_lista_t **uscita_p,
                   elem_lista_t **ingresso_p,
                   int          valore)
{
  elem_lista_t *nuovo_p;

  nuovo_p = (elem_lista_t *)malloc(sizeof(elem_lista_t));
  nuovo_p->valore = valore;
  nuovo_p->succ_p = NULL;
  if (*ingresso_p != NULL)
    (*ingresso_p)->succ_p = nuovo_p;
  else
    *uscita_p = nuovo_p;
  *ingresso_p = nuovo_p;
}
```

  Se la coda ha n elementi, la complessità è:

$$T(n) = 1 + 1 + 1 + 1 + 1 + 1 = 6 = O(1)$$

- Algoritmo di rimozione di un valore da una coda (restituisce l'indirizzo del primo elemento della coda):

```
elem_lista_t *togli_da_coda(elem_lista_t **uscita_p,
                            elem_lista_t **ingresso_p)
{
  elem_lista_t *elem_p;

  elem_p = *uscita_p;
  if (*uscita_p != NULL)
  {
    *uscita_p = (*uscita_p)->succ_p;
    if (*uscita_p == NULL)
      *ingresso_p = NULL;
  }
  return(elem_p);
}
```

  Se la coda ha n elementi, nel caso pessimo (coda di un elemento) la complessità è:

$$T(n) = 1 + 1 + 1 + 1 + 1 = 5 = O(1)$$

  mentre nel caso ottimo (coda vuota) la complessità è:

$$T(n) = 1 + 1 = 2 = O(1)$$

5.4 Algoritmi di inserimento e rimozione per pile

- Una pila è una lista governata in base al principio LIFO (last in, first out): gli inserimenti e le rimozioni hanno tutti luogo presso la stessa estremità della lista.
- Come una generica lista, una pila è individuata tramite l'indirizzo del suo primo elemento (detto cima), il quale è indefinito se e solo se la pila è vuota.
- Algoritmo di inserimento di un valore in una pila:

```
void metti_su_pila(elem_lista_t **cima_p,
                   int          valore)
{
  elem_lista_t *nuovo_p;

  nuovo_p = (elem_lista_t *)malloc(sizeof(elem_lista_t));
  nuovo_p->valore = valore;
  nuovo_p->succ_p = *cima_p;
  *cima_p = nuovo_p;
}
```

  Se la pila ha n elementi, la complessità è:
  $$T(n) = 1 + 1 + 1 + 1 = 4 = O(1)$$

- Algoritmo di rimozione di un valore da una pila (restituisce l'indirizzo del primo elemento della pila):

```
elem_lista_t *togli_da_pila(elem_lista_t **cima_p)
{
  elem_lista_t *elem_p;

  elem_p = *cima_p;
  if (*cima_p != NULL)
    *cima_p = (*cima_p)->succ_p;
  return(elem_p);
}
```

  Se la pila ha n elementi, nel caso pessimo (pila non vuota) la complessità è:
  $$T(n) = 1 + 1 + 1 = 3 = O(1)$$
  mentre nel caso ottimo (pila vuota) la complessità è:
  $$T(n) = 1 + 1 = 2 = O(1)$$

■fff_8

Capitolo 6

Algoritmi per alberi

6.1 Alberi: definizioni di base e problemi classici

- Si dice albero con radice, o semplicemente albero, una tripla $T = (N, r, \mathcal{B})$ dove N è un insieme di nodi, $r \in N$ è detto radice e $\mathcal{B}$ è una relazione binaria su N che soddisfa le seguenti proprietà:
  - Per ogni $n \in N$, $(n, r) \notin \mathcal{B}$.
  - Per ogni $n \in N$, se $n \neq r$ allora esiste uno ed un solo $n' \in N$ tale che $(n', n) \in \mathcal{B}$.
  - Per ogni $n \in N$, se $n \neq r$ allora n è raggiungibile da r, cioè esistono $n'_1, \ldots, n'_k \in N$ con $k \geq 2$ tali che $n'_1 = r$, $(n'_i, n'_{i+1}) \in \mathcal{B}$ per ogni $1 \leq i \leq k-1$, ed $n'_k = n$.
- Nella definizione di albero, la seconda e la terza proprietà di $\mathcal{B}$ sono equivalenti alla seguente proprietà:
  - Per ogni $n \in N$, se $n \neq r$ allora esistono e sono unici $n'_1, \ldots, n'_k \in N$ con $k \geq 2$ tali che $n'_1 = r$, $(n'_i, n'_{i+1}) \in \mathcal{B}$ per ogni $1 \leq i \leq k-1$, ed $n'_k = n$.
- La lista è un caso particolare di albero in cui esiste un unico percorso, il quale parte dalla radice e attraversa tutti i nodi senza effettuare diramazioni.
- Siano $T = (N, r, \mathcal{B})$ un albero ed $n \in N$. Si dice sottoalbero generato da n l'albero $T' = (N', n, \mathcal{B}')$ dove N' è il sottoinsieme dei nodi di N raggiungibili da n e $\mathcal{B}' = \mathcal{B} \cap (N' \times N')$.
- Sia $T = (N, r, \mathcal{B})$ un albero e siano $T_1 = (N_1, n_1, \mathcal{B}_1)$ e $T_2 = (N_2, n_2, \mathcal{B}_2)$ i sottoalberi generati da $n_1, n_2 \in N$. Allora $N_1 \cap N_2 = \emptyset$ oppure $N_1 \subseteq N_2$ oppure $N_2 \subseteq N_1$.
- Per meglio comprendere la caratteristiche di un albero, consideriamo una sua variante priva di radice.
- Si dice albero libero una coppia $T = (N, \mathcal{B})$ dove N è un insieme di nodi e $\mathcal{B}$ è una relazione binaria su N che soddisfa le seguenti proprietà:
  - Antiriflessività: per ogni $n \in N$, $(n, n) \notin \mathcal{B}$.
  - Simmetria: per ogni $n_1, n_2 \in N$, se $(n_1, n_2) \in \mathcal{B}$ allora anche $(n_2, n_1) \in \mathcal{B}$.
  - Connettività: per ogni $n_1, n_2 \in N$ diversi tra loro, esistono $n'_1, \ldots, n'_k \in N$ con $k \geq 2$ tali che $n'_1 = n_1$, $(n'_i, n'_{i+1}) \in \mathcal{B}$ per ogni $1 \leq i \leq k-1$, ed $n'_k = n_2$.
  - Aciclicità: per ogni $n \in N$, non esistono $n'_1, \ldots, n'_k \in N$ distinti con $k \geq 2$ tali che $(n, n'_1) \in \mathcal{B}$, $(n'_i, n'_{i+1}) \in \mathcal{B}$ per ogni $1 \leq i \leq k-1$, ed $(n'_k, n) \in \mathcal{B}$.
- Sia $T = (N, \mathcal{B})$ una coppia in cui $\mathcal{B}$ è una relazione binaria su N antiriflessiva e simmetrica. Le seguenti affermazioni sono equivalenti:
  - T è un albero libero.

  - Per ogni $n_1, n_2 \in N$ diversi tra loro, esistono e sono unici $n'_1, \ldots, n'_k \in N$ con $k \geq 2$ tali che $n'_1 = n_1$, $(n'_i, n'_{i+1}) \in \mathcal{B}$ per ogni $1 \leq i \leq k-1$, ed $n'_k = n_2$.
  - $\mathcal{B}$ soddisfa la proprietà di connettività ed è minimale, cioè la rimozione di una coppia di nodi da $\mathcal{B}$ fa perdere la connettività.
  - $\mathcal{B}$ soddisfa la proprietà di aciclicità ed è massimale, cioè l'aggiunta di una coppia di nodi a $\mathcal{B}$ fa perdere la aciclicità.
  - N e $\mathcal{B}$ sono finiti con $\mathcal{B}$ che soddisfa la proprietà di connettività e $|\mathcal{B}| = |N| - 1$.
  - N e $\mathcal{B}$ sono finiti con $\mathcal{B}$ che soddisfa la proprietà di aciclicità e $|\mathcal{B}| = |N| - 1$.

- Un albero libero può essere trasformato in un albero con radice designando uno dei suoi nodi come radice e rendendo la sua relazione asimmetrica in modo coerente con la radice scelta.

- Sia $T = (N, r, \mathcal{B})$ un albero:

  - Se $(n, n') \in \mathcal{B}$, allora n è detto padre di n' ed n' è detto figlio di n.
  - Se $(n, n_1), (n, n_2) \in \mathcal{B}$, allora n_1 ed n_2 sono detti fratelli.
  - I nodi privi di figli sono detti nodi esterni o foglie, mentre tutti gli altri nodi sono detti nodi interni.
  - Gli elementi di $\mathcal{B}$ sono detti rami.

- Sia $T = (N, r, \mathcal{B})$ un albero:

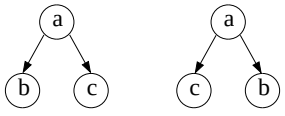
  - Si dice grado di T il massimo numero di figli di un nodo di T:
    $$d(T) = \max_{n \in N} |\{n' \in N \mid (n, n') \in \mathcal{B}\}|$$
  - Si dice che:
    * r è al livello 1.
    * Se $n \in N$ è al livello i ed $(n, n') \in \mathcal{B}$, allora n' è al livello $i + 1$.
  - Si dice altezza o profondità di T il massimo numero di nodi che si attraversano nel percorso di T che va dalla radice alla foglia più distante:
    $$h(T) = \max\{i \in \mathbb{N} \mid \exists n \in N.\ \ n \text{ è al livello } i\}$$
  - Si dice larghezza o ampiezza di T il massimo numero di nodi di T che si trovano allo stesso livello:
    $$b(T) = \max_{1 \leq i \leq h(T)} |\{n \in N \mid n \text{ è al livello } i\}|$$

- Un albero di grado $d > 1$ e altezza h può contenere un massimo numero di nodi pari a:
  $$n(d, h) = \sum_{i=0}^{h-1} d^i = \tfrac{d^h - 1}{d - 1}$$
  essendo d^i il massimo numero di nodi che possono trovarsi al livello i.

- Un albero $T = (N, r, \mathcal{B})$ è detto ordinato se per ogni nodo di T le chiavi contenute nei suoi figli soddisfano una relazione d'ordine totale.

- L'ordinamento si riflette a livello grafico riportando ordinatamente i figli di ciascun nodo da sinistra a destra.

- Esempio: i due seguenti alberi

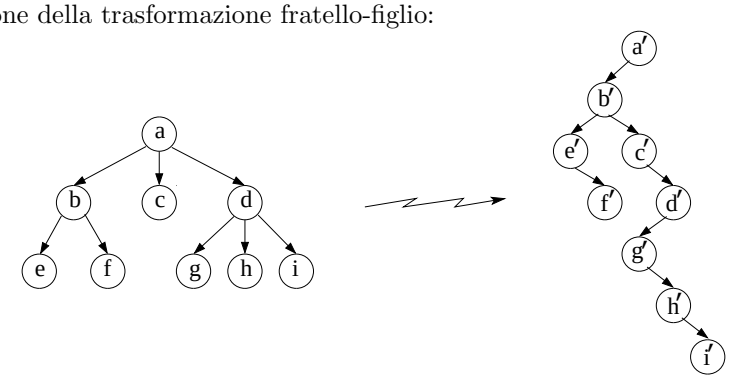


  sono diversi se considerati come ordinati.

- Un albero $T = (N, r, \mathcal{B})$ è detto binario se:
  - $\mathcal{B} = \mathcal{B}_{sx} \cup \mathcal{B}_{dx}$.
  - $\mathcal{B}_{sx} \cap \mathcal{B}_{dx} = \emptyset$.
  - Per ogni $n, n_1, n_2 \in N$, se $(n, n_1) \in \mathcal{B}_{sx}$ (risp. $\mathcal{B}_{dx}$) ed $(n, n_2) \in \mathcal{B}_{sx}$ (risp. $\mathcal{B}_{dx}$), allora $n_1 = n_2$.

  Se $(n, n') \in \mathcal{B}_{sx}$ (risp. $\mathcal{B}_{dx}$), allora n' è detto figlio sinistro (risp. destro) di n.
- Il fatto che un nodo è figlio sinistro (risp. destro) di un altro nodo si riflette a livello grafico riportando il nodo figlio a sinistra (risp. a destra) del nodo padre.
- Se un nodo di un albero binario ha un solo figlio, quest'ultimo può essere classificato come figlio sinistro oppure come figlio destro, mentre ciò non è possibile nel caso di un albero ordinato.
- Esempio: i due seguenti alberi

  sono diversi se considerati come binari.
- Nel seguito considereremo soltanto alberi binari, in quanto ogni albero non binario è equivalente ad un albero binario ottenuto applicando la seguente trasformazione fratello-figlio ad ogni nodo n dell'albero originario avente come figli i nodi $n_1, \ldots, n_k$:
  - Creare i nuovi nodi $n', n'_1, \ldots, n'_k$.
  - Mettere n'_1 come figlio sinistro di n'.
  - Per ogni $i = 1, \ldots, k-1$, mettere n'_{i+1} come figlio destro di n'_i.
- Esempio di applicazione della trasformazione fratello-figlio:



- Un albero binario viene rappresentato come una struttura dati dinamica gerarchica, in cui ogni elemento contiene l'indirizzo del suo figlio sinistro (indefinito se assente) e l'indirizzo del suo figlio destro (indefinito se assente):

```
typedef struct nodo_albero_bin
{
  int                    valore;
  struct nodo_albero_bin *sx_p, *dx_p;
} nodo_albero_bin_t;
```

- I nodi di un albero binario non sono necessariamente memorizzati in modo consecutivo. L'accesso ad ognuno di essi necessita dell'indirizzo del primo nodo dell'albero, detto radice, il quale è indefinito se e solo se l'albero è vuoto. L'accesso al nodo in questione avviene scorrendo tutti i nodi che precedono il nodo lungo l'unico percorso che congiunge il primo nodo dell'albero al nodo dato.
- I problemi classici per gli alberi sono gli stessi delle liste: visita, ricerca, inserimento e rimozione. ■fff_9

6.2 Algoritmi di visita e ricerca per alberi binari

- Mentre per una lista l'ordine nel quale visitare i suoi elementi non può che essere quello stabilito dalla sequenza in cui si trovano disposti gli elementi stessi, nel caso degli alberi binari sono possibili più alternative tra cui la visita in ordine anticipato, la visita in ordine simmetrico e la visita in ordine posticipato.

- Nella visita in ordine anticipato, prima si elabora il valore contenuto nel nodo al quale si è giunti, poi si visitano con lo stesso algoritmo il sottoalbero sinistro e il sottoalbero destro del nodo:

```
void visita_albero_bin_ant(nodo_albero_bin_t *nodo_p)
{
  if (nodo_p != NULL)
  {
    elabora(nodo_p->valore);
    visita_albero_bin_ant(nodo_p->sx_p);
    visita_albero_bin_ant(nodo_p->dx_p);
  }
}
```

- Nella visita in ordine simmetrico, prima si visita con lo stesso algoritmo il sottoalbero sinistro del nodo al quale si è giunti, poi si elabora il valore contenuto nel nodo e infine si visita con lo stesso algoritmo il sottoalbero destro del nodo:

```
void visita_albero_bin_simm(nodo_albero_bin_t *nodo_p)
{
  if (nodo_p != NULL)
  {
    visita_albero_bin_simm(nodo_p->sx_p);
    elabora(nodo_p->valore);
    visita_albero_bin_simm(nodo_p->dx_p);
  }
}
```

- Nella visita in ordine posticipato, prima si visitano con lo stesso algoritmo il sottoalbero sinistro e il sottoalbero destro del nodo al quale si è giunti, poi si elabora il valore contenuto nel nodo:

```
void visita_albero_bin_post(nodo_albero_bin_t *nodo_p)
{
  if (nodo_p != NULL)
  {
    visita_albero_bin_post(nodo_p->sx_p);
    visita_albero_bin_post(nodo_p->dx_p);
    elabora(nodo_p->valore);
  }
}
```

- Indicato con n il numero di nodi dell'albero binario e con d il numero massimo di passi eseguiti durante l'elaborazione del valore contenuto in un nodo, il tempo d'esecuzione di ciascuno dei tre algoritmi di visita riportati sopra è dato dalla relazione di ricorrenza lineare, di ordine non costante e con lavoro di combinazione costante:

$$\begin{cases} T(n) &= 1 + T(k) + T(n-1-k) + d = T(k) + T(n-1-k) + (d+1) \quad \text{per } n > 0 \\ T(0) &= 1 \end{cases}$$

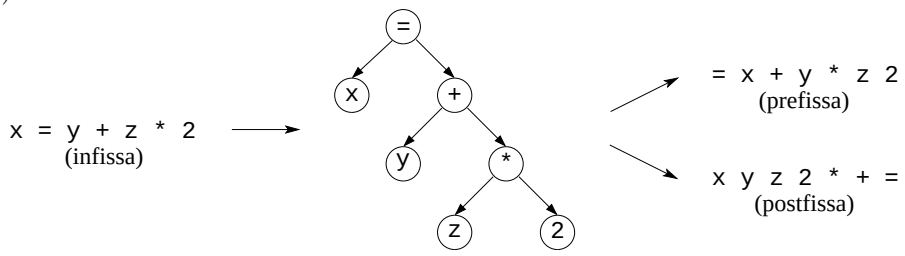
dove k è il numero di nodi del sottoalbero sinistro del nodo al quale si è giunti ed $n-1-k$ è il numero di nodi del suo sottoalbero destro. Applichiamo il metodo delle sostituzioni e proviamo che $T(n) = (d+2) \cdot n + 1$ procedendo per induzione su $n \geq 0$:

- Sia $n = 0$. Risulta $T(0) = 1$ e $(d+2) \cdot 0 + 1 = 1$, da cui l'asserto è vero per $n = 0$.
- Consideriamo $n > 0$ e supponiamo $T(n') = (d+2) \cdot n' + 1$ per ogni n' tale che $0 \leq n' \leq n-1$. Risulta $T(n) = T(k)+T(n-1-k)+(d+1) = (d+2) \cdot k+1+(d+2) \cdot (n-1-k)+1+(d+1)$ per ipotesi induttiva. Pertanto $T(n) = (d+2) \cdot (n-1)+(d+3) = (d+2) \cdot n-(d+2)+(d+3) = (d+2) \cdot n+1$.

Di conseguenza si ha:

$$T(n) = O(n)$$

- Esempio: dato l'albero di valutazione di un'espressione aritmetico-logica, la visita di tale albero in ordine anticipato (risp. posticipato) determina la trasposizione dell'espressione in notazione prefissa (risp. postfissa)



- I tre precedenti algoritmi di visita possono essere utilizzati anche come algoritmi di ricerca se opportunamente modificati. Ad esempio, dall'algoritmo di visita in ordine anticipato si ottiene il seguente algoritmo di ricerca:

```
nodo_albero_bin_t *cerca_in_albero_bin_ant(nodo_albero_bin_t *nodo_p,
                                           int               valore)
{
  nodo_albero_bin_t *nodo_ris_p;

  if ((nodo_p == NULL) || (nodo_p->valore == valore))
    nodo_ris_p = nodo_p;
  else
  {
    nodo_ris_p = cerca_in_albero_bin_ant(nodo_p->sx_p,
                                         valore);
    if (nodo_ris_p == NULL)
      nodo_ris_p = cerca_in_albero_bin_ant(nodo_p->dx_p,
                                           valore);
  }
  return(nodo_ris_p);
}
```

Se l'albero binario ha n nodi, nel caso pessimo (albero binario non vuoto e valore non presente nell'albero binario) la complessità è uguale a quella dell'algoritmo di visita in quanto bisogna attraversare tutti i nodi:

$$T(n) = O(n)$$

mentre nel caso ottimo (albero binario vuoto oppure valore presente nella radice) la complessità è:

$$T(n) = 1 + 1 = 2 = O(1)$$

6.3 Algoritmi di ricerca, inserimento e rimozione per alberi binari di ricerca

- Per un array di n elementi abbiamo visto che nel caso pessimo la ricerca di un valore dato avviene in modo più efficiente – $O(\log n)$ anziché $O(n)$ – se l'array è ordinato. Per quanto riguarda la ricerca di un valore dato all'interno di un albero binario con n nodi, è possibile ottenere un risultato analogo nel caso medio adattando il concetto di ordinamento per strutture lineari alle strutture gerarchiche.
- Si dice albero binario di ricerca un albero binario tale che, per ogni nodo, se il nodo contiene una chiave di valore k, allora ogni nodo del suo sottoalbero sinistro contiene una chiave di valore $\leq k$, mentre ogni nodo del suo sottoalbero destro contiene una chiave di valore $\geq k$.
- In un albero binario di ricerca non è necessario visitare tutti i nodi sistematicamente per determinare se un valore dato è presente oppure no. Basta invece fare un unico percorso tra quelli che partono dalla radice, scendendo ad ogni nodo incontrato che non contiene il valore dato a sinistra o a destra a seconda che il valore dato sia minore o maggiore, rispettivamente, della chiave contenuta nel nodo.
- Algoritmo di ricerca di un valore in un albero binario di ricerca (restituisce l'indirizzo del nodo che contiene il valore se presente, `NULL` se il valore è assente):

```
nodo_albero_bin_t *cerca_in_albero_bin_ric(nodo_albero_bin_t *radice_p,
                                           int               valore)
{
  nodo_albero_bin_t *nodo_p;

  for (nodo_p = radice_p;
       ((nodo_p != NULL) && (nodo_p->valore != valore));
       nodo_p = (valore < nodo_p->valore)?
                  nodo_p->sx_p:
                  nodo_p->dx_p);
  return(nodo_p);
}
```

- L'algoritmo di inserimento di un valore in un albero binario di ricerca deve garantire che l'albero binario ottenuto a seguito dell'inserimento sia ancora di ricerca. Questo è il caso se l'inserimento avviene esclusivamente tramite l'aggiunta di una nuova foglia (come per le liste ordinate, assumiamo che i valori contenuti nei nodi siano tutti distinti):

```
int inserisci_in_albero_bin_ric(nodo_albero_bin_t **radice_p,
                                int               valore)
{
  int               inserito;
  nodo_albero_bin_t *nodo_p,
                    *padre_p,
                    *nuovo_p;

  for (nodo_p = padre_p = *radice_p;
       ((nodo_p != NULL) && (nodo_p->valore != valore));
       padre_p = nodo_p, nodo_p = (valore < nodo_p->valore)?
                                    nodo_p->sx_p:
                                    nodo_p->dx_p);
  if (nodo_p != NULL)
    inserito = 0;
```

```
  else
  {
    inserito = 1;
    nuovo_p = (nodo_albero_bin_t *)malloc(sizeof(nodo_albero_bin_t));
    nuovo_p->valore = valore;
    nuovo_p->sx_p = nuovo_p->dx_p = NULL;
    if (nodo_p == *radice_p)
      *radice_p = nuovo_p;
    else
      if (valore < padre_p->valore)
        padre_p->sx_p = nuovo_p;
      else
        padre_p->dx_p = nuovo_p;
  }
  return(inserito);
}
```

- Analogamente all'algoritmo di inserimento, l'algoritmo di rimozione di un valore da un albero binario di ricerca deve garantire che l'albero binario ottenuto a seguito della rimozione sia ancora di ricerca. Se il nodo contenente il valore da rimuovere è una foglia, basta eliminarlo. Se il nodo contenente il valore da rimuovere ha un solo figlio, basta eliminarlo collegando suo padre direttamente a suo figlio. Se infine il nodo contenente il valore da rimuovere ha ambedue i figli, si procede sostituendone il valore con quello del nodo più a destra del suo sottoalbero sinistro, in quanto tale nodo contiene la massima chiave minore di quella del nodo da rimuovere (in alternativa, si può prendere il nodo più a sinistra del sottoalbero destro):

```
int rimuovi_da_albero_bin_ric(nodo_albero_bin_t **radice_p,
                              int               valore)
{
  int               rimosso;
  nodo_albero_bin_t *nodo_p,
                    *padre_p,
                    *sost_p;

  for (nodo_p = padre_p = *radice_p;
       ((nodo_p != NULL) && (nodo_p->valore != valore));
       padre_p = nodo_p, nodo_p = (valore < nodo_p->valore)?
                                  nodo_p->sx_p:
                                  nodo_p->dx_p);
  if (nodo_p == NULL)
    rimosso = 0;
  else
  {
    rimosso = 1;
    if (nodo_p->sx_p == NULL)
    {
      if (nodo_p == *radice_p)
        *radice_p = nodo_p->dx_p;
      else
        if (valore < padre_p->valore)
          padre_p->sx_p = nodo_p->dx_p;
        else
          padre_p->dx_p = nodo_p->dx_p;
    }
```

```
    else
      if (nodo_p->dx_p == NULL)
      {
        if (nodo_p == *radice_p)
          *radice_p = nodo_p->sx_p;
        else
          if (valore < padre_p->valore)
            padre_p->sx_p = nodo_p->sx_p;
          else
            padre_p->dx_p = nodo_p->sx_p;
      }
      else
      {
        sost_p = nodo_p;
        for (padre_p = sost_p, nodo_p = sost_p->sx_p;
             (nodo_p->dx_p != NULL);
             padre_p = nodo_p, nodo_p = nodo_p->dx_p);
        sost_p->valore = nodo_p->valore;
        if (padre_p == sost_p)
          padre_p->sx_p = nodo_p->sx_p;
        else
          padre_p->dx_p = nodo_p->sx_p;
      }
    free(nodo_p);
  }
  return(rimosso);
}
```

■fff_10

- La complessità asintotica dei tre precedenti algoritmi dipende dal numero di iterazioni delle istruzioni `for`. Nel seguito indichiamo con n il numero di nodi dell'albero binario di ricerca e con h la sua altezza.
- Nel caso ottimo non hanno luogo iterazioni in quanto il valore cercato o da inserire si trova nella radice, oppure la rimozione deve essere effettuata in un albero binario di ricerca vuoto. In questo caso:
$$T(n) = O(1)$$
- Nel caso pessimo il numero di iterazioni è proporzionale ad h, in quanto bisogna compiere tutto il percorso dalla radice alla foglia più distante da essa prima di concludere l'operazione di ricerca, inserimento o rimozione. Poiché l'altezza di un albero è massima quando l'albero degenera in una lista, nel caso pessimo:
$$T(n) = O(n)$$
- Nel caso medio il numero di iterazioni è proporzionale alla lunghezza media del percorso per raggiungere un nodo dalla radice. Assumiamo che le n chiavi presenti nell'albero binario di ricerca siano state inserite in ordine casuale (cioè che tutte le $n!$ sequenze di inserimento siano equiprobabili) e che la distribuzione di probabilità di ricerca delle n chiavi sia uniforme. Denotiamo con l_j la lunghezza dell'unico percorso che congiunge il nodo contenente la j-esima chiave alla radice. La lunghezza media del percorso per raggiungere il nodo contenente una chiave a partire dalla radice in un albero binario di ricerca con n nodi è:
$$L_n = \frac{1}{n} \cdot \sum_{j=1}^{n} l_j$$
Se la radice contiene la i-esima chiave, allora il suo sottoalbero sinistro contiene le $i-1$ chiavi che la precedono e il suo sottoalbero di destra contiene le $n-i$ chiavi che la seguono. Pertanto, nel caso in cui la radice contiene la i-esima chiave, la lunghezza media del percorso per raggiungere a partire dalla radice il nodo contenente una chiave in un albero binario di ricerca con n nodi è:

$$\begin{aligned} L_{n,i} &= \tfrac{1}{n} \cdot (\sum_{j=1}^{i-1} l_j + 1 + \sum_{j=i+1}^{n} l_j) \\ &= \tfrac{1}{n} \cdot ((i-1) \cdot (L_{i-1}+1) + 1 + (n-i) \cdot (L_{n-i}+1)) \\ &= \tfrac{1}{n} \cdot ((i-1) \cdot L_{i-1} + (n-i) \cdot L_{n-i} + n) \end{aligned}$$

Al variare della chiave contenuta nella radice si ottiene pertanto:

$$\begin{aligned} L_n &= \tfrac{1}{n} \cdot \sum_{i=1}^{n} L_{n,i} \\ &= \tfrac{1}{n^2} \cdot \sum_{i=1}^{n} ((i-1) \cdot L_{i-1} + (n-i) \cdot L_{n-i} + n) \\ &= \tfrac{1}{n^2} \cdot \sum_{i=1}^{n} ((i-1) \cdot L_{i-1}) + \tfrac{1}{n^2} \cdot \sum_{i=1}^{n} ((n-i) \cdot L_{n-i}) + 1 \end{aligned}$$

Ponendo $k = n - i + 1$ nella seconda sommatoria si ha:

$$L_n = \tfrac{1}{n^2} \cdot \sum_{i=1}^{n} ((i-1) \cdot L_{i-1}) + \tfrac{1}{n^2} \cdot \sum_{k=1}^{n} ((k-1) \cdot L_{k-1}) + 1 = \tfrac{2}{n^2} \cdot \sum_{i=1}^{n} ((i-1) \cdot L_{i-1}) + 1$$

Ponendo $k = i - 1$ nell'unica sommatoria rimasta si ha:

$$L_n = \tfrac{2}{n^2} \cdot \sum_{k=0}^{n-1} (k \cdot L_k) + 1 = \tfrac{2}{n^2} \cdot (n-1) \cdot L_{n-1} + \tfrac{2}{n^2} \cdot \sum_{k=1}^{n-2} (k \cdot L_k) + 1$$

Poiché nel caso di un albero binario di ricerca con $n-1$ nodi sarebbe:

$$L_{n-1} = \tfrac{2}{(n-1)^2} \cdot \sum_{k=1}^{n-2} (k \cdot L_k) + 1$$

da cui (anche dividendo ambo i membri per n^2):

$$\tfrac{2}{n^2} \cdot \sum_{k=1}^{n-2} (k \cdot L_k) = \tfrac{(n-1)^2}{n^2} \cdot (L_{n-1} - 1)$$

andando a sostituire l'unica sommatoria rimasta si ricava:

$$L_n = \tfrac{2}{n^2} \cdot (n-1) \cdot L_{n-1} + \tfrac{(n-1)^2}{n^2} \cdot (L_{n-1} - 1) + 1 = \tfrac{n^2-1}{n^2} \cdot L_{n-1} + \tfrac{2 \cdot n - 1}{n^2}$$

Assieme all'equazione $L_1 = 1$, ciò dà luogo ad una relazione di ricorrenza lineare, di ordine 1, a coefficienti non costanti (in quanto dipendenti da n) e non omogenea, la cui soluzione in forma chiusa è:

$$L_n = 2 \cdot \tfrac{n+1}{n} \cdot \sum_{i=1}^{n} \tfrac{1}{i} - 3$$

Poiché $(n+1)/n$ tende asintoticamente ad 1 e inoltre:

$$\ln(n+1) = \int_1^{n+1} \tfrac{1}{x}\,dx \leq \sum_{i=1}^{n} \tfrac{1}{i} \leq \int_1^{n} \tfrac{1}{x}\,dx + 1 = \ln n + 1$$

si ha:

$$L_n = O(\log n)$$

In conclusione, nel caso medio la complessità degli algoritmi di ricerca, inserimento e rimozione per un albero binario di ricerca con n nodi è:

$$T(n) = O(\log n)$$

6.4 Criteri di bilanciamento per alberi binari di ricerca

- Se si vuole ottenere anche nel caso pessimo (e non solo in quello medio) una complessità del tipo $O(\log n)$ anziché $O(n)$ per l'operazione di ricerca di un valore dato all'interno di un albero binario di ricerca con n nodi, bisogna mantenere l'albero bilanciato. Infatti, solo in questo caso si può garantire che l'altezza h dell'albero cresca logaritmicamente al crescere del numero n di nodi.

- Dal punto di vista dell'inserimento, per mantenere il bilanciamento bisogna mettere i nuovi nodi nei livelli già esistenti così da riempirli il più possibile prima di creare nuovi livelli.

- Dal punto di vista della rimozione, per mantenere il bilanciamento bisogna ridistribuire i nodi rimasti all'interno dei livelli nel modo più uniforme possibile.

- Inserimenti e rimozioni debbono inoltre garantire che l'albero binario risultante sia ancora un albero binario di ricerca.

- Si dice che un albero binario di ricerca è perfettamente bilanciato se, per ogni nodo, il numero di nodi del suo sottoalbero sinistro e il numero di nodi del suo sottoalbero destro differiscono al più di 1.

- Poiché il perfetto bilanciamento è troppo costoso da mantenere dopo un inserimento o una rimozione, sono state introdotte delle forme di bilanciamento meno restrittive che approssimano quello perfetto.

- Si dice che un albero binario di ricerca è AVL-bilanciato (Adelson-Velsky & Landis – 1962) se, per ogni nodo, l'altezza del suo sottoalbero sinistro e l'altezza del suo sottoalbero destro differiscono al più di 1.

- L'altezza h di un albero binario di ricerca AVL-bilanciato con n nodi soddisfa:
$$\log(n+1) \leq h \leq 1.4404 \cdot \log(n+2) - 0.328$$
tuttavia l'AVL-bilanciamento è ancora piuttosto costoso da mantenere dopo una rimozione (il numero di operazioni da effettuare in quel caso per ribilanciare l'albero potrebbe non essere costante).

- Si dice che un albero binario di ricerca è rosso-nero (Bayer – 1972) se, dopo aver aggiunto un nodo sentinella come figlio di tutte le foglie e dopo aver colorato ogni nodo di rosso o di nero, si ha che:

  (1) La radice è nera.

  (2) La sentinella è nera.

  (3) Se un nodo è rosso, allora tutti i suoi figli sono neri.

  (4) Per ogni nodo, tutti i percorsi dal nodo alla sentinella attraversano lo stesso numero di nodi neri.

- L'altezza h di un albero binario di ricerca rosso-nero con n nodi soddisfa:
$$h \leq 2 \cdot \log_2(n+1)$$
Infatti, indichiamo con $bn(x)$ il numero di nodi neri che si incontrano lungo un qualsiasi percorso che va da un nodo x alla sentinella, esclusi x (se nero) e la sentinella in quanto essi fanno parte di tutti i percorsi da considerare. Il valore di $bn(x)$ è univocamente definito in virtù di (4). Proviamo che il numero $n(x)$ di nodi del sottoalbero generato da x soddisfa $n(x) \geq 2^{bn(x)} - 1$ procedendo per induzione sull'altezza $h(x)$ di tale sottoalbero:

  – Se $h(x) = 1$, allora $n(x) = 1$ in quanto il sottoalbero contiene il solo nodo sentinella x. Pertanto $bn(x) = 0$ da cui segue che $2^{bn(x)} - 1 = 0 \leq 1 = n(x)$, quindi l'asserto è vero per $h(x) = 1$.

  – Se $h(x) > 1$, allora x è diverso dalla sentinella e quindi ha ambedue i figli (eventualmente coincidenti con la sentinella), che denotiamo con y e z. Pertanto $n(x) = n(y)+1+n(z)$. Concentriamoci ora su y. Poiché $h(y) \leq h(x)-1$, per ipotesi induttiva $n(y) \geq 2^{bn(y)} - 1$. Se y è rosso o coincide con la sentinella, $bn(y) = bn(x)$. Se invece y è nero e diverso dalla sentinella, $bn(y) = bn(x)-1$. Pertanto $bn(y) \geq bn(x) - 1$ e quindi $n(y) \geq 2^{bn(x)-1} - 1$. Analogamente si ricava che $n(z) \geq 2^{bn(x)-1} - 1$. Di conseguenza $n(x) \geq (2^{bn(x)-1} - 1) + 1 + (2^{bn(x)-1} - 1) = 2 \cdot (2^{bn(x)-1} - 1) + 1 = 2^{bn(x)} - 1$.

  In virtù di (3), almeno la metà dei nodi su un qualsiasi percorso dalla radice r alla sentinella sono neri, quindi $bn(r) \geq h/2$. Poiché da quanto provato prima segue che $n \geq 2^{bn(r)} - 1$, si ha $n \geq 2^{h/2} - 1$ da cui $n + 1 \geq 2^{h/2}$ e quindi $h \leq 2 \cdot \log_2(n+1)$.

- Un albero binario di ricerca rosso-nero è quindi una buona approssimazione di un albero binario di ricerca perfettamente bilanciato (sebbene ciò non sia evidente dalla sua definizione) e vedremo che il mantenimento del bilanciamento dopo un inserimento o una rimozione è meno oneroso rispetto al caso di un albero binario di ricerca AVL-bilanciato (in termini di numero di operazioni di ribilanciamento).

■fff_11

6.5 Algoritmi di ricerca, inserimento e rimozione per alberi binari di ricerca rosso-nero

- Nella rappresentazione di un albero binario di ricerca rosso-nero, ogni elemento contiene in aggiunta l'informazione sul colore del nodo e l'indirizzo del padre del nodo:

```
typedef enum {rosso, nero} colore_t;

typedef struct nodo_albero_bin_rn
{
  int                       valore;
  colore_t                  colore;
  struct nodo_albero_bin_rn *sx_p, *dx_p, *padre_p;
} nodo_albero_bin_rn_t;
```

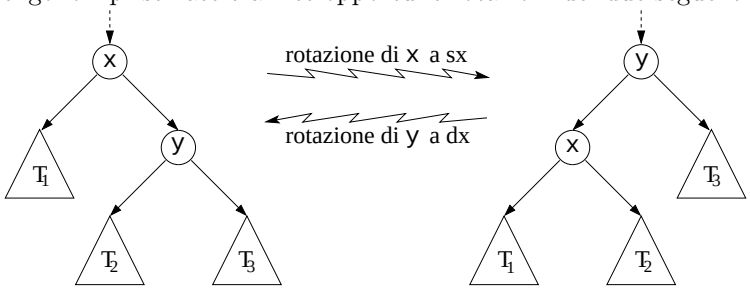
- Un albero binario di ricerca rosso-nero è individuato attraverso l'indirizzo della sentinella. Poiché la sentinella ha più padri (tutte le foglie), l'indirizzo del padre della sentinella è `NULL`. Gli indirizzi dei figli della sentinella coincidono con l'indirizzo della radice se l'albero non è vuoto, con l'indirizzo della sentinella stessa altrimenti. Pertanto l'indirizzo del padre della radice è l'indirizzo della sentinella.

- L'algoritmo di ricerca di un valore in un albero binario di ricerca rosso-nero è una lieve variazione dello stesso algoritmo per un albero binario di ricerca:

```
nodo_albero_bin_rn_t *cerca_in_albero_bin_ric_rn(nodo_albero_bin_rn_t *sent_p,
                                                 int                  valore)
{
  nodo_albero_bin_rn_t *nodo_p;

  for (nodo_p = sent_p->sx_p;
       ((nodo_p != sent_p) && (nodo_p->valore != valore));
       nodo_p = (valore < nodo_p->valore)?
                  nodo_p->sx_p:
                  nodo_p->dx_p);
  return(nodo_p);
}
```

  Se l'albero binario di ricerca rosso-nero ha n nodi, la complessità nel caso pessimo è:
  $$T(n) = O(\log n)$$

- A seguito di inserimenti e rimozioni, le proprietà (1), (3) e (4) potrebbero essere violate. Quando ciò accade, le proprietà vengono ripristinate tramite opportune rotazioni dei due seguenti tipi:



  Se l'albero binario di partenza è di ricerca, lo sono anche gli alberi ottenuti a seguito delle rotazioni.

- Tali rotazioni vengono applicate risalendo lungo il percorso che aveva portato all'individuazione del punto in cui effettuare l'inserimento o la rimozione. Per questo motivo nella rappresentazione di un albero binario di ricerca rosso-nero ogni elemento contiene anche l'indirizzo del padre.

- La rotazione a sinistra può essere applicata a qualsiasi nodo avente il figlio destro:

```
void ruota_sx(nodo_albero_bin_rn_t *sent_p,
              nodo_albero_bin_rn_t *x_p)
{
  nodo_albero_bin_rn_t *y_p;

  y_p = x_p->dx_p;
  x_p->dx_p = y_p->sx_p;
  y_p->sx_p->padre_p = x_p;
  y_p->padre_p = x_p->padre_p;
  if (x_p == sent_p->sx_p)
    sent_p->sx_p = sent_p->dx_p = y_p;
  else
    if (x_p == x_p->padre_p->sx_p)
      x_p->padre_p->sx_p = y_p;
    else
      x_p->padre_p->dx_p = y_p;
  y_p->sx_p = x_p;
  x_p->padre_p = y_p;
}
```

  La rotazione a destra è perfettamente simmetrica. La complessità di ciascuna delle due rotazioni effettuata in un albero binario di ricerca rosso-nero con n nodi è:

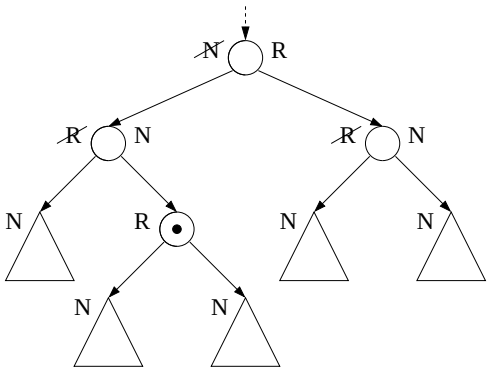
$$T(n) = O(1)$$

- L'algoritmo di inserimento di un valore in un albero binario di ricerca rosso-nero è simile allo stesso algoritmo per un albero binario di ricerca, con in più l'inclusione di un algoritmo che ripristina le proprietà eventualmente violate (come al solito, assumiamo che i valori contenuti nei nodi siano tutti distinti):

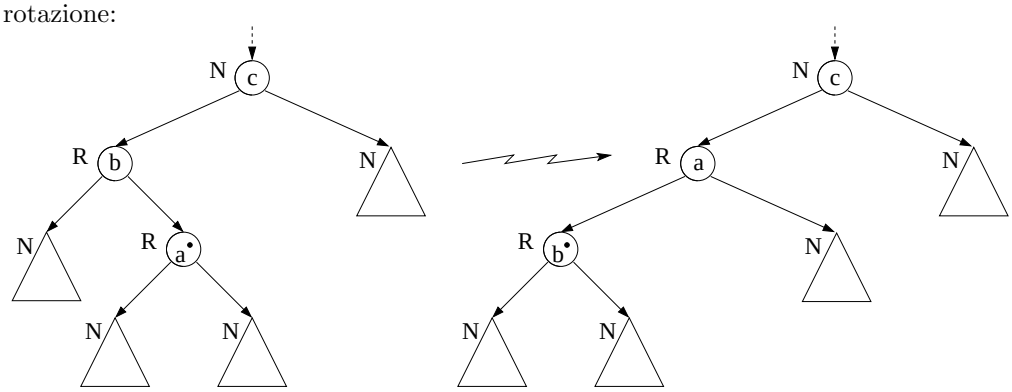
```
int inserisci_in_albero_bin_ric_rn(nodo_albero_bin_rn_t *sent_p,
                                   int                  valore)
{
  int                  inserito;
  nodo_albero_bin_rn_t *nodo_p,
                       *padre_p,
                       *nuovo_p;

  for (nodo_p = sent_p->sx_p, padre_p = sent_p;
       ((nodo_p != sent_p) && (nodo_p->valore != valore));
       padre_p = nodo_p, nodo_p = (valore < nodo_p->valore)?
                                   nodo_p->sx_p:
                                   nodo_p->dx_p);
  if (nodo_p != sent_p)
    inserito = 0;
  else
  {
    inserito = 1;
    nuovo_p = (nodo_albero_bin_rn_t *)malloc(sizeof(nodo_albero_bin_rn_t));
    nuovo_p->valore = valore;
    nuovo_p->colore = rosso;
    nuovo_p->sx_p = nuovo_p->dx_p = sent_p;
    nuovo_p->padre_p = padre_p;
```

```
    if (padre_p == sent_p)
      sent_p->sx_p = sent_p->dx_p = nuovo_p;
    else
      if (valore < padre_p->valore)
        padre_p->sx_p = nuovo_p;
      else
        padre_p->dx_p = nuovo_p;
    ripristina_ins_albero_bin_ric_rn(sent_p,
                                     nuovo_p);
  }
  return(inserito);
}
```

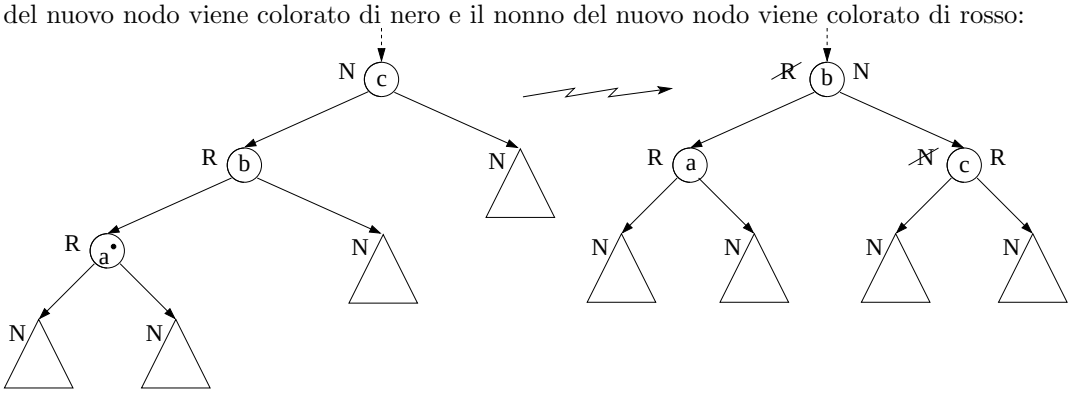
- Poiché il nuovo nodo viene colorato di rosso, il suo inserimento non può violare la proprietà (4), ma solo la (1) o la (3). La proprietà (1) viene violata quando l'albero binario di ricerca rosso-nero è vuoto e quindi il nuovo nodo rosso è la radice. La proprietà viene ripristinata semplicemente cambiando di colore il nuovo nodo inserito.
- La proprietà (3) viene violata quando il padre del nuovo nodo rosso è anch'esso rosso. Per quanto riguarda il ripristino di questa proprietà, ci sono due casi nell'albero ottenuto a seguito dell'inserimento:
  - Se lo zio del nuovo nodo è esso pure rosso – da cui segue che il nonno del nuovo nodo è nero altrimenti la proprietà (3) non poteva essere valida nell'albero di partenza – il padre e lo zio del nuovo nodo vengono colorati di nero, il nonno del nuovo nodo viene colorato di rosso, e poi si applica l'algoritmo di ripristino al nonno in quanto esso potrebbe essere la radice oppure suo padre potrebbe essere rosso. Il nonno viene colorato di rosso perché, se rimanesse nero e non fosse la radice, la proprietà (4) verrebbe violata dal bisnonno:



  - Se lo zio del nodo è nero, ci sono due sottocasi:
    - Se il nuovo nodo è figlio destro, il padre del nuovo nodo viene ruotato a sinistra, riconducendosi al secondo sottocaso a patto di considerare il padre del nuovo nodo come il nuovo nodo dopo la rotazione:



∗ Se il nuovo nodo è figlio sinistro, il nonno del nuovo nodo viene ruotato a destra e poi il padre del nuovo nodo viene colorato di nero e il nonno del nuovo nodo viene colorato di rosso:



■fff_12

- Questo è l'algoritmo di ripristino in caso di inserimento:

```
void ripristina_ins_albero_bin_ric_rn(nodo_albero_bin_rn_t *sent_p,
                                      nodo_albero_bin_rn_t *nodo_p)
{
  nodo_albero_bin_rn_t *zio_p;

  while (nodo_p->padre_p->colore == rosso)
    if (nodo_p->padre_p == nodo_p->padre_p->padre_p->sx_p)
    {
      zio_p = nodo_p->padre_p->padre_p->dx_p;
      if (zio_p->colore == rosso)
      {
        nodo_p->padre_p->colore = nero;
        zio_p->colore = nero;
        nodo_p->padre_p->padre_p->colore = rosso;
        nodo_p = nodo_p->padre_p->padre_p;
      }
      else
      {
        if (nodo_p == nodo_p->padre_p->dx_p)
        {
          nodo_p = nodo_p->padre_p;
          ruota_sx(sent_p,
                   nodo_p);
        }
        nodo_p->padre_p->colore = nero;
        nodo_p->padre_p->padre_p->colore = rosso;
        ruota_dx(sent_p,
                 nodo_p->padre_p->padre_p);
      }
    }
    else
      /* stesse istruzioni in cui sx e dx sono scambiati */
      ...
  sent_p->sx_p->colore = nero;
}
```

- In conclusione, nel caso pessimo l'algoritmo di inserimento in un albero binario di ricerca rosso-nero con n nodi percorre l'intera altezza due volte (prima scendendo e poi risalendo) ed esegue 2 rotazioni, quindi:
$$T(n) = O(\log n)$$
- L'algoritmo di rimozione di un valore da un albero binario di ricerca rosso-nero è simile allo stesso algoritmo per un albero binario di ricerca, con in più l'inclusione di un algoritmo che ripristina le proprietà eventualmente violate:

```
int rimuovi_da_albero_bin_ric_rn(nodo_albero_bin_rn_t *sent_p,
                                 int                  valore)
{
  int                  rimosso;
  nodo_albero_bin_rn_t *nodo_p,
                       *padre_p,
                       *figlio_p,
                       *sost_p;

  for (nodo_p = sent_p->sx_p, padre_p = sent_p;
       ((nodo_p != sent_p) && (nodo_p->valore != valore));
       padre_p = nodo_p, nodo_p = (valore < nodo_p->valore)?
                                    nodo_p->sx_p:
                                    nodo_p->dx_p);
  if (nodo_p == sent_p)
    rimosso = 0;
  else
  {
    rimosso = 1;
    if ((nodo_p->sx_p == sent_p) || (nodo_p->dx_p == sent_p))
    {
      figlio_p = (nodo_p->sx_p == sent_p)?
                  nodo_p->dx_p:
                  nodo_p->sx_p;
      figlio_p->padre_p = padre_p;
      if (padre_p == sent_p)
        sent_p->sx_p = sent_p->dx_p = figlio_p;
      else
        if (valore < padre_p->valore)
          padre_p->sx_p = figlio_p;
        else
          padre_p->dx_p = figlio_p;
    }
    else
    {
      sost_p = nodo_p;
      for (padre_p = sost_p, nodo_p = sost_p->sx_p;
           (nodo_p->dx_p != sent_p);
           padre_p = nodo_p, nodo_p = nodo_p->dx_p);
      sost_p->valore = nodo_p->valore;
      figlio_p = nodo_p->sx_p;
      figlio_p->padre_p = padre_p;
      if (padre_p == sost_p)
        padre_p->sx_p = figlio_p;
      else
        padre_p->dx_p = figlio_p;
    }
```

```
    if (nodo_p->colore == nero)
      ripristina_rim_albero_bin_ric_rn(sent_p,
                                       figlio_p);
    free(nodo_p);
  }
  return(rimosso);
}
```

- La rimozione di un nodo (non dotato di entrambi i figli) viola la proprietà (1), (3) o (4) solo se tale nodo è nero.
- La proprietà (1) viene violata quando il nodo rimosso è la radice e l'unico suo figlio, che diventa quindi la nuova radice, è rosso. La proprietà viene ripristinata semplicemente cambiando di colore l'unico figlio del nodo rimosso.
- La proprietà (3) viene violata quando l'unico figlio e il padre del nodo rimosso sono ambedue rossi. Anche in questo caso la proprietà viene ripristinata semplicemente cambiando di colore l'unico figlio del nodo rimosso.
- In caso di violazione della proprietà (4), se l'unico figlio del nodo rimosso è rosso, allora è sufficiente cambiarlo di colore, altrimenti ci sono quattro sottocasi nell'albero ottenuto a seguito della rimozione:
  - Se il fratello del figlio è rosso, il colore del fratello del figlio viene scambiato con quello del padre del figlio e poi il padre del figlio viene ruotato a sinistra, riconducendosi ad uno degli altri tre sottocasi.
  - Se il fratello del figlio è nero e i suoi due figli sono ambedue neri, il fratello del figlio viene colorato di rosso e si continua considerando come nuovo figlio il padre dell'attuale figlio.
  - Se il fratello del figlio è nero, suo figlio sinistro è rosso e suo figlio destro è nero, il colore del fratello del figlio viene scambiato con quello di suo figlio sinistro e poi il fratello del figlio viene ruotato a destra, riconducendosi all'ultimo sottocaso rimasto.
  - Se il fratello del figlio è nero e suo figlio destro è rosso, il fratello del figlio viene colorato con il colore del padre, il padre e il figlio destro vengono colorati di nero, e poi il padre del figlio viene ruotato a sinistra.
- Questo è l'algoritmo di ripristino in caso di rimozione:

```
void ripristina_rim_albero_bin_ric_rn(nodo_albero_bin_rn_t *sent_p,
                                      nodo_albero_bin_rn_t *nodo_p)
{
  nodo_albero_bin_rn_t *fratello_p;

  while ((nodo_p != sent_p->sx_p) && (nodo_p->colore == nero))
    if (nodo_p == nodo_p->padre_p->sx_p)
    {
      fratello_p = nodo_p->padre_p->dx_p;
      if (fratello_p->colore == rosso)
      {
        fratello_p->colore = nero;
        nodo_p->padre_p->colore = rosso;
        ruota_sx(sent_p,
                 nodo_p->padre_p);
        fratello_p = nodo_p->padre_p->dx_p;
      }
```

```
      if ((fratello_p->sx_p->colore == nero) && (fratello_p->dx_p->colore == nero))
      {
        fratello_p->colore = rosso;
        nodo_p = nodo_p->padre_p;
      }
      else
      {
        if (fratello_p->dx_p->colore == nero)
        {
          fratello_p->sx_p->colore = nero;
          fratello_p->colore = rosso;
          ruota_dx(sent_p,
                   fratello_p);
          fratello_p = nodo_p->padre_p->dx_p;
        }
        fratello_p->colore = nodo_p->padre_p->colore;
        nodo_p->padre_p->colore = nero;
        fratello_p->dx_p->colore = nero;
        ruota_sx(sent_p,
                 nodo_p->padre_p);
        nodo_p = sent_p->sx_p;
      }
    }
    else
      /* stesse istruzioni in cui sx e dx sono scambiati */
      ...
  nodo_p->colore = nero;
}
```

- In conclusione, nel caso pessimo l'algoritmo di rimozione da un albero binario di ricerca rosso-nero con n nodi percorre l'intera altezza due volte (prima scendendo e poi risalendo) ed esegue 3 rotazioni, quindi:

$$T(n) = O(\log n)$$

■fff_13

Capitolo 7

Algoritmi per grafi

7.1 Grafi: definizioni di base e problemi classici

- Si dice grafo diretto o orientato una coppia $G = (V, \mathcal{E})$ dove V è un insieme di vertice ed $\mathcal{E}$ è una relazione binaria su V.
- L'albero è un caso particolare di grafo diretto in cui esiste un unico vertice dal quale ciascuno degli altri vertici è raggiungibile attraverso un unico percorso.
- Sia $G = (V, \mathcal{E})$ un grafo diretto. Si dice che $G' = (V', \mathcal{E}')$ è:
  - un sottografo di G se $V' \subseteq V$ ed $\mathcal{E}' \subseteq \mathcal{E} \cap (V' \times V')$;
  - un sottografo indotto di G se $V' \subseteq V$ ed $\mathcal{E}' = \mathcal{E} \cap (V' \times V')$;
  - il grafo trasposto di G se $V' = V$ ed $\mathcal{E}' = \{(v', v) \mid (v, v') \in \mathcal{E}\}$.
- Sia $G = (V, \mathcal{E})$ un grafo diretto:
  - Se $(v, v') \in \mathcal{E}$, allora si dice che v' è adiacente a v o, equivalentemente, che c'è un arco da v a v'.
  - Si dice grado uscente di $v \in V$ il numero di vertici adiacenti a v:
    $$d_{\mathrm{o}}(v) = |\{v' \in V \mid (v, v') \in \mathcal{E}\}|$$
  - Si dice grado entrante di $v \in V$ il numero di vertici ai quali v è adiacente:
    $$d_{\mathrm{i}}(v) = |\{v' \in V \mid (v', v) \in \mathcal{E}\}|$$
  - Si dice grado di $v \in V$ il numero di archi in cui v è coinvolto:
    $$d(v) = d_{\mathrm{o}}(v) + d_{\mathrm{i}}(v)$$
  - Se $v \in V$ è tale che:
    - $d_{\mathrm{o}}(v) = 0$ e $d_{\mathrm{i}}(v) > 0$ allora si dice che v è un vertice terminale;
    - $d_{\mathrm{i}}(v) = 0$ e $d_{\mathrm{o}}(v) > 0$ allora si dice che v è un vertice iniziale;
    - $d(v) = 0$ allora si dice che v è un vertice isolato.
  - Si dice grado di G il massimo grado di un vertice di G:
    $$d(G) = \max\{d(v) \mid v \in V\}$$
  - Si dice che G è completo se $\mathcal{E} = V \times V$.

- Sia $G = (V, \mathcal{E})$ un grafo diretto:
  - Siano $v_1, v_2 \in V$. Si dice che v_2 è raggiungibile da v_1 se esiste un percorso da v_1 a v_2, cioè se esistono $v'_1, \ldots, v'_k \in V$ con $k \geq 2$ tali che $v'_1 = v_1$, $(v'_i, v'_{i+1}) \in \mathcal{E}$ per ogni $1 \leq i \leq k-1$, e $v'_k = v_2$.
  - Si dice che un percorso è semplice se tutti i vertici che lo compongono sono distinti, eccetto al più il primo e l'ultimo vertice.
  - Si dice che un percorso è un ciclo se il suo primo vertice coincide con il suo ultimo vertice.
  - Si dice che G è connesso se per ogni $v_1, v_2 \in V$ esiste un percorso da v_1 a v_2 o da v_2 a v_1.
  - Si dice che G è fortemente connesso se per ogni $v_1, v_2 \in V$ esistono un percorso da v_1 a v_2 e un percorso da v_2 a v_1.
  - Si dice che $G' = (V', \mathcal{E}')$ è una componente connessa (risp. fortemente connessa) di G se G' è un sottografo indotto di G che è connesso (risp. fortemente connesso) e massimale.
  - Si dice che G è ciclico se contiene almeno un ciclo, aciclico se non contiene nessun ciclo.
- Si dice grafo indiretto o non orientato una coppia $G = (V, \mathcal{E})$ dove V è un insieme di vertici ed $\mathcal{E}$ è una relazione binaria su V antiriflessiva e simmetrica.
- L'albero libero è un caso particolare di grafo indiretto in cui la relazione binaria gode anche delle proprietà di connettività e aciclicità.
- Tutte le definizioni date per i grafi diretti valgono per i grafi indiretti tranne le definizioni di grado uscente e grado entrante e la definizione di ciclo, la quale richiede in aggiunta che il numero di vertici distinti facenti parte del percorso sia almeno 3 (altrimenti ogni coppia di vertici adiacenti genererebbe un ciclo per la simmetria).
- Si dice grafo pesato una tripla $G = (V, \mathcal{E}, w)$ dove $G = (V, \mathcal{E})$ è un grafo e $w : \mathcal{E} \longrightarrow \mathbb{R}$ è una funzione detta peso. Il peso associato ad un arco rappresenta di solito un tempo, una distanza, una capacità o un guadagno/perdita.
- Un grafo viene solitamente rappresentato come una struttura dati dinamica reticolare detta lista di adiacenza, formata da una lista primaria dei vertici e più liste secondarie degli archi. La lista primaria contiene un elemento per ciascun vertice del grafo, il quale contiene a sua volta la testa della relativa lista secondaria. La lista secondaria associata ad un vertice descrive tutti gli archi uscenti da quel vertice, in quanto contiene gli indirizzi di tutti i vertici adiacenti al vertice in questione:

```
typedef struct vertice_grafo
{
  int                 valore;
  struct vertice_grafo *vertice_succ_p;
  struct arco_grafo    *lista_archi_p;
} vertice_grafo_t;

typedef struct arco_grafo
{
  double              peso;              /* presente in caso di grafo pesato */
  struct vertice_grafo *vertice_adiac_p;
  struct arco_grafo    *arco_succ_p;
} arco_grafo_t;
```

- In questo caso i vertici e gli archi di un grafo non sono necessariamente memorizzati in modo consecutivo. L'accesso ad ognuno di essi necessita dell'indirizzo del primo elemento della lista primaria, il quale è indefinito se e solo se il grafo è vuoto. L'accesso ad un vertice avviene scorrendo tutti i vertici che precedono il vertice in questione nella lista primaria. L'accesso ad un arco avviene individuando prima l'elemento della lista primaria associato al vertice di partenza e poi l'elemento della corrispondente lista secondaria associato al vertice di arrivo.

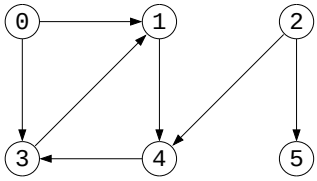
- Dato un grafo $G = (V, \mathcal{E})$, la sua rappresentazione mediante lista di adiacenza occupa uno spazio di memoria che è $O(|V| + |\mathcal{E}|)$. Ciò è particolarmente conveniente nel caso di grafi sparsi, cioè grafi per i quali $|\mathcal{E}| \ll |V|^2$. Per contro, il tempo per verificare l'esistenza di un arco tra due vertici dati è $O(|V| + |\mathcal{E}|)$ nel caso pessimo (il vertice di partenza è l'ultimo della lista primaria, tutti gli archi del grafo escono da questo vertice e l'arco in questione non esiste).

- Se la struttura di un grafo non cambia oppure è importante fare accesso rapidamente alle informazioni contenute nel grafo, allora conviene ricorrere ad una rappresentazione a matrice di adiacenza. Questa matrice ha tante righe e tante colonne quanti sono i vertici. L'elemento di indici i e j vale 1 (risp. d) se esiste un arco dal vertice i al vertice j (di peso d), 0 altrimenti:

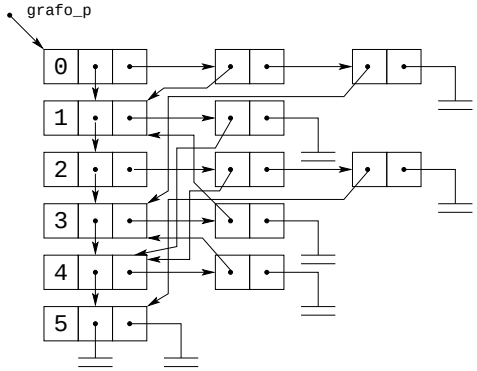
```
double grafo[NUMERO_VERTICI][NUMERO_VERTICI];
```

- Dato un grafo $G = (V, \mathcal{E})$, la sua rappresentazione mediante matrice di adiacenza consente la verifica dell'esistenza di un arco tra due vertici dati in un tempo che è $O(1)$. Per contro, l'occupazione di memoria è $O(|V|^2)$, che non è conveniente nel caso di grafi sparsi.

- Esempio: il seguente grafo diretto



ha la seguente rappresentazione a lista di adiacenza



e la seguente rappresentazione a matrice di adiacenza

0	1	0	1	0	0
0	0	0	0	1	0
0	0	0	0	1	1
0	1	0	0	0	0
0	0	0	1	0	0
0	0	0	0	0	0

- La complessità asintotica degli algoritmi sui grafi, la quale dipende dalla modalità di rappresentazione scelta, è espressa in funzione del numero di vertici e del numero di archi, considerati in modo aggregato. Supponiamo ad esempio che in un algoritmo che opera su un grafo $G = (V, \mathcal{E})$ sia presente un'iterazione sui vertici, all'interno della quale si annida un'iterazione sugli archi uscenti dal vertice in esame. Nel caso pessimo la complessità asintotica delle due iterazioni annidate è limitata superiormente sia da $|V| \cdot |\mathcal{E}|$ che da $|V| + |\mathcal{E}|$. Tra questi due valori, il secondo costituisce una maggiorazione più stretta, cioè più accurata rispetto alla complessità effettiva. In casi come questo conviene quindi usare $O(|V| + |\mathcal{E}|)$ piuttosto che $O(|V| \cdot |\mathcal{E}|)$.

- Tra i problemi classici per i grafi rientrano alcuni dei problemi che abbiamo già visto per le liste e per gli alberi, come il problema della visita e il problema della ricerca. Oltre a questi, considereremo anche alcuni problemi specifici dei grafi.

- Problema dell'ordinamento topologico: dato un grafo $G = (V, \mathcal{E})$ diretto e aciclico, determinare un ordinamento lineare dei suoi vertici tale che, se $(v, v') \in \mathcal{E}$, allora v precede v' nell'ordinamento. Un grafo diretto e aciclico può essere usato per stabilire che certi eventi associati ai propri vertici sono causalmente dipendenti tra loro mentre altri sono indipendenti tra loro, a seconda che i relativi vertici siano connessi da un percorso o meno. Un ordinamento topologico del grafo mette quindi tutti gli eventi in sequenza rispettando i vincoli di dipendenza causale.

- Problema delle componenti fortemente connesse: dato un grafo, decomporlo nelle sue componenti fortemente connesse. Molti algoritmi che lavorano su un grafo iniziano calcolando tale decomposizione, poi procedono operando sulle singole componenti fortemente connesse considerate separatamente, e infine assemblano i risultati parziali ottenuti dalle varie componenti fortemente connesse.

- Problema dell'albero ricoprente minimo: dato un grafo indiretto, connesso e pesato, determinare l'albero nell'insieme degli alberi liberi comprendenti tutti i vertici del grafo tale che la somma dei pesi dei suoi rami è minima. L'albero ricoprente minimo di un grafo indiretto, connesso e pesato indica come collegare i vertici del grafo nel modo più economico possibile sulla base dei pesi attribuiti agli archi, garantendo al tempo stesso una totale copertura dei vertici.

- Problema del percorso più breve: dati un grafo diretto e pesato e due suoi vertici distinti, determinare il percorso nell'insieme dei percorsi (se esistono) che congiungono il primo vertice al secondo vertice tale che la somma dei pesi dei suoi archi è minima. ■fff_14

7.2 Algoritmi di visita e ricerca per grafi

- Data la generalità della loro struttura, nel caso dei grafi ci sono ancora più alternative per effettuare una visita di quelle viste nel caso degli alberi binari. Di solito i vertici di un grafo $G = (V, \mathcal{E})$ vengono attraversati in maniera sistematica procedendo in ampiezza o in profondità.

- Ambedue le visite determinano un albero che ricopre il sottografo indotto comprendente tutti e soli i vertici raggiungibili da quello di partenza. Poiché non è necessariamente detto che questo sottografo indotto coincida con l'intero grafo, si può verificare una situazione in cui ci sono ancora dei vertici da visitare. Intuitivamente, l'attraversamento deve allora essere ripetuto tante volte quante sono le componenti connesse del grafo, partendo ogni volta da un vertice di una componente connessa diversa. Ciò non è necessario nel caso di grafi che sono liste o alberi, perché in tal caso c'è un'unica componente connessa e si parte dal suo unico vertice iniziale.

- Ambedue le visite hanno inoltre bisogno di attribuire dei colori ai vertici per ricordare se questi sono già stati incontrati o meno, in modo da garantire la terminazione nel caso di grafi ciclici. Inizialmente ogni vertice è bianco, poi diventa grigio la prima volta che viene incontrato, infine diventa nero dopo che tutti i vertici ad esso adiacenti sono stati incontrati. Nemmeno questo è necessario nel caso di grafi che sono liste o alberi, perché in tal caso non possono esserci cicli.

- Nella visita in ampiezza, i vertici vengono attraversati in ordine di distanza crescente dal vertice di partenza. Nella rappresentazione a lista di adiacenza estendiamo il tipo `vertice_grafo_t` in modo da ricordare per ogni vertice il colore, la distanza dal vertice di partenza e l'indirizzo del padre nell'albero ricoprente della componente connessa a cui appartiene il vertice di partenza:

```
typedef enum {bianco, grigio, nero} colore_t;

typedef struct vertice_grafo
{
  int                  valore;
```

```
  struct vertice_grafo *vertice_succ_p;
  struct arco_grafo    *lista_archi_p;
  colore_t             colore;
  int                  distanza;
  struct vertice_grafo *padre_p;
} vertice_grafo_t;
```

- L'algoritmo di visita in ampiezza è iterativo e fa uso esplicito di una coda di tipo `elem_lista_vertici_t`, il quale è una variante del tipo `elem_lista_t` in cui il campo `valore` è di tipo `vertice_grafo_t *`:

```
void avvia_visita_grafo_amp(vertice_grafo_t *grafo_p)
{
  vertice_grafo_t *vertice_p;

  for (vertice_p = grafo_p;
       (vertice_p != NULL);
       vertice_p = vertice_p->vertice_succ_p)
  {
    vertice_p->colore = bianco;
    vertice_p->distanza = -1;
    vertice_p->padre_p = NULL;
  }
  for (vertice_p = grafo_p;
       (vertice_p != NULL);
       vertice_p = vertice_p->vertice_succ_p)
    if (vertice_p->colore == bianco)
      visita_grafo_amp(vertice_p);
}

void visita_grafo_amp(vertice_grafo_t *vertice_partenza_p)
{
  vertice_grafo_t      *vertice_p;
  arco_grafo_t         *arco_p;
  elem_lista_vertici_t *uscita_p,
                       *ingresso_p;

  vertice_partenza_p->colore = grigio;
  vertice_partenza_p->distanza = 0;
  uscita_p = ingresso_p = NULL;
  metti_in_coda(&uscita_p,
                &ingresso_p,
                vertice_partenza_p);
  while (uscita_p != NULL)
  {
    vertice_p = togli_da_coda(&uscita_p,
                              &ingresso_p)->valore;
    elabora(vertice_p->valore);
    for (arco_p = vertice_p->lista_archi_p;
         (arco_p != NULL);
         arco_p = arco_p->arco_succ_p)
      if (arco_p->vertice_adiac_p->colore == bianco)
      {
        arco_p->vertice_adiac_p->colore = grigio;
        arco_p->vertice_adiac_p->distanza = vertice_p->distanza + 1;
        arco_p->vertice_adiac_p->padre_p = vertice_p;
```

```
        metti_in_coda(&uscita_p,
                      &ingresso_p,
                      arco_p->vertice_adiac_p);
      }
    vertice_p->colore = nero;
  }
}
```

Poiché ogni vertice entra in coda ed esce dalla coda esattamente una volta e ciascuno dei suoi archi viene attraversato esattamente una volta, la complessità è:

$$T(|V|, |\mathcal{E}|) = O(|V| + |\mathcal{E}|)$$

- Nella visita in profondità, prima si elabora il valore contenuto nel vertice al quale si è giunti, poi si visitano con lo stesso algoritmo i vertici ad esso adiacenti (coincide con la visita in ordine anticipato nel caso di grafi che sono alberi). Nella rappresentazione a lista di adiacenza estendiamo il tipo `vertice_grafo_t` in modo da ricordare per ogni vertice il colore, il tempo al quale viene incontrato per la prima volta, il tempo al quale finisce la visita di tutti i vertici adiacenti e l'indirizzo del padre nell'albero ricoprente della componente connessa a cui appartiene il vertice di partenza:

```
typedef struct vertice_grafo
{
  int                  valore;
  struct vertice_grafo *vertice_succ_p;
  struct arco_grafo    *lista_archi_p;
  colore_t             colore;
  int                  inizio, fine;
  struct vertice_grafo *padre_p;
} vertice_grafo_t;
```

- L'algoritmo di visita in profondità è ricorsivo e quindi fa uso implicito di una pila:

```
void avvia_visita_grafo_prof(vertice_grafo_t *grafo_p)
{
  vertice_grafo_t *vertice_p;
  int             tempo;

  for (vertice_p = grafo_p;
       (vertice_p != NULL);
       vertice_p = vertice_p->vertice_succ_p)
  {
    vertice_p->colore = bianco;
    vertice_p->inizio = vertice_p->fine = -1;
    vertice_p->padre_p = NULL;
  }
  for (vertice_p = grafo_p, tempo = 0;
       (vertice_p != NULL);
       vertice_p = vertice_p->vertice_succ_p)
    if (vertice_p->colore == bianco)
      visita_grafo_prof(vertice_p,
                        &tempo);
}
```

```
void visita_grafo_prof(vertice_grafo_t *vertice_p,
                       int             *tempo)
{
  arco_grafo_t *arco_p;

  vertice_p->colore = grigio;
  vertice_p->inizio = ++(*tempo);
  elabora(vertice_p->valore);
  for (arco_p = vertice_p->lista_archi_p;
       (arco_p != NULL);
       arco_p = arco_p->arco_succ_p)
    if (arco_p->vertice_adiac_p->colore == bianco)
    {
      arco_p->vertice_adiac_p->padre_p = vertice_p;
      visita_grafo_prof(arco_p->vertice_adiac_p,
                        tempo);
    }
  vertice_p->colore = nero;
  vertice_p->fine = ++(*tempo);
}
```

Poiché ogni vertice viene considerato esattamente una volta durante la ricorsione e ciascuno dei suoi archi viene attraversato esattamente una volta, la complessità è:

$$T(|V|, |\mathcal{E}|) = O(|V| + |\mathcal{E}|)$$

- I due precedenti algoritmi di visita possono essere utilizzati anche come algoritmi di ricerca se opportunamente modificati. Ad esempio, dall'algoritmo di visita in profondità si ottiene il seguente algoritmo di ricerca:

```
vertice_grafo_t *avvia_ricerca_in_grafo_prof(vertice_grafo_t *grafo_p,
                                             int             valore)
{
  vertice_grafo_t *vertice_ris_p,
                  *vertice_p;

  for (vertice_p = grafo_p;
       (vertice_p != NULL);
       vertice_p = vertice_p->vertice_succ_p)
    vertice_p->colore = bianco;
  for (vertice_p = grafo_p, vertice_ris_p = NULL;
       ((vertice_p != NULL) && (vertice_ris_p == NULL));
       vertice_p = vertice_p->vertice_succ_p)
    if (vertice_p->colore == bianco)
      vertice_ris_p = cerca_in_grafo_prof(vertice_p,
                                          valore);
  return(vertice_ris_p);
}

vertice_grafo_t *cerca_in_grafo_prof(vertice_grafo_t *vertice_p,
                                     int             valore)
{
  vertice_grafo_t *vertice_ris_p;
  arco_grafo_t    *arco_p;
```

```
  vertice_p->colore = grigio;
  if (vertice_p->valore == valore)
    vertice_ris_p = vertice_p;
  else
    for (arco_p = vertice_p->lista_archi_p, vertice_ris_p = NULL;
         ((arco_p != NULL) && (vertice_ris_p == NULL));
         arco_p = arco_p->arco_succ_p)
      if (arco_p->vertice_adiac_p->colore == bianco)
        vertice_ris_p = cerca_in_grafo_prof(arco_p->vertice_adiac_p,
                                            valore);
  vertice_p->colore = nero;
  return(vertice_ris_p);
}
```

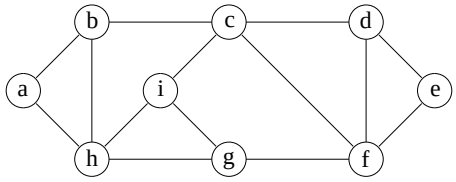
Nel caso pessimo (valore non presente nel grafo) la complessità è:

$$T(|V|, |\mathcal{E}|) = O(|V| + |\mathcal{E}|)$$

mentre nel caso ottimo (valore presente nel primo vertice considerato) la complessità è:

$$T(|V|, |\mathcal{E}|) = O(1)$$

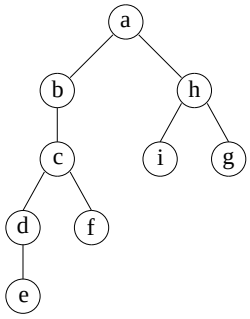
- Esempio: dato il seguente grafo indiretto e connesso



la visita in ampiezza procede nel seguente modo

ordine di visita	contenuto della coda
a	b, h
b	h, c
h	c, i, g
c	i, g, d, f
i	g, d, f
g	d, f
d	f, e
f	e
e	

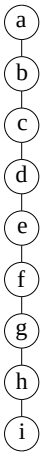
e produce il seguente albero libero ricoprente



mentre la visita in profondità procede nel seguente modo

ordine di visita	inizio / fine
a	1/18
b	2/17
c	3/16
d	4/15
e	5/14
f	6/13
g	7/12
h	8/11
i	9/10

e produce il seguente albero libero ricoprente



■fff_15

7.3 Algoritmo di ordinamento topologico per grafi diretti e aciclici

- Per risolvere il problema dell'ordinamento topologico si può fare uso dell'algoritmo di visita in profondità, considerando i vertici in ordine di tempo decrescente di fine visita.
- Anziché applicare prima l'algoritmo di visita in profondità e poi un algoritmo di ordinamento sui vertici (una volta trasferiti in un array con i loro tempi di fine visita), conviene creare direttamente una lista all'inizio della quale i vertici vengono inseriti man mano che vengono colorati di nero:

```
vertice_grafo_t *avvia_ord_top_grafo(vertice_grafo_t *grafo_p)
{
  vertice_grafo_t *testa_p,
                  *vertice_p;

  for (vertice_p = grafo_p;
       (vertice_p != NULL);
       vertice_p = vertice_p->vertice_succ_p)
    vertice_p->colore = bianco;
  for (vertice_p = grafo_p, testa_p = NULL;
       (vertice_p != NULL);
       vertice_p = vertice_p->vertice_succ_p)
    if (vertice_p->colore == bianco)
      ordina_top_grafo(vertice_p,
                       &testa_p);
  return(testa_p);
}
```

```
void ordina_top_grafo(vertice_grafo_t *vertice_p,
                      vertice_grafo_t **testa_p)
{
  arco_grafo_t    *arco_p;
  vertice_grafo_t *nuovo_elem_p;

  vertice_p->colore = grigio;
  for (arco_p = vertice_p->lista_archi_p;
       (arco_p != NULL);
       arco_p = arco_p->arco_succ_p)
    if (arco_p->vertice_adiac_p->colore == bianco)
      ordina_top_grafo(arco_p->vertice_adiac_p,
                       testa_p);
  vertice_p->colore = nero;
  nuovo_elem_p = (vertice_grafo_t *)malloc(sizeof(vertice_grafo_t));
  nuovo_elem_p->valore = vertice_p->valore;
  nuovo_elem_p->lista_archi_p = vertice_p->lista_archi_p;
  nuovo_elem_p->vertice_succ_p = *testa_p;
  *testa_p = nuovo_elem_p;
}
```

Poiché gli inserimenti avvengono sempre all'inizio della lista, la complessità è:

$$T(|V|, |\mathcal{E}|) = O(|V| + |\mathcal{E}|)$$

7.4 Algoritmo delle componenti fortemente connesse per grafi

- Per risolvere il problema delle componenti fortemente connesse si può procedere effettuando due visite in profondità, una sul grafo dato e una sul grafo trasposto.
- Dopo aver esteso il tipo `vertice_grafo_t` come segue:

```
typedef struct vertice_grafo
{
  int                  valore;
  struct vertice_grafo *vertice_succ_p;
  struct arco_grafo    *lista_archi_p, *lista_archi_inv_p;
  colore_t             colore;
  int                  inizio, fine;
  struct vertice_grafo *padre_p;
} vertice_grafo_t;
```

il grafo trasposto può essere costruito tramite il seguente algoritmo:

```
void trasponi_grafo(vertice_grafo_t *grafo_p)
{
  vertice_grafo_t *vertice_p;
  arco_grafo_t    *arco_p,
                  *nuovo_arco_inv_p,
                  *tmp;

  for (vertice_p = grafo_p;
       (vertice_p != NULL);
       vertice_p = vertice_p->vertice_succ_p)
    for (arco_p = vertice_p->lista_archi_p;
         (arco_p != NULL);
         arco_p = arco_p->arco_succ_p)
```

```
    {
      nuovo_arco_inv_p = (arco_grafo_t *)malloc(sizeof(arco_grafo_t));
      nuovo_arco_inv_p->vertice_adiac_p = vertice_p;
      nuovo_arco_inv_p->arco_succ_p = arco_p->vertice_adiac_p->lista_archi_inv_p;
      arco_p->vertice_adiac_p->lista_archi_inv_p = nuovo_arco_inv_p;
    }
  for (vertice_p = grafo_p;
       (vertice_p != NULL);
       vertice_p = vertice_p->vertice_succ_p)
  {
    tmp = vertice_p->lista_archi_p;
    vertice_p->lista_archi_p = vertice_p->lista_archi_inv_p;
    vertice_p->lista_archi_inv_p = tmp;
  }
}
```

la cui complessità è:

$$T(|V|, |\mathcal{E}|) = O(|V| + |\mathcal{E}|)$$

- Nell'algoritmo per decomporre un grafo nelle sue componenti fortemente connesse, la seconda visita in profondità (quella sul grafo trasposto) deve essere effettuata considerando i vertici in ordine di tempo decrescente di fine prima visita. In altri termini, nella seconda visita i vertici debbono essere considerati in ordine topologico. È possibile dimostrare che così facendo gli alberi ricoprenti massimali costruiti durante la seconda visita individuano esattamente le componenti fortemente connesse:

```
void calcola_comp_fort_conn_grafo(vertice_grafo_t *grafo_p)
{
  vertice_grafo_t *grafo_top_ord_trasp_p;

  grafo_top_ord_trasp_p = avvia_ord_top_grafo(grafo_p);
  trasponi_grafo(grafo_top_ord_trasp_p);
  avvia_visita_grafo_prof(grafo_top_ord_trasp_p);
}
```

La complessità è:

$$T(|V|, |\mathcal{E}|) = O(|V| + |\mathcal{E}|)$$

7.5 Algoritmo di Kruskal

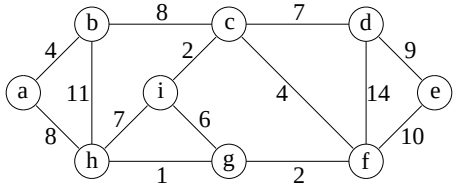
- L'algoritmo di Kruskal (1956) costruisce l'albero ricoprente minimo procedendo come segue. Partendo da tanti alberi liberi quanti sono i singoli vertici del grafo, ad ogni passo esso include l'arco di peso minimo che collega due diversi alberi liberi, fino ad ottenere un singolo albero libero:

```
void kruskal(vertice_grafo_t *grafo_p)
{
  ◁trasforma ogni vertice in un albero libero▷;
  ◁metti gli archi in ordine di peso non decrescente▷;
  for (◁ogni arco considerato nell'ordine di peso non decrescente▷)
    if (◁i due vertici dell'arco stanno in due alberi liberi diversi▷)
      ◁unisci i due alberi liberi aggiungendo l'arco in esame▷;
}
```

- Utilizzando opportune strutture dati la complessità è:

$$T(|V|, |\mathcal{E}|) = O(|\mathcal{E}| \cdot \log |V|)$$

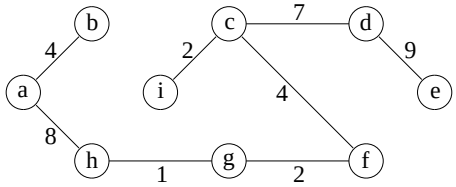
- Esempio: dato il seguente grafo pesato, indiretto e connesso



l'algoritmo di Kruskal procede nel seguente modo

arco	peso	aggiunto?
(h, g)	1	sì
(i, c)	2	sì
(g, f)	2	sì
(a, b)	4	sì
(c, f)	4	sì
(i, g)	6	no
(c, d)	7	sì
(h, i)	7	no
(a, h)	8	sì
(b, c)	8	no
(d, e)	9	sì
(e, f)	10	no
(b, h)	11	no
(d, f)	14	no

e produce il seguente albero ricoprente minimo di peso totale 37



7.6 Algoritmo di Prim

- L'algoritmo di Prim (1957) costruisce l'albero ricoprente minimo procedendo come segue. Partendo da un albero libero costituito da un singolo vertice del grafo, ad ogni passo esso include l'arco di peso minimo che collega un vertice dell'albero libero ad un vertice che non sta ancora nell'albero, fino ad ottenere un albero libero comprendente tutti i vertici del grafo:

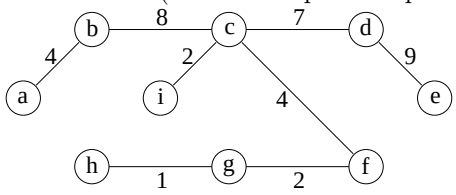
```
void prim(vertice_grafo_t *grafo_p)
{
  ◁costruisci un albero libero composto da un singolo vertice▷;
  ◁costruisci una struttura composta da tutti gli altri vertici,
   memorizzando per ciascuno di essi il peso dell'arco di peso minimo
   che lo collega all'albero libero (∞ se tale arco non esiste)▷;
  while (◁la struttura non è vuota▷)
  {
    ◁rimuovi dalla struttura il vertice il cui peso associato è minimo▷;
    ◁aggiungi all'albero libero tale vertice e il relativo arco di peso minimo▷;
    for (◁ogni vertice rimasto nella struttura▷)
      if (◁il vertice è adiacente a quello rimosso dalla struttura▷)
        ◁ridetermina il peso dell'arco di peso minimo
         che collega il vertice all'albero libero esteso▷;
  }
}
```

- Utilizzando opportune strutture dati la complessità è:
$$T(|V|, |\mathcal{E}|) = O(|\mathcal{E}| \cdot \log |V|)$$

- Esempio: dato lo stesso grafo pesato, indiretto e connesso di prima, l'algoritmo di Prim procede nel seguente modo partendo dal vertice pià sinistra

b	4	–	–	–	–	–	–	–	–
c	∞	8	–	–	–	–	–	–	–
d	∞	∞	7	7	7	7	7	–	–
e	∞	∞	∞	∞	10	10	10	9	–
f	∞	∞	4	4	–	–	–	–	–
g	∞	∞	∞	6	2	–	–	–	–
h	8	8	8	7	7	1	–	–	–
i	∞	∞	2	–	–	–	–	–	–

e produce il seguente albero ricoprente minimo (diverso da quello di prima) di peso totale 37



■flf_16

7.7 Proprietà del percorso più breve

- Consideriamo un grafo diretto e pesato $G = (V, \mathcal{E}, w)$ e due suoi vertici distinti. Se il grafo contiene un ciclo tale che la somma dei pesi dei suoi archi è negativa, il peso del percorso minimo tra i due vertici potrebbe divergere a $-\infty$. Supporremo quindi che il grafo sia privo di cicli di peso negativo.

- Il percorso più breve tra i due vertici non può contenere cicli di peso positivo. Se così non fosse, il percorso in questione non potrebbe essere quello più breve.

- Al fine di evitare anche i cicli di peso nullo, considereremo solo i percorsi semplici. Ciascuno di essi attraversa un numero di vertici non superiore a $|V|$ e quindi un numero di archi non superiore a $|V|-1$.

- Poiché si può dimostrare che ogni percorso tra due vertici contenuto in un percorso minimo tra altri due vertici è esso pure minimo, è possibile costruire il percorso più breve tra i due vertici considerati in modo incrementale, cioè aggiungendo un arco alla volta.

- Ciascuno dei tre algoritmi che vedremo utilizza una rappresentazione a lista di adiacenza e calcola i percorsi più brevi da un vertice dato a tutti gli altri vertici del grafo, producendo un albero con radice nel vertice dato che riporta i percorsi più brevi calcolati. A tal fine il tipo `vertice_grafo_t` viene esteso in modo da ricordare per ogni vertice la stima della distanza minima dal vertice sorgente e l'indirizzo del padre nell'albero dei percorsi più brevi:

```
typedef struct vertice_grafo
{
  int                 valore;
  struct vertice_grafo *vertice_succ_p;
  struct arco_grafo    *lista_archi_p;
  double              distanza_min;
  struct vertice_grafo *padre_p;
} vertice_grafo_t;
```

- Ciascuno dei tre algoritmi effettua la seguente inizializzazione:

```
void inizializza(vertice_grafo_t *grafo_p,
                 vertice_grafo_t *sorgente_p)
{
  vertice_grafo_t *vertice_p;

  for (vertice_p = grafo_p;
       (vertice_p != NULL);
       vertice_p = vertice_p->vertice_succ_p)
  {
    vertice_p->distanza_min = INFINITO;
    vertice_p->padre_p = NULL;
  }
  sorgente_p->distanza_min = 0.0;
}
```

  la cui complessità è:

$$T(|V|, |\mathcal{E}|) = O(|V|)$$

- Indicata con $\delta(s, \_)$ la distanza minima di un vertice dal vertice sorgente s, per ogni $(v, v') \in \mathcal{E}$ risulta $\delta(s, v') \leq \delta(s, v) + w(v, v')$. Di conseguenza, giunti ad un vertice v, per ogni arco (v, v') ciascuno dei tre algoritmi verifica se è possibile migliorare la stima della distanza minima per il vertice v' attraverso il seguente algoritmo:

```
void riduci(arco_grafo_t    *arco_p,
            vertice_grafo_t *vertice_p)   /* vertice da cui l'arco esce */
{
  if (arco_p->vertice_adiac_p->distanza_min > vertice_p->distanza_min + arco_p->peso)
  {
    arco_p->vertice_adiac_p->distanza_min = vertice_p->distanza_min + arco_p->peso;
    arco_p->vertice_adiac_p->padre_p = vertice_p;
  }
}
```

  la cui complessità è:

$$T(|V|, |\mathcal{E}|) = O(1)$$

- Si può dimostrare che ciascuno dei tre algoritmi gode delle seguenti proprietà, le quali implicano la correttezza dell'algoritmo stesso:
  - Per ogni $v \in V$, v`->distanza_min` $\geq \delta(s, v)$ con v`->distanza_min` immutabile non appena diventa uguale a $\delta(s, v)$.
  - Per ogni $v \in V$ non raggiungibile dal vertice sorgente, durante tutta l'esecuzione dell'algoritmo vale v`->distanza_min` $= \delta(s, v) = \infty$.
  - Se $s, \ldots, u, v$ è il percorso minimo da s a v e u`->distanza_min` $= \delta(s, u)$ prima di applicare `riduci` a (u, v), allora v`->distanza_min` $= \delta(s, v)$ dopo aver applicato `riduci` a (u, v).
  - Se $s, v_1, v_2, \ldots, v_k$ è un percorso minimo i cui archi $(s, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ vengono ridotti in quest'ordine, allora v_k`->distanza_min` $= \delta(s, v_k)$.

- Vedremo inoltre un quarto algoritmo, anch'esso operante per miglioramenti successivi della stima della distanza minima, il quale calcola (se esiste) il percorso più breve tra tutte le coppie di vertici del grafo. Diversamente dai tre algoritmi che lo precedono, esso lavora su una rappresentazione a matrice di adiacenza del grafo pesato.

7.8 Algoritmo di Bellman-Ford

- L'algoritmo di Bellman-Ford (1958-1962) consente la presenza di archi di peso negativo e procede alla riduzione sistematica della stima della distanza minima calcolata per ciascun vertice, sfruttando il fatto che un percorso minimo non può contenere più di $|V|-1$ archi (restituisce 1 se non ci sono cicli di peso negativo, 0 altrimenti):

```
int bellman_ford(vertice_grafo_t *grafo_p,
                 vertice_grafo_t *sorgente_p,
                 int             numero_vertici)
{
  vertice_grafo_t *vertice_p;
  arco_grafo_t    *arco_p;
  int             cicli_negativi_assenti,
                  i;

  inizializza(grafo_p,
              sorgente_p);
  for (i = 1;
       (i < numero_vertici);
       i++)
    for (vertice_p = grafo_p;
         (vertice_p != NULL);
         vertice_p = vertice_p->vertice_succ_p)
      for (arco_p = vertice_p->lista_archi_p;
           (arco_p != NULL);
           arco_p = arco_p->arco_succ_p)
        riduci(arco_p,
               vertice_p);
  for (vertice_p = grafo_p, cicli_negativi_assenti = 1;
       ((vertice_p != NULL) && cicli_negativi_assenti);
       vertice_p = vertice_p->vertice_succ_p)
    for (arco_p = vertice_p->lista_archi_p;
         ((arco_p != NULL) && cicli_negativi_assenti);
         arco_p = arco_p->arco_succ_p)
      if (arco_p->vertice_adiac_p->distanza_min > vertice_p->distanza_min + arco_p->peso)
        cicli_negativi_assenti = 0;
  return(cicli_negativi_assenti);
}
```

La complessità è:

$$T(|V|, |\mathcal{E}|) = O(|V| \cdot (|V| + |\mathcal{E}|))$$

- Il seguente algoritmo è una variante più efficiente applicabile solo a grafi diretti e aciclici, il quale effettua una sola passata di riduzioni anziché $|V|-1$ considerando i vertici in ordine topologico:

```
void bellman_ford_gda(vertice_grafo_t *grafo_p,
                      vertice_grafo_t *sorgente_p)
{
  vertice_grafo_t *grafo_top_ord_p,
                  *vertice_p;
  arco_grafo_t    *arco_p;
```

```
  grafo_top_ord_p = avvia_ord_top_grafo(grafo_p);
  inizializza(grafo_top_ord_p,
              sorgente_p);
  for (vertice_p = grafo_top_ord_p;
       (vertice_p != NULL);
       vertice_p = vertice_p->vertice_succ_p)
    for (arco_p = vertice_p->lista_archi_p;
         (arco_p != NULL);
         arco_p = arco_p->arco_succ_p)
      riduci(arco_p,
             vertice_p);
}
```

La complessità è:

$$T(|V|, |\mathcal{E}|) = O(|V| + |\mathcal{E}|)$$ ■fff_17

7.9 Algoritmo di Dijkstra

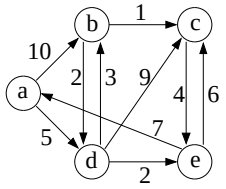
- L'algoritmo di Dijkstra (1959) è più efficiente di quello di Bellman-Ford, ma si applica solo a grafi privi di archi di peso negativo (limitazione non necessariamente restrittiva nella pratica):

```
void dijkstra(vertice_grafo_t *grafo_p,
              vertice_grafo_t *sorgente_p)
{
  vertice_grafo_t *vertice_p;
  arco_grafo_t    *arco_p;

  inizializza(grafo_p,
              sorgente_p);
  ◁costruisci un insieme per i vertici già considerati (inizialmente vuoto)▷;
  ◁costruisci una struttura per i vertici da considerare (inizialmente tutti)▷;
  while (◁la struttura non è vuota▷)
  {
    ◁rimuovi dalla struttura il vertice vertice_p con distanza_min minima▷;
    ◁inserisci vertice_p nell'insieme dei vertici già considerati▷;
    for (arco_p = vertice_p->lista_archi_p;
         (arco_p != NULL);
         arco_p = arco_p->arco_succ_p)
      if (◁arco_p->vertice_adiac_p non è nell'insieme dei vertici già considerati▷)
        riduci(arco_p,
               vertice_p);
  }
}
```

- Utilizzando opportune strutture dati la complessità è:

$$T(|V|, |\mathcal{E}|) = O(|V| \cdot \log |V| + |\mathcal{E}|)$$

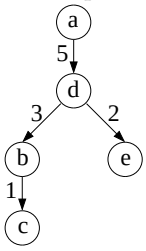
- Esempio: dato il seguente grafo diretto e pesato



l'algoritmo di Dijkstra procede nel seguente modo quando il vertice sorgente è quello più a sinistra

a	0	–	–	–	–	–
b	∞	10	8	8	–	–
c	∞	∞	14	13	9	–
d	∞	5	–	–	–	–
e	∞	∞	7	–	–	–

e produce il seguente albero dei percorsi più brevi che partono dal vertice sorgente



7.10 Algoritmo di Floyd-Warshall

- L'algoritmo di Floyd-Warshall (1962) calcola il percorso più breve tra tutte le coppie di vertici del grafo – anziché il percorso più breve tra una singola coppia di vertici – avvalendosi di una rappresentazione del grafo a matrice di adiacenza – anziché a lista di adiacenza come i tre algoritmi precedenti.
- L'algoritmo di Floyd-Warshall si basa sulla soluzione del problema della chiusura transitiva di un grafo diretto, ovvero del problema di determinare l'esistenza di un percorso per ogni coppia di vertici.
- Poiché un percorso semplice tra due vertici attraversa al più $n = |V|$ vertici, il problema della chiusura transitiva può essere risolto calcolando una sequenza di n matrici $T^{(1)}, T^{(2)}, \ldots, T^{(n)}$ a partire dalla matrice $T^{(0)}$ così definita:
$$t_{i,j}^{(0)} = \begin{cases} 1 & \text{se } i = j \vee (v_i, v_j) \in \mathcal{E} \\ 0 & \text{se } i \neq j \wedge (v_i, v_j) \notin \mathcal{E} \end{cases}$$
- Per ogni $k = 1, \ldots, n$, esiste un percorso da v_i a v_j che passa per vertici di indice $\leq k$ se esiste un percorso da v_i a v_j che passa per vertici di indice $\leq k-1$ oppure se esistono un percorso da v_i a v_k e un percorso da v_k a v_j che passano per vertici di indice $\leq k-1$. Indicata con $t_{i,j}^{(k)}$ l'esistenza di un percorso da v_i a v_j che passa per vertici di indice $\leq k$, la matrice $T^{(k)}$ viene allora calcolata come segue:
$$t_{i,j}^{(k)} = t_{i,j}^{(k-1)} \vee (t_{i,k}^{(k-1)} \wedge t_{k,j}^{(k-1)})$$
- Dal punto di vista dell'occupazione di memoria, è sufficiente una sola matrice perché risulta:
$$t_{i,k}^{(k)} = t_{i,k}^{(k-1)} \vee (t_{i,k}^{(k-1)} \wedge t_{k,k}^{(k-1)}) = t_{i,k}^{(k-1)}$$
$$t_{k,j}^{(k)} = t_{k,j}^{(k-1)} \vee (t_{k,k}^{(k-1)} \wedge t_{k,j}^{(k-1)}) = t_{k,j}^{(k-1)}$$
quindi al passo k non è necessario tener traccia dei vecchi valori di $t_{i,k}$ e $t_{k,j}$.
- L'algoritmo per il calcolo della chiusura transitiva è quindi il seguente:

```
void calcola_chiusura_trans_grafo(int grafo[][NUMERO_VERTICI],
                                  int chiusura[][NUMERO_VERTICI])
{
  int i,
      j,
      k;
```

```
  for (i = 0;
       (i < NUMERO_VERTICI);
       i++)
    for (j = 0;
         (j < NUMERO_VERTICI);
         j++)
      if ((i == j) || (grafo[i][j] == 1))
        chiusura[i][j] = 1;
      else
        chiusura[i][j] = 0;
  for (k = 0;
       (k < NUMERO_VERTICI);
       k++)
    for (i = 0;
         (i < NUMERO_VERTICI);
         i++)
      for (j = 0;
           (j < NUMERO_VERTICI);
           j++)
        chiusura[i][j] = chiusura[i][j] || (chiusura[i][k] && chiusura[k][j]);
}
```

e la sua complessità è:

$$T(|V|, |\mathcal{E}|) = O(|V|^3)$$

- Indicata con $dist_{i,j}^{(k)}$ la distanza minima di v_j da v_i passando per vertici di indice $\leq k$, l'algoritmo di Floyd-Warshall si basa sull'analoga relazione:

$$dist_{i,j}^{(k)} = \min(dist_{i,j}^{(k-1)}, dist_{i,k}^{(k-1)} + dist_{k,j}^{(k-1)})$$

da cui:

$$dist_{i,k}^{(k)} = dist_{i,k}^{(k-1)}$$
$$dist_{k,j}^{(k)} = dist_{k,j}^{(k-1)}$$

per calcolare la matrice delle distanze minime e la matrice dei padri negli alberi dei percorsi più brevi:

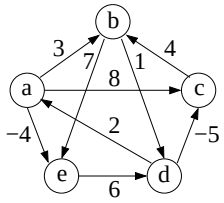
```
void floyd_warshall(double grafo[][NUMERO_VERTICI],
                    double dist[][NUMERO_VERTICI],
                    int    padre[][NUMERO_VERTICI])
{
  int i,
      j,
      k;

  for (i = 0;
       (i < NUMERO_VERTICI);
       i++)
    for (j = 0;
         (j < NUMERO_VERTICI);
         j++)
    {
      dist[i][j] = grafo[i][j];
      padre[i][j] = (grafo[i][j] != INFINITO)?
                     i:
                     -1;
    }
```

```
  for (k = 0;
       (k < NUMERO_VERTICI);
       k++)
    for (i = 0;
         (i < NUMERO_VERTICI);
         i++)
      for (j = 0;
           (j < NUMERO_VERTICI);
           j++)
        if (dist[i][j] > dist[i][k] + dist[k][j])
        {
          dist[i][j] = dist[i][k] + dist[k][j];
          padre[i][j] = padre[k][j];
        }
}
```

La complessità è:

$$T(|V|, |\mathcal{E}|) = O(|V|^3)$$

- Esempio: dato il seguente grafo diretto e pesato



e attribuiti gli indici ai suoi vertici seguendo l'ordine alfabetico, l'algoritmo di Floyd-Warshall calcola la seguente sequenza di matrici delle distanze minime e dei padri negli alberi dei percorsi più brevi

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad P^{(0)} = \begin{pmatrix} a & a & a & - & a \\ - & b & - & b & b \\ - & c & c & - & - \\ d & - & d & d & - \\ - & - & - & e & e \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad P^{(1)} = \begin{pmatrix} a & a & a & - & a \\ - & b & - & b & b \\ - & c & c & - & - \\ d & a & d & d & a \\ - & - & - & e & e \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad P^{(2)} = \begin{pmatrix} a & a & a & b & a \\ - & b & - & b & b \\ - & c & c & b & b \\ d & a & d & d & a \\ - & - & - & e & e \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad P^{(3)} = \begin{pmatrix} a & a & a & b & a \\ - & b & - & b & b \\ - & c & c & b & b \\ d & c & d & d & a \\ - & - & - & e & e \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad P^{(4)} = \begin{pmatrix} \mathrm{a} & \mathrm{a} & \mathrm{d} & \mathrm{b} & \mathrm{a} \\ \mathrm{d} & \mathrm{b} & \mathrm{d} & \mathrm{b} & \mathrm{a} \\ \mathrm{d} & \mathrm{c} & \mathrm{c} & \mathrm{b} & \mathrm{a} \\ \mathrm{d} & \mathrm{c} & \mathrm{d} & \mathrm{d} & \mathrm{a} \\ \mathrm{d} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{e} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \qquad P^{(5)} = \begin{pmatrix} \mathrm{a} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{a} \\ \mathrm{d} & \mathrm{b} & \mathrm{d} & \mathrm{b} & \mathrm{a} \\ \mathrm{d} & \mathrm{c} & \mathrm{c} & \mathrm{b} & \mathrm{a} \\ \mathrm{d} & \mathrm{c} & \mathrm{d} & \mathrm{d} & \mathrm{a} \\ \mathrm{d} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{e} \end{pmatrix}$$

■fff_18

Capitolo 8

Tecniche algoritmiche

8.1 Tecnica del divide et impera

- La tecnica del divide et impera è una tecnica di progettazione di algoritmi che si articola in tre fasi:
  - Dividere l'istanza dei dati di ingresso del problema in più sottoistanze disgiunte (divide).
  - Risolvere ricorsivamente il problema su ciascuna sottoistanza considerata separatamente.
  - Combinare le soluzioni delle sottoistanze per ottenere la soluzione dell'istanza data (impera).
- L'efficacia della tecnica del divide et impera dipende dal metodo di decomposizione delle istanze e dal metodo di ricomposizione delle soluzioni parziali. In particolare, la sua efficacia in termini di complessità computazionale dipende in modo critico dalla capacità di effettuare un partizionamento bilanciato dell'istanza dei dati di ingresso nella prima fase.
- La tecnica del divide et impera lavora dall'alto verso il basso, in quanto procede dall'istanza generale alle sue sottoistanze.
- Esempi di applicazione visti nel corso:
  - L'algoritmo ricorsivo per calcolare il massimo e il submassimo di un insieme di n interi (Sez. 3.3) divide l'insieme in due sottoinsiemi disgiunti di $n/2$ interi, calcola ricorsivamente il massimo e il submassimo di ciascuno dei due sottoinsiemi considerati separatamente, e infine confronta i risultati calcolati per i due sottoinsiemi in modo da determinare il massimo e il submassimo dell'insieme originario.
  - L'algoritmo di ricerca binaria per un array ordinato di n elementi (Sez. 4.4) messo in versione ricorsiva divide l'array in due sottoarray disgiunti di $n/2$ elementi e poi continua ricorsivamente in uno solo dei due sottoarray. Un modo di procedere analogo si riscontra anche negli algoritmi di ricerca, inserimento e rimozione per un albero binario di ricerca di n nodi (Sez. 6.3) messi in versione ricorsiva.
  - Mergesort per un array di n elementi (Sez. 4.9) divide l'array in due sottoarray disgiunti di $n/2$ elementi, ordina ricorsivamente ciascuno dei due sottoarray considerati separatamente, e infine fonde ordinatamente i due sottoarray.
  - Quicksort (Sez. 4.10) divide l'array in tre sottoarray disgiunti composti da elementi contenenti valori $\leq$, $=$ o $\geq$ del pivot rispettivamente, ordina ricorsivamente il primo e il terzo sottoarray considerati separatamente, e infine concatena i tre sottoarray. Il fatto che il primo e il terzo sottoarray contengano circa lo stesso numero di elementi dipende in modo critico dalla scelta del pivot.

8.2 Programmazione dinamica

- La programmazione dinamica è una tecnica di progettazione di algoritmi che, diversamente dal divide et impera, lavora dal basso verso l'alto, in quanto considera implicitamente dei sottoproblemi definiti su opportune istanze dei dati di ingresso e procede dai sottoproblemi più piccoli ai sottoproblemi più grandi. Nel fare questo, la programmazione dinamica memorizza esplicitamente le soluzioni dei sottoproblemi in un'apposita tabella, diversamente da quanto accade con la tecnica del divide et impera.

- La programmazione dinamica si articola in quattro fasi:

  - Identificare dei sottoproblemi del problema originario.
  - Predisporre una tabella per memorizzare dinamicamente le soluzioni dei sottoproblemi, definendo i valori iniziali degli elementi di questa tabella che corrispondono ai sottoproblemi più semplici.
  - Avanzare in modo opportuno sulla tabella calcolando la soluzione di un sottoproblema sulla base delle soluzioni dei sottoproblemi precedentemente risolti e quindi già presenti nella tabella.
  - Restituire la soluzione del problema originario, la quale si troverà memorizzata in un particolare elemento della tabella.

- Qualora la tabella non occupi uno spazio eccessivo, questa tecnica risulta vantaggiosa nel caso di un problema in cui si presentano sottoproblemi che non sono del tutto indipendenti, cioè in cui uno stesso sottoproblema può apparire più volte durante la risoluzione del problema originario. Poiché la programmazione dinamica utilizza una tabella per memorizzare le soluzioni dei sottoproblemi incontrati durante l'esecuzione dell'algoritmo, quando si incontra di nuovo un certo sottoproblema basterà esaminare l'elemento corrispondente nella tabella, senza quindi dover risolvere daccapo il sottoproblema.

- Esempi di applicazione visti nel corso:

  - Dal punto di vista concettuale, l'algoritmo iterativo per calcolare il fattoriale di $n \geq 1$ (Sez. 3.2) identifica come sottoproblemi quelli di calcolare $m!$ per ogni $1 \leq m < n$, predispone una tabella di n elementi inizializzando ad 1 il primo elemento, avanza di un elemento alla volta determinandone il valore come prodotto tra il valore dell'elemento precedente e l'indice dell'elemento corrente, e restituisce come risultato il valore dell'ultimo elemento. In pratica, non serve predisporre una tabella in quanto è sufficiente un'unica variabile il cui valore deve essere moltiplicato ad ogni passo di avanzamento per l'indice del corrispondente elemento della tabella.
  - Dal punto di vista concettuale, l'algoritmo iterativo per calcolare l'n-esimo numero di Fibonacci per $n \geq 1$ (Sez. 3.2) identifica come sottoproblemi quelli di calcolare l'm-esimo numero di Fibonacci per ogni $1 \leq m < n$, predispone una tabella di n elementi inizializzando ad 1 i primi due elementi, avanza di un elemento alla volta determinandone il valore come somma dei valori dei due elementi precedenti, e restituisce come risultato il valore dell'ultimo elemento. In pratica, non serve predisporre una tabella in quanto sono sufficienti due variabili per contenere i valori degli ultimi due elementi considerati nella tabella e una terza variabile in cui porre la somma dei valori delle prime due variabili. Poiché evita di fare più volte gli stessi calcoli, questo algoritmo ha una complessità molto inferiore rispetto a quella dell'algoritmo ricorsivo per calcolare l'n-esimo numero di Fibonacci (Sez. 3.3).
  - Dal punto di vista concettuale, l'algoritmo di Floyd-Warshall per un grafo di n vertici (Sez. 7.10) identifica come sottoproblemi quelli di calcolare i percorsi più brevi tra tutte le coppie di vertici del grafo che passano per vertici di indice $\leq k$ per ogni $1 \leq k < n$, predispone una tabella di $n+1$ elementi (che sono delle matrici quadrate di ordine n) inizializzando il primo elemento in base alla matrice di adiacenza del grafo, avanza di un elemento alla volta determinandone il valore tramite una specifica formula applicata al valore dell'elemento precedente, e restituisce come risultato il valore dell'ultimo elemento. In pratica, non serve predisporre una tabella di matrici in quanto è sufficiente un'unica matrice per effettuare tutti i calcoli.

8.3 Tecnica golosa

- La tecnica golosa si applica a quei problemi di ottimizzazione per i quali la soluzione ottima globale per un'istanza dei dati di ingresso può essere ottenuta incrementalmente compiendo ogni volta che è richiesto delle scelte che sono localmente ottime.

- Questo è lo schema di un algoritmo goloso:

```
soluzione risolvi_goloso(problema p)
{
  soluzione    s;
  insieme_cand ic;
  candidato    c;

  inizializza(s);
  ic = costruisci_insieme_cand(p);
  while (!ottima(s) && !vuoto(ic))
  {
    c = scegli_cand_ottimo(ic);
    ic = rimuovi(ic, c);
    if (ammissibile(unione(s, c)))
      s = inserisci(s, c);
  }
  return(s);
}
```

- Esempi di applicazione visti nel corso:

  – L'algoritmo di Kruskal (Sez. 7.5) prende come soluzione iniziale l'insieme dei singoli vertici del grafo, considera come candidati i singoli archi del grafo, sceglie ad ogni passo l'arco di peso minimo tra quelli rimasti, e ritiene ammissibile l'inserimento di tale arco nella soluzione se i due vertici dell'arco appartengono a due alberi liberi diversi tra quelli costruiti fino a quel momento.

  – L'algoritmo di Prim (Sez. 7.6) prende come soluzione iniziale un singolo vertice del grafo, considera come candidati i rimanenti vertici del grafo, sceglie ad ogni passo il vertice tra quelli rimasti che sta a distanza minima dall'albero libero costruito fino a quel momento, e inserisce comunque tale vertice con il relativo arco nell'albero libero.

  – L'algoritmo di Dijkstra (Sez. 7.9) prende come soluzione iniziale un insieme vuoto (nel quale verrà inserito come primo elemento il vertice sorgente), considera come candidati i singoli vertici del grafo, sceglie ad ogni passo il vertice tra quelli rimasti che sta a distanza minima dal vertice sorgente, e inserisce comunque tale vertice nell'albero dei percorsi più brevi costruito fino a quel momento. ■fff_19

8.4 Tecnica per tentativi e revoche

- Esistono problemi per i quali non si riesce a trovare una regola fissa di computazione per determinare la soluzione. Per tali problemi non si può che procedere per tentativi, ritornando indietro sui propri passi ogni volta che il tentativo corrente fallisce.

- Concettualmente, un algoritmo per tentativi e revoche visita in profondità il suo albero di decisione, il quale comprende tutte le scelte che possono presentarsi durante l'esecuzione. Di conseguenza, la complessità asintotica nel caso pessimo (fallimento di tutti i tentativi) è inevitabilmente esponenziale.

- Gli algoritmi per tentativi e revoche possono anche essere usati per simulare gli algoritmi non deterministici. Poiché ad ogni punto di scelta gli algoritmi non deterministici esplorano tutte le vie contemporaneamente (riportando successo se trovano la soluzione lungo una via, fallimento se non trovano la soluzione lungo nessuna via), essi non sono implementabili in pratica a meno di conoscere il numero massimo di esecutori paralleli di cui necessitano. Gli algoritmi per tentativi e revoche possono invece essere implementati (seppure in maniera inefficiente) in quanto esplorano una sola via alla volta.
- Questo è lo schema di un algoritmo per tentativi e revoche che calcola una singola soluzione:

```
void tenta(problema  p,
           soluzione *s)
{
  insieme_cand ic;
  candidato    c;

  ic = costruisci_insieme_cand(p, *s);
  while (!vuoto(ic) && !completata(*s))
  {
    c = scegli_cand(ic);
    ic = rimuovi(ic, c);
    if (ammissibile(unione(*s, c)))
    {
      *s = inserisci(*s, c);
      if (!completa(*s))
      {
        tenta(p, s);
        if (!completata(*s))
          *s = rimuovi(*s, c);
      }
    }
  }
}
```

- Lo schema di un algoritmo per tentativi e revoche che deve calcolare tutte le soluzioni può essere ottenuto dallo schema precedente cambiando l'istruzione `while` come segue:

```
while (!vuoto(ic))
{
  c = scegli_cand(ic);
  ic = rimuovi(ic, c);
  if (ammissibile(unione(*s, c)))
  {
    *s = inserisci(*s, c);
    if (!completa(*s))
      tenta(p, s);
    else
      stampa(*s);
    *s = rimuovi(*s, c);
  }
}
```

- Esempi di applicazione tratti dal mondo degli scacchi:
  - Problema del giro del cavallo: partendo da una data casella di una scacchiera dove è inizialmente posizionato un cavallo, stabilire se esiste un percorso del cavallo di 63 mosse ad L che visita ogni casella esattamente una volta.
    Rappresentata la scacchiera come una matrice bidimensionale i cui elementi contengono i numeri seriali delle mosse effettuate dal cavallo, le 8 possibili variazioni delle coordinate del cavallo determinate dal compimento di una mossa ad "L" possono essere formalizzate come segue:

```
int mossa_x[8] = {2, 1, -1, -2, -2, -1, 1, 2},
    mossa_y[8] = {1, 2, 2, 1, -1, -2, -2, -1};
```

    Il seguente algoritmo per tentativi e revoche calcola una soluzione (se esiste) al problema del giro del cavallo per una data casella iniziale:

```
int tenta_mossa_cavallo(int numero_mossa,
                        int x,
                        int y,
                        int scacchiera[][8])
{
  int soluz_completata,
      indice_mossa,
      nuova_x,
      nuova_y;

  soluz_completata = 0;
  indice_mossa = -1;
  while ((indice_mossa < 7) && !soluz_completata)
  {
    indice_mossa++;
    nuova_x = x + mossa_x[indice_mossa];
    nuova_y = y + mossa_y[indice_mossa];
    if ((0 <= nuova_x) && (nuova_x <= 7) &&
        (0 <= nuova_y) && (nuova_y <= 7) &&
        (scacchiera[nuova_x][nuova_y] == 0))
    {
      scacchiera[nuova_x][nuova_y] = numero_mossa;
      if (numero_mossa < 64)
      {
        soluz_completata = tenta_mossa_cavallo(numero_mossa + 1,
                                               nuova_x,
                                               nuova_y,
                                               scacchiera);
        if (!soluz_completata)
          scacchiera[nuova_x][nuova_y] = 0;
      }
      else
        soluz_completata = 1;
    }
  }
  return(soluz_completata);
}
```

– Problema delle regine: posizionare 8 regine su una scacchiera in maniera tale che non ci siano due regine sulla stessa riga, colonna o diagonale.
Assumendo che la i-esima regina giaccia nell'i-esima colonna, è sufficiente usare un array chiamato `regina` per memorizzare la riga in cui si trova ciascuna regina, un array chiamato `riga_libera` per memorizzare per ogni riga se nessuna regina giace su di essa, un array chiamato `diag_sx_libera` per memorizzare per ogni diagonale $\swarrow$ se nessuna regina giace su di essa, e un array chiamato `diag_dx_libera` per memorizzare per ogni diagonale $\searrow$ se nessuna regina giace su di essa.
Osserviamo che ognuna delle 15 diagonali con la stessa orientazione può essere individuata in modo univoco attraverso un numero. Infatti, per ogni diagonale $\swarrow$ la somma delle coordinate di ciascuna delle sue caselle è costante. Analogamente, per ogni diagonale $\searrow$ la differenza delle coordinate di ciascuna delle sue caselle è costante.
Il seguente algoritmo per tentativi e revoche (nel quale viene usata per comodità un'istruzione `for` invece di un'istruzione `while`) calcola tutte le soluzioni del problema delle regine:

```
#define DIAG_DX 7   /* fattore per trasformare -7..7 in 0..14 */

void tenta_posiz_regina(int colonna,
                        int regina[],
                        int riga_libera[],
                        int diag_sx_libera[],
                        int diag_dx_libera[])
{
  int riga;

  for (riga = 0;
       (riga <= 7);
       riga++)
    if (riga_libera[riga] &&
        diag_sx_libera[riga + colonna] &&
        diag_dx_libera[riga - colonna + DIAG_DX])
    {
      regina[colonna] = riga;
      riga_libera[riga] = 0;
      diag_sx_libera[riga + colonna] = 0;
      diag_dx_libera[riga - colonna + DIAG_DX] = 0;
      if (colonna < 7)
        tenta_posiz_regina(colonna + 1,
                           regina,
                           riga_libera,
                           diag_sx_libera,
                           diag_dx_libera);
      else
        stampa(regina);
      regina[colonna] = -1;
      riga_libera[riga] = 1;
      diag_sx_libera[riga + colonna] = 1;
      diag_dx_libera[riga - colonna + DIAG_DX] = 1;
    }
}
```

■flf_20

Capitolo 9

Correttezza degli algoritmi

9.1 Triple di Hoare

- Dati un problema decidibile e un algoritmo che si suppone risolvere il problema, si pone la questione di verificare se l'algoritmo è corretto rispetto al problema.
- Ciò richiede di definire formalmente cosa l'algoritmo calcola. L'approccio tradizionalmente adottato nel caso di un algoritmo sequenziale è quello di definirne il significato mediante una funzione che descrive l'effetto dell'esecuzione dell'algoritmo sul contenuto della memoria.
- In questo contesto, per stato della computazione si intende il contenuto della memoria ad un certo punto dell'esecuzione dell'algoritmo. La funzione che rappresenta il significato dell'algoritmo descrive quale sia lo stato finale della computazione a fronte dello stato iniziale della computazione determinato da una generica istanza dei dati di ingresso.
- L'approccio assiomatico di Hoare alla verifica di correttezza (1969) si basa sull'idea di annotare gli algoritmi con dei predicati per esprimere le proprietà che sono valide nei vari stati della computazione.
- Si dice tripla di Hoare una tripla della seguente forma:
$$\{Q\}\ S\ \{R\}$$
dove Q è un predicato detto precondizione, S è un'istruzione ed R è un predicato detto postcondizione.
- La tripla $\{Q\}\ S\ \{R\}$ è vera se l'esecuzione dell'istruzione S inizia in uno stato della computazione in cui Q è soddisfatta e termina raggiungendo uno stato della computazione in cui R è soddisfatta.

9.2 Determinazione della precondizione più debole

- Nella pratica, data una tripla di Hoare $\{Q\}\ S\ \{R\}$ in cui S è un intero algoritmo, S è ovviamente noto come pure è nota la postcondizione R, la quale rappresenta in sostanza il risultato che si vuole ottenere alla fine dell'esecuzione dell'algoritmo. La precondizione Q è invece ignota.
- Verificare la correttezza di un algoritmo S che si prefigge di calcolare un risultato R consiste quindi nel determinare se esiste un predicato Q che risolve la seguente equazione logica:
$$\{Q\}\ S\ \{R\} \equiv \mathit{vero}$$
- Poiché l'equazione logica riportata sopra potrebbe ammettere più soluzioni, può essere utile concentrarsi sulla determinazione della precondizione più debole (nel senso di meno vincolante), cioè la precondizione logicamente implicata da tutte le altre precondizioni che risolvono l'equazione logica. Dati un algoritmo S e una postcondizione R, denotiamo con $wp(S, R)$ la precondizione più debole rispetto a S ed R.

- Premesso che *vero* è soddisfatto da ogni stato della computazione mentre *falso* non è soddisfatto da nessuno stato della computazione, dati un algoritmo S e una postcondizione R si hanno i tre seguenti casi:
  - Se $wp(S, R) = vero$, allora qualunque sia la precondizione Q risulta che la tripla di Hoare $\{Q\}\ S\ \{R\}$ è vera. Infatti $Q \Longrightarrow vero$ per ogni predicato Q. In questo caso, l'algoritmo è sempre corretto rispetto al problema, cioè è corretto a prescindere da quale sia lo stato iniziale della computazione.
  - Se $wp(S, R) = falso$, allora qualunque sia la precondizione Q risulta che la tripla di Hoare $\{Q\}\ S\ \{R\}$ è falsa. Infatti $Q \Longrightarrow falso$ se Q non è soddisfatto nello stato iniziale della computazione, mentre $Q \not\Longrightarrow falso$ (quindi Q non può essere una soluzione) se Q è soddisfatto nello stato iniziale della computazione. In questo caso, l'algoritmo non è mai corretto rispetto al problema, cioè non è corretto a prescindere da quale sia lo stato iniziale della computazione.
  - Se $wp(S, R) \notin \{vero, falso\}$, allora la correttezza dell'algoritmo rispetto al problema potrebbe dipendere dallo stato iniziale della computazione.

- Dato un algoritmo S, $wp(S, \_)$ soddisfa le seguenti proprietà:

$$\begin{array}{rcl} wp(S, falso) & \equiv & falso \\ wp(S, R_1 \wedge R_2) & \equiv & wp(S, R_1) \wedge wp(S, R_2) \\ (R_1 \Longrightarrow R_2) & \Longrightarrow & (wp(S, R_1) \Longrightarrow wp(S, R_2)) \\ \{Q\}\ S\ \{R'\} \wedge (R' \Longrightarrow R) & \Longrightarrow & \{Q\}\ S\ \{R\} \end{array}$$

- Dato un algoritmo S privo di iterazione e ricorsione e data una postcondizione R, $wp(S, R)$ può essere determinata per induzione sulla struttura di S nel seguente modo proposto da Dijkstra:
  - Se S è un'istruzione di assegnamento `x = e;` si applica la seguente regola di retropropagazione:
    $$wp(S, R) = R_{\mathtt{x,e}}$$
    dove $R_{\mathtt{x,e}}$ è il predicato ottenuto da R sostituendo tutte le occorrenze di `x` con `e`.
  - Se S è un'istruzione di selezione `if` (β) S_1 `else` S_2 allora:
    $$wp(S, R) = (\beta \Longrightarrow wp(S_1, R) \wedge \neg\beta \Longrightarrow wp(S_2, R))$$
  - Se S è una sequenza di istruzioni $S_1 S_2$ allora:
    $$wp(S, R) = wp(S_1, wp(S_2, R))$$

- Come indica la regola per la sequenza di istruzioni, il calcolo della precondizione più debole procede andando a ritroso a partire dalla postcondizione e dall'ultima istruzione dell'algoritmo.

- Esempi:
  - La correttezza di un algoritmo può dipendere dallo stato iniziale della computazione. Data l'istruzione di assegnamento:

    ```
    x = x + 1;
    ```

    e la postcondizione:
    $$x < 1$$
    l'ottenimento del risultato prefissato dipende ovviamente dal valore di `x` prima che venga eseguita l'istruzione. Infatti la precondizione più debole risulta essere:
    $$(x < 1)_{\mathtt{x,x+1}} = (x + 1 < 1) \equiv (x < 0)$$
  - Il seguente algoritmo per determinare quella tra due variabili che contiene il valore minimo:

    ```
    if (x <= y)
      z = x;
    else
      z = y;
    ```

è sempre corretto perché, avendo come postcondizione:

$$z = \min(x, y)$$

la precondizione più debole risulta essere *vero* in quanto:

$$((x \leq y) \Longrightarrow (z = \min(x, y))_{\mathtt{z,x}}) = ((x \leq y) \Longrightarrow (x = \min(x, y))) \equiv \mathit{vero}$$
$$((x > y) \Longrightarrow (z = \min(x, y))_{\mathtt{z,y}}) = ((x > y) \Longrightarrow (y = \min(x, y))) \equiv \mathit{vero}$$
$$\mathit{vero} \wedge \mathit{vero} \equiv \mathit{vero}$$

– Il seguente algoritmo per scambiare i valori di due variabili:

```
tmp = x;
x = y;
y = tmp;
```

è sempre corretto perché, avendo come postcondizione:

$$x = Y \wedge y = X$$

la precondizione più debole risulta essere il generico predicato:

$$x = X \wedge y = Y$$

in quanto:

$$(x = Y \wedge y = X)_{\mathtt{y,tmp}} = (x = Y \wedge \mathit{tmp} = X)$$
$$(x = Y \wedge \mathit{tmp} = X)_{\mathtt{x,y}} = (y = Y \wedge \mathit{tmp} = X)$$
$$(y = Y \wedge \mathit{tmp} = X)_{\mathtt{tmp,x}} = (y = Y \wedge x = X) \equiv (x = X \wedge y = Y)$$

■fff_21

9.3 Verifica della correttezza di algoritmi iterativi

- Per verificare mediante triple di Hoare la correttezza di un'istruzione di ripetizione bisogna individuare un invariante di ciclo, cioè un predicato che è soddisfatto sia nello stato iniziale della computazione che nello stato finale della computazione di ciascuna iterazione, assieme ad una funzione descrescente che misura il tempo residuo alla fine dell'esecuzione dell'istruzione di ripetizione basandosi sulle variabili di controllo del ciclo. Sotto certe ipotesi, l'invariante di ciclo è la precondizione dell'istruzione di ripetizione.

- Teorema dell'invariante di ciclo: data un'istruzione di ripetizione `while (`β`)` S, se esistono un predicato P e una funzione intera tr tali che:

$$\{P \wedge \beta\}\ S\ \{P\} \qquad \text{(invarianza)}$$
$$\{P \wedge \beta \wedge tr(i) = T\}\ S\ \{tr(i+1) < T\} \qquad \text{(progresso)}$$
$$(P \wedge tr(i) \leq 0) \Longrightarrow \neg\beta \qquad \text{(limitatezza)}$$

allora:

$$\{P\}\ \texttt{while (}\beta\texttt{)}\ S\ \{P \wedge \neg\beta\}$$

- Corollario: data un'istruzione di ripetizione `while (`β`)` S per la quale è possibile trovare un invariante di ciclo P e data una postcondizione R, se:

$$(P \wedge \neg\beta) \Longrightarrow R$$

allora:

$$\{P\}\ \texttt{while (}\beta\texttt{)}\ S\ \{R\}$$

- Esempio: il seguente algoritmo per calcolare la somma dei valori contenuti in un array di 10 elementi

```
somma = 0;
i = 0;
while (i <= 9)
{
  somma = somma + a[i];
  i = i + 1;
}
```

è corretto perché, avendo come postcondizione

$$R = (somma = \sum_{j=0}^{9} a[j])$$

si può rendere la tripla vera mettendo come precondizione *vero* in quanto

- Il predicato

$$P = (0 \leq i \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j])$$

e la funzione

$$tr(i) = 10 - i$$

soddisfano le ipotesi del teorema dell'invariante di ciclo in quanto

  - L'invarianza

$$\{P \wedge i \leq 9\}\ \texttt{somma = somma + a[i]; i = i + 1;}\ \{P\}$$

segue da

$$\begin{aligned} P_{\texttt{i,i+1}} &= (0 \leq i+1 \leq 10 \wedge somma = \sum_{j=0}^{i+1-1} a[j]) \\ &\equiv (0 \leq i+1 \leq 10 \wedge somma = \sum_{j=0}^{i} a[j]) \end{aligned}$$

e, denotato con P' quest'ultimo predicato, da

$$\begin{aligned} P'_{\texttt{somma,somma+a[i]}} &= (0 \leq i+1 \leq 10 \wedge somma + a[i] = \sum_{j=0}^{i} a[j]) \\ &\equiv (0 \leq i+1 \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j]) \end{aligned}$$

in quanto, denotato con P'' quest'ultimo predicato, si ha

$$\begin{aligned} (P \wedge i \leq 9) &= (0 \leq i \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j] \wedge i \leq 9) \\ &\Longrightarrow P'' \end{aligned}$$

  - Il progresso è garantito dal fatto che $tr(i)$ decresce di un'unità ad ogni iterazione in quanto `i` viene incrementata di un'unità ad ogni iterazione.
  - La limitatezza segue da

$$\begin{aligned} (P \wedge tr(i) \leq 0) &= (0 \leq i \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j] \wedge 10 - i \leq 0) \\ &\equiv (i = 10 \wedge somma = \sum_{j=0}^{9} a[j]) \\ &\Longrightarrow i \not\leq 9 \end{aligned}$$

- Poiché

$$\begin{aligned} (P \wedge i \not\leq 9) &= (0 \leq i \leq 10 \wedge somma = \sum_{j=0}^{i-1} a[j] \wedge i \not\leq 9) \\ &\equiv (i = 10 \wedge somma = \sum_{j=0}^{9} a[j]) \\ &\Longrightarrow R \end{aligned}$$

dal corollario del teorema dell'invariante di ciclo segue che P può essere usato come precondizione dell'intera istruzione di ripetizione.

- Proseguendo infine a ritroso si ottiene prima

$$P_{\texttt{i,0}} = (0 \leq 0 \leq 10 \wedge somma = \sum_{j=0}^{0-1} a[j]) \equiv (somma = 0)$$

e poi, denotato con P''' quest'ultimo predicato, si ha

$$P'''_{\texttt{somma,0}} = (0 = 0) \equiv vero$$

9.4 Verifica della correttezza di algoritmi ricorsivi

- Per verificare la correttezza di un algoritmo ricorsivo conviene ricorrere al principio di induzione, avvalendosi eventualmente anche delle triple di Hoare.
- Esempi relativi agli algoritmi ricorsivi della Sez. 3.3:
  - L'algoritmo ricorsivo per calcolare il fattoriale di $n \geq 1$ è tale che `calcola_fatt_ric(n)` $= n!$ come si dimostra procedendo per induzione su n:
    - Sia $n = 1$. Risulta `calcola_fatt_ric(1)` $= 1 = 1!$, da cui l'asserto è vero per $n = 1$.
    - Sia vero l'asserto per un certo $n \geq 1$. Risulta `calcola_fatt_ric(n + 1)` $=$ `(n + 1) *` `calcola_fatt_ric(n)` $= (n+1) \cdot n!$ per ipotesi induttiva. Poiché $(n+1) \cdot n! = (n+1)!$, l'asserto è vero per $n+1$.
  - L'algoritmo ricorsivo per calcolare l'n-esimo numero di Fibonacci ($n \geq 1$) è tale che `calcola_fib_ric(n)` $= fib_n$ come si dimostra procedendo per induzione su n:
    - Sia $n \in \{1, 2\}$. Risulta `calcola_fib_ric(n)` $= 1 = fib_n$, da cui l'asserto è vero per $n \in \{1, 2\}$.
    - Dato un certo $n > 2$ sia vero l'asserto per ogni $1 \leq m < n$. Risulta `calcola_fib_ric(n)` $=$ `calcola_fib_ric(n - 1)` $+$ `calcola_fib_ric(n - 2)` $= fib_{n-1} + fib_{n-2}$ per ipotesi induttiva. Poiché $fib_{n-1} + fib_{n-2} = fib_n$, l'asserto è vero per n.
  - L'algoritmo ricorsivo per calcolare il massimo e il submassimo di un insieme I_n di $n \geq 2$ elementi è tale che (con abuso di notazione) `calcola_max_submax_ric(`I_n`)` $=$ max_submax(I_n) come si dimostra procedendo per induzione su n:
    - Sia $n = 2$. Risulta `calcola_max_submax_ric(`I_2`)` $=$ max_submax(I_2), da cui l'asserto è vero per $n = 2$, perché usando le triple di Hoare e procedendo a ritroso si ha:

```
{vero}
if (a[sx] >= a[dx])
  {a[sx] = max(I2)}
  ms.max = a[sx];
else
  {a[dx] = max(I2)}
  ms.max = a[dx];
{ms.max = max(I2)}
if (a[sx] >= a[dx])
  {ms.max = max(I2) /\ a[dx] = submax(I2)}
  ms.submax = a[dx];
else
  {ms.max = max(I2) /\ a[sx] = submax(I2)}
  ms.submax = a[sx];
{ms.max = max(I2) /\ ms.submax = submax(I2)}
```

    - Dato un certo $n > 3$ sia vero l'asserto per ogni $2 \leq m < n$. In questo caso l'algoritmo calcola ricorsivamente `calcola_max_submax_ric(`$I'_{n/2}$`)` e `calcola_max_submax_ric(`$I''_{n/2}$`)`, quindi per ipotesi induttiva `ms1` $=$ max_submax($I'_{n/2}$) ed `ms2` $=$ max_submax($I''_{n/2}$), rispettivamente. L'asserto è allora vero per n, perché usando le triple di Hoare e procedendo a ritroso si ha:

```
{vero}
if (ms1.max >= ms2.max)
  {ms1.max = max(In)}
  ms.max = ms1.max;
else
  {ms2.max = max(In)}
  ms.max = ms2.max;
{ms.max = max(In)}
```

```
if (ms1.max >= ms2.max)
  {ms.max = max(In) /\
   ms2.max >= ms1.submax ==> ms2.max = submax(In) /\
   ms2.max < ms1.submax ==> ms1.submax = submax(In)}
  if (ms2.max >= ms1.submax)
    {ms.max = max(In) /\ ms2.max = submax(In)}
    ms.submax = ms2.max;
  else
    {ms.max = max(In) /\ ms1.submax = submax(In)}
    ms.submax = ms1.submax;
else
  {ms.max = max(In) /\
   ms1.max >= ms2.submax ==> ms1.max = submax(In) /\
   ms1.max < ms2.submax ==> ms2.submax = submax(In)}
  if (ms1.max >= ms2.submax)
    {ms.max = max(In) /\ ms1.max = submax(In)}
    ms.submax = ms1.max;
  else
    {ms.max = max(In) /\ ms2.submax = submax(In)}
    ms.submax = ms2.submax;
{ms.max = max(In) /\ ms.submax = submax(In)}
```

■flf_22

Capitolo 10

Attività di laboratorio

10.1 Valutazione sperimentale della complessità degli algoritmi

- La valutazione della complessità degli algoritmi può essere condotta non solo per via analitica, ma anche per via sperimentale.
- La valutazione sperimentale della complessità di un algoritmo richiede di compiere le seguenti attività:
  - Individuare le operazioni dell'algoritmo che sono significative ai fini del calcolo della complessità (p.e. numero di certi confronti, numero di certi assegnamenti, numero di certe chiamate ricorsive). In alternativa, considerare tutti i passi base dell'algoritmo.
  - Introdurre nell'algoritmo delle apposite variabili, con le relative istruzioni di incremento nei punti opportuni, per contare il numero di volte che le operazioni precedentemente individuate vengono effettuate.
  - Eseguire l'algoritmo su più istanze dei dati di ingresso, tracciando un grafico del valore delle variabili contatore in funzione della dimensione dei dati di ingresso.

10.2 Confronto sperimentale degli algoritmi di ordinamento per array

- Scrivere una libreria ANSI C contenente una funzione da esportare per ciascuno dei sei algoritmi di ordinamento per array mostrati nel Cap. 4.
- Scrivere un ulteriore file che include la precedente libreria e contiene la funzione `main` più altre funzioni di utilità, allo scopo di testare il corretto funzionamento delle funzioni della libreria e di confrontare sperimentalmente le rispettive complessità. Le sequenze di valori contenuti in un array prima e dopo l'ordinamento devono entrambe essere stampate a video.
- Creare il `Makefile` per agevolare la compilazione e testare il programma risultante.
- Confrontare sperimentalmente le complessità dei sei algoritmi di ordinamento al variare del numero di elementi dell'array. A tal fine, modificare l'implementazione di ciascun algoritmo in modo tale da contare il numero di passi base oppure il numero di confronti/scambi che esso esegue.
- Si consiglia di sviluppare una funzione di utilità per l'acquisizione da tastiera oppure la generazione automatica e casuale di array aventi un dato numero di elementi.
- Si consiglia inoltre di effettuare più esperimenti su array aventi lo stesso numero di elementi, riportando il valor medio del numero di operazioni calcolato sull'intera batteria di esperimenti, anziché il numero di operazioni relativo al singolo esperimento. ■feg_1&2

10.3 Confronto sperimentale degli algoritmi di ricerca per alberi binari

- Scrivere una libreria ANSI C contenente una funzione da esportare per ciascuno dei seguenti algoritmi di ricerca di un valore all'interno di un albero binario mostrati nel Cap. 6: ricerca in ordine anticipato, ricerca in ordine simmetrico, ricerca in ordine posticipato, ricerca in albero binario di ricerca, ricerca in albero binario di ricerca rosso-nero.
- Scrivere un ulteriore file che include la precedente libreria e contiene la funzione `main` più altre funzioni di utilità, allo scopo di testare il corretto funzionamento delle funzioni della libreria e di confrontare sperimentalmente le rispettive complessità. I valori contenuti nei nodi di un albero binario devono essere stampati a video assieme al valore da cercare e all'esito della ricerca.
- Creare il `Makefile` per agevolare la compilazione e testare il programma risultante.
- Confrontare sperimentalmente le complessità dei cinque algoritmi di ricerca al variare del numero di nodi dell'albero binario. A tal fine, modificare l'implementazione di ciascun algoritmo in modo tale da contare il numero di passi base oppure il numero di attraversamenti di nodi che esso esegue.
- Si consiglia di sviluppare una funzione di utilità per l'acquisizione da tastiera oppure la generazione automatica e casuale di alberi binari (eventualmente di ricerca e rosso-nero) aventi un dato numero di nodi.
- Si consiglia inoltre di effettuare più esperimenti su alberi binari aventi lo stesso numero di nodi, riportando il valor medio del numero di operazioni calcolato sull'intera batteria di esperimenti, anziché il numero di operazioni relativo al singolo esperimento. ■feg_3&4