ALGORITMI E STRUTTURE DATI

Laurea triennale in scienze informatiche

Magliani Andrea
Perego Luca

Università degli studi di Milano-Bicocca

A.A. 2022/2023

INDICE

Problema Computazionale e Algoritmi

1.1 Problema Computazionale

Relazione matematica tra input e output.
Un problema è definito come: $\pi \subseteq input \times output$

1.2 Istanza

Set di input specifici legati ad un determinato problema.

1.3 Algoritmo

Descrizione finita, composta da una sequenza di istruzioni elementari e non ambigue che, se eseguita, trasforma gli input in output.

1.4 Analisi degli Algoritmi

Gli algoritmi vengono **analizzati** per valutarne diversi aspetti:

- Correttezza: verificata con test e dimostrazioni;
- Efficienza: verificata misurando i tempi e lo spazio occupato;

Un algoritmo che risolve un problema per ogni sua istanza in un tempo finito è detto **corretto**.

Un algoritmo che, per almeno una delle istanze, non risolve correttamente il problema è detto **non corretto**.

1.5 Struttura dati

Un modo per memorizzare e manipolare dati.

Correttezza & Efficienza

2.1 Dimostrazione di Correttezza

Invariante di ciclo: metodo per dimostrare la correttezza di un algoritmo contenente un loop. L'invariante di ciclo si divide in 3 fasi:

- **Inizializzazione**: dimostra la correttezza per la prima iterazione;
- **Conservazione**: l'algoritmo è corretto per ogni valore di i e questa incrementa correttamente ad ogni iterazione;
- **Conclusione**: assumendo la condizione del ciclo False, l'algoritmo termina restituendo il risultato corretto;

2.2 Calcolo dell'Efficienza

Un algoritmo efficiente utilizza il minor **tempo** e **risorse** possibili.
È necessario definire una funzione $T(n)$, ovvero il tempo di calcolo impiegato per gestire un input di lunghezza n.

Ad ogni tipo di istruzione viene assegnato un **valore temporale** di esecuzione, per poi contarne le occorrenze nel codice.

$T(n)$ equivale alla somma delle occorrenze di tutti i valori temporali.

In un algoritmo è presente:

Caso Migliore: $T_{migl}(n) \rightarrow$
Sottoinsieme delle istanze in cui l'algoritmo impiega meno.

Caso Peggiore: $T_{pegg}(n) \rightarrow$
Sottoinsieme delle istanze in cui l'algoritmo impiega di più.

Notazioni Asintotiche

3.1 O-Grande

O(n) rappresenta il **limite superiore** [asintoticamente] della funzione $T(n)$ di un algoritmo.

$$O(g(n)) = \{f(n) \mid \exists\, c, n_0 > 0,\ f(n) \leq c \cdot g(n)\ \ \forall n > n_0\} \qquad f(n) \in N \ \wedge\ f(n) > 0\ def.$$

3.2 Ω-Grande

$\Omega(n)$ rappresenta il **limite inferiore** [asintoticamente] della funzione $T(n)$ di un algoritmo.

$$\Omega(g(n)) = \{f(n) \mid \exists\, c, n_0 > 0,\ 0 \leq c \cdot g(n) < f(n)\ \ \forall n > n_0\}$$

3.3 Theta

$\theta(n)$ rappresenta la funzione che **delimita superiormente** ed **inferiormente** la funzione $T(n)$ di un algoritmo.

$$\theta(g(n)) = \{f(n) \mid \exists\, c_1, c_2, n_0 > 0,\ \ 0 \leq c_2 \cdot g(n) \leq f(n)\ \ \leq f(n) \leq c_1 \cdot g(n)\ \forall n > n_0\}$$

3.4 Gerarchie di crescita Asintotica

La crescita di $T(n)$ varia in base alla funzione a cui è associata.
La **scala di crescita** è:

$$c \rightarrow log\, n \rightarrow \sqrt{n} \rightarrow n \rightarrow n\, log\, n \rightarrow n^a [a > 1] \rightarrow a^n \rightarrow n! \rightarrow n^n$$

Caratteristiche degli Algoritmi

4.1 Stabile

Algoritmo che, se nel vettore di input sono presenti due valori **uguali**, mantiene l'ordine tra di loro anche nel vettore ordinato di output.

4.2 In-Place

Algoritmo che non utilizza una **struttura dati ausiliaria**, ma lavora direttamente sull'input.

Algoritmi di Ordinamento

5.1 Definizione

Gli algoritmi di ordinamento sono utilizzati per posizionare gli elementi di un insieme secondo una **relazione d'ordine**.

5.2 Struttura del problema

Ogni algoritmo di ordinamento condivide problema e risultato.

Problema: Ordinamento di un vettore V di n elementi.
Input: Un vettore V di n elementi.
Output: Un vettore V t.c.:

- L'output è una permutazione di V;
- $\forall\, i \in [1, n-1] \quad V'[i] \leq V[i+1]$;

Teorema dell'esperto

6.1 Enunciato

Sia $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ $\quad a \geq 1,\ b > 1,\ f(n)\ asin.\ pos.$

1. se $\exists\, \varepsilon > 0\ t.c.\ f(n) = 0\left(n^{log_b a - \varepsilon}\right)$ allora $T(n) = \theta\left(n^{log_b a}\right)$
2. se $f(n) = \theta\left(n^{log_b a}\right)$ allora $T(n) = \theta\left(n^{log_b a} \cdot log\, n\right)$
3. se $\exists \varepsilon > 0\ t.c.\ f(n) = \Omega\left(n^{log_b a + \varepsilon}\right)$ e se $\exists\, c < 1\ t.c.\ a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ $\forall n > n_0$ allora $T(n) = \theta(f(n))$

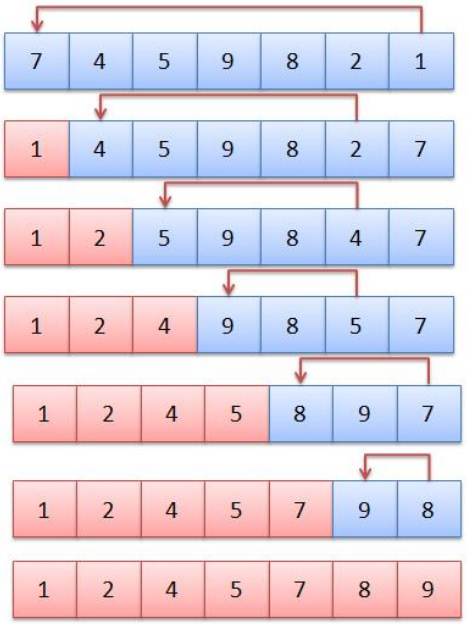
Selection sort

7.1 Pseudocodice

```
SELECTION_SORT (V)

      for i := 1 to V.length - 1

            posmin := i

            for j := i + 1 to V.length - 1

                  if V[posmin] > V[j] then

                        posmin := j

            scambia V[i] con V[posmin]
```

7.2 Funzionamento

Si cerca il numero minore navigando tutto il vettore e si mette nella posizione i, con i che varia dalla prima posizione del vettore fino all'ultima.



7.3 Correttezza

I primi i - 1 elementi di V sono i più piccoli i - 1 elementi di V ordinati in ordine crescente.
INIZIALIZZAZIONE

I primi 0 elementi di V^I sono i più piccoli 0 elementi di V ordinati in ordine crescente

CONSERVAZIONE

Corretto ad inizio e fine ciclo.

TERMINAZIONE

Corretto al termine dell'algoritmo.

7.4 Tempi di calcolo

CASO MIGLIORE e CASO PEGGIORE sono asintoticamente uguali.
$T(n) = \theta(n^2)$

7.5 Caratteristiche

STABILE

No, il selection sort non è un algoritmo di ordinamento stabile in quanto scambiando l'elemento di posizione i con l'elemento più piccolo dell'array, non sempre mantiene l'ordine originale degli elementi uguali nel vettore.

IN PLACE

Si, il selection sort è un algoritmo di ordinamento in place in quanto non utilizza altre strutture dati per ordinare il vettore in input.

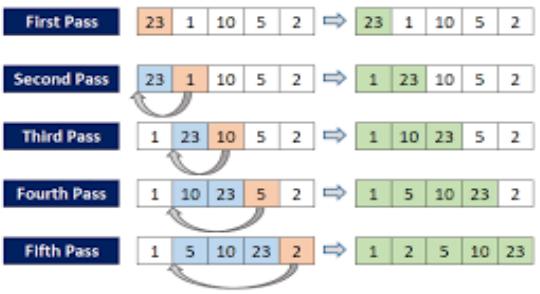
Insertion sort

8.1 Pseudocodice

```
INSERTION_SORT (V)

      for i := 2 to V.length

            j := i - 1
            key := V[i]

            while j >= 1 AND V[j] > Key

                  V[j + 1] := V[j]
                  V[j] := Key
                  j := j - 1
```

8.2 Funzionamento

Si parte dal secondo elemento del vettore in input e si controlla se l'elemento precedente è minore, nel caso si scambiano i due valori, si continua successivamente con l'elemento i + 1 fino alla fine dell'array.



8.3 Correttezza

All'inizio di ogni iterazione i primi i-1 elementi di V^I sono i primi i-1 elementi di V in ordine crescente.

INIZIALIZZAZIONE

Il primo elemento di V^I è il primo elemento di V in ordine crescente

CONSERVAZIONE

Vero ad inizio e fine ciclo

TERMINAZIONE

Vero a fine algoritmo

8.4 Tempi di calcolo

CASO MIGLIORE

Il vettore V è già ordinato. Tmigl(n) = $\theta(n)$ -> T(n) = $\Omega(n)$

CASO PEGGIORE

V è ordinato in senso decrescente. Tpegg(n) = $\theta(n^2)$ -> T(n) = $O(n^2)$

8.5 Caratteristiche

STABILE

Si l'insertion sort è un algoritmo di ordinamento stabile in quanto mantiene l'ordine degli elementi uguali tra di loro.

IN PLACE

Si l'insertion sort è un algoritmo di ordinamento in place in quanto non utilizza strutture d'appoggio per eseguire le sue operazioni.

Mergesort

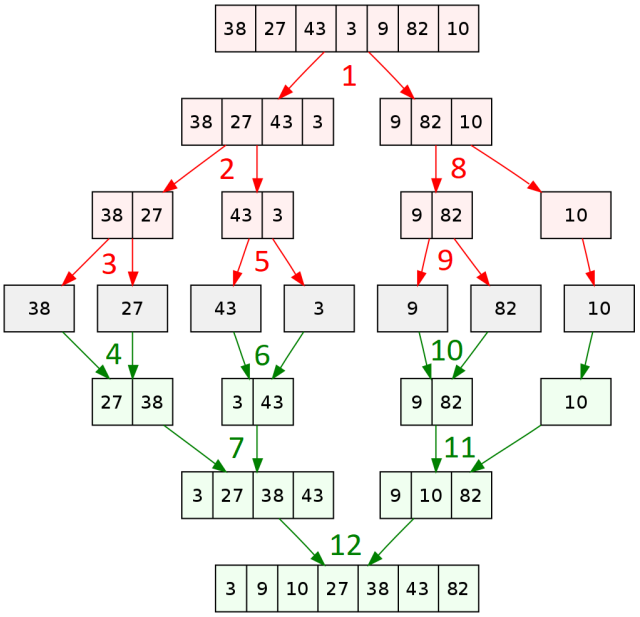
9.1 Pseudocodice

```
MERGESORT (V,l, r)

     if l < r then

          mid := floor[(l+r)/2]
          MERGESORT (V, l, mid)
          MERGESORT (V, mid+1, r)
          MERGE (V,l, mid, r)

MERGE (V, l, mid, r)

     T := Vettore di lunghezza r-l + 1
     i := l
     j := mid + 1
     k := 1

     while i <= mid AND j <= r do

          if V[i] <= V[j] then
               T[k] := V[i]
               i := i + 1

          else

               T[k] := V[j]
               j := j + 1

          k := k + 1

     while i <= min do

          T[k] := V[i]
          i := i + 1
          k := k + 1

     for k := 1 to T.length do

          V[l + k - 1] := T[k]
```

9.2 Funzionamento

Il mergesort utilizza la strategia di programmazione **divide et impera** per ridurre il problema in più sottoproblemi.

Una volta ottenuti i singoletti e arrivati nel caso base si esegue la procedura di merge, andando a unire i singoletti riordinandoli durante il processo.



9.3 Tempi di calcolo

a = 2 (volte che viene chiamato il metodo ricorsivo)
b = 2 (in quante porzioni divido l'array [mid = l+r / 2]
f(n) = $\theta(n)$ (righe di codice che non c'entrano con la ricorsione)

quindi applico il secondo caso del teorema del maestro:

$\theta(n) = \theta(n^{log_2 2}) \rightarrow \theta(n) = \theta(n) \rightarrow T(n) = \theta(n^{log_2 2} \cdot log\, n) \rightarrow T(n) = \theta(n \cdot log\, n)$

Quicksort

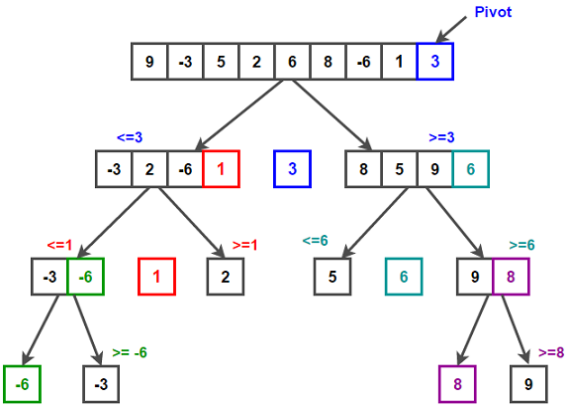
10.1 Pseudocodice

```
QUICKSORT (V, l, r)

      if l < r then

            q := PARTITION (V, l, r)

      QUICKSORT(A, l, q)
      QUICKSORT(A, q+1, r)

PARTITION (V, l, r)

      x := A[l]
      i := l-1
      j := r+1

      while True

            do
                  repeat j := j-1
                  until  A[j] <= x

                  repeat i := i+1
                  until A[i] >= x

            if i < j then
                  scambia (A[j], A[l])

            else
                  return j
```

10.2 Funzionamento

Il quicksort è un algoritmo divide et impera che divide il problema in sottoproblemi in 3 fasi:

- Divide l'array A in 2 sottoarray [p…q] e [q+1…r] in modo che ogni elemento A[p…q] sia minore o uguale di ogni elemento A[q+1…r];
- I due sottoarray sono ordinati dalle chiamate ricorsive;
- I sottoarray sono riordinati in loco, quindi non servono sforzi per ricomporre l'array A ordinato;



10.3 Tempi di calcolo

CASO MIGLIORE

I due sottoarray hanno dimensione n/2.

$$\text{Tmigl(n)} = \theta(n \cdot \log n) \text{ -> } \text{T}(n \cdot \log n) = O(n \cdot \log n)$$

CASO PEGGIORE

Il primo sottoarray ha dimensione n-1 ed il secondo ha dimensione 1.

$$\text{Tpegg(n)} = \theta(n^2) \text{ -> } \text{T}(n^2) = \Omega(n^2)$$

Ricerca Dicotomica

Problema: Ricerca di un elemento in un vettore V ordinato.
Input: un vettore V di n elementi e un intero x.
Output: un intero n t.c. $n \in V \ \wedge \ n = x$

11.1 Pseudocodice

```
Ricerca_Dicotomica(V, x, l, r)

     if r < l then
          return false

     if r = l then
          return x == V[l]

     mid := floor((l+r)/2)

     if x > V[mid] then
          return Ricerca_Dicotomica(V, x, mid + 1, r)
     else
          return Ricerca_Dicotomica(V, x, l, mid)
```

11.2 Tempi di calcolo

Caso migliore e caso peggiore sono **asintoticamente uguali**: $T(n) = \Theta(log_2 n)$.

Problema di selezione

Input: un vettore V di n interi distinti, un intero **i** t.c $1 \leq i \leq n$.
Output: Il valore di V che è maggiore di esattamente i-1 elementi di V.

12.1 Pseudocodice

```
SELEZIONE (V, i, l, r)

     if l = r then
          return V[l]

     cut := RANDOM_PARTITION (V, l, r)
     dim_sx := cut - l + 1

     if i <= dim_sx then
          return SELEZIONE (V, i, l, cut)

     return SELEZIONE (V, i - dim_sx, cut + 1, V)
```

12.2 Tempi di calcolo

Caso peggiore: $\theta(n^2)$

Caso migliore: $\theta(n)$

Si può notare che il caso peggiore è $\theta(n^2)$ quindi asintoticamente maggiore del caso peggiore di un algoritmo di ordinamento come il mergesort, che potremmo usare per riordinare il nostro vettore in input e risolvere facilmente il nostro problema di ricerca, quindi, perchè usare questo algoritmo e non ordinare l'array prima col mergesort?

Utilizzando random partition rendiamo l'algoritmo randomico e non più deterministico ed il suo tempo di calcolo diventa $\theta(n)$ in quanto non consideriamo il caso peggiore essendo un evenienza molto sfortunata e improbabile con l'approccio randomico, per cui l'algoritmo di selezione sviluppato è asintoticamente più veloce rispetto al mergesort, essendo $\theta(n)$ asintoticamente minore di $\theta(n \cdot log\, n)$.

Counting sort

13.1 Pseudocodice

```
COUNTING_SORT (A, k)

      C := vettore di k elementi

      for i := 1 to k
            C[i] := 0

      for j := 1 to A.length
            C[A[j]] := C[A[j] + 1]

      for i := 2 to k
            C[i] := C[i - 1] + C[i]

      B := vettore con n elementi

      for j := A.length down to 1
            B[C[A[j]]] := A[j]
            C[A[j]] := C[A[j]] - 1

      return B
```

13.2 Tempi di calcolo

Primo for: $\theta(k)$
Secondo for: $\theta(n)$
Terzo for: $\theta(k)$
Quarto for: $\theta(n)$

$T(n, k) = \theta(n + k) \quad con\ k = O(n)$
$\rightarrow T(n) = \theta(n)$

Radix sort

14.1 Funzionamento

Ordina l’array iniziando l’ordinamento dalla cifra meno significativa e scalando fino a quella più significativa, è conveniente usarlo su array di dimensioni non troppo elevate, in modo da sfruttare al massimo il suo tempo di esecuzione lineare.

14.2 Pseudocodice

```
RADIXSORT(A)

     for i := 1 to k
          ordina A secondo l’i-esima cifra meno significativa con
          ord.stabile
```

Heap Binario

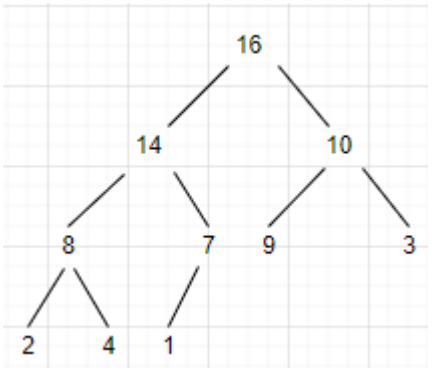
15.1 Definizione

Memorizzato come array + **proprietà**, ma che può essere visto come albero binario quasi completo (completo almeno a sinistra).

Come memorizziamo l'heap:

A = 16 14 10 8 7 9 3 2 4 1

Come vediamo l'heap:



Come calcoliamo la posizione dei nodi nell'array:

parent(i) := floor $(\frac{i}{2})$
left(i) := $2 \cdot i$
right(i) := $(2 \cdot i) + 1$

15.2 Proprietà

length → quanti elementi contiene l'array
heap_size → quanti elementi dell'array sono nell'heap
max heap → un heap che soddisfa: $A[parent(i)] \geq A[i]$
min heap → un heap che soddisfa: $A[parent(i)] \leq A[i]$

Attenzione: Se un array non rispetta almeno una tra le proprietà max-heap e min-heap allora l'array **non è** un heap.

15.3 Nomenclatura

1. altezza di un nodo: numero di archi del cammino più lungo dal nodo stesso ad una sua foglia.
2. altezza heap: altezza dalla radice.

h → quanti nodi ha un heap di altezza h?

1. **Max** = $\sum_{i=0}^{h} i = 2^{h+1} - 1$
2. **Min** = 2^h

3. MAX_HEAP: A[parent(i)] >= A[i]
4. MIN_HEAP: A[parent(i)] <= A[i]

15.4 Max Heapify

Pseudocodice:

```
MAXHEAPIFY (A, i)

      l := left(i)
      r := right(i)
      largest := i

      if l <= A.heap_size AND A[l] > A[largest] then
            largest := l

      if r <= A.heap_size AND A[r] > A[largest] then
            largest := r

      if i ≠ largest then
            scambia A[i] con A[largest]
            MAXHEAPIFY (A, largest)
```

Tempi di calcolo:

In base all'altezza: $T(h) = O(h)$
In base all'heap_size: $T(n) = O(log\, n)$

15.5 Build heap

Pseudocodice:

```
BUILD_HEAP (A)

      A.heap_size := A.length

      for i := floor(A.length/2) down to 1 do
            MAXHEAPIFY (A, i)
```

Tempi di calcolo:

$T(n) \leq \frac{n}{2} O(\log n) = O(n \log n)$ → **stima molto pessimistica**

Dato un heap con n elementi quanti nodi avranno altezza h?

1. Ultimo foglia ha indice n
2. Ultimo nodo ad h = 1 → floor($\frac{n}{2}$)
3. # foglie ? n - floor($\frac{n}{2}$) = ceiling($\frac{n}{2}$) → ceiling($\frac{n}{2^1}$)
4. Ultimo nodo ad altezza 2 → floor($\frac{n}{4}$)
5. # n nodi ad h = 1 → floor($\frac{n}{2}$) - floor($\frac{n}{4}$) → ceiling($\frac{n}{4}$)

→ n nodi ad altezza h = ceiling($\frac{n}{2^{h+1}}$)

$$T(n) \leq \sum_{h=0}^{\log n} ceiling(\frac{n}{2^{h+1}}) \cdot O(h) = O(n \cdot \sum_{h=0}^{\log n} ceiling(\frac{h}{2^{h+1}})) = O(n)$$

15.6 Heap sort

Pseudocodice:

```
HEAPSORT (A)

      BUILDMAXHEAP(A)

      for i := A.length down to 2 do
            scambia A[1] con A[i]
            A.heap_size := A.heap_size - 1
            MAXHEAPIFY(A,1)
```

Tempi di calcolo:

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

Coda con priorità

16.1 Procedure

1. Insert (S, x) $\rightarrow S = S \cup \{x\}$
2. ExtractMax (S) → x (elemento di S con priorità più alta)
3. IncreaseKey (S, x, k) → modifica la priorità dell'elemento x con k

16.2 Extract max

Pseudocodice:

```
HEAPEXTRACTMAX (A)

     if A.heap_size < 1 then
          error "underflow"

     max := A[1]
     A[1] := A[A.heap_size]
     A.heap_size := A.heap_size - 1
     MAXHEAPIFY (A, 1)
     return max
```

Tempi di calcolo:

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

16.3 Increase key

Pseudocodice:

```
HEAPINCREASEKEY (A, i, k)

      if A[i] > k then
            error “Key minore”

      A[i] := k

      while i > 1 AND A[parent(i)] < A[i] do
            scambia A[i] con A[parent(i)]
            i := parent(i)
```

Tempi di calcolo:

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

16.4 Insert

Pseudocodice:

```
HEAPINSERT (A, k)

      if A.length = A.heap_size then
            error “overflow”

      A.heap_size = A.heap_size + 1
      A[heap_size] = -inf
      HEAPINCREASEKEY (A, A.heap_size, k)
```

Tempi di calcolo:

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

Strutture dati dinamiche

17.1 Operazioni

k → valore , x → posizione

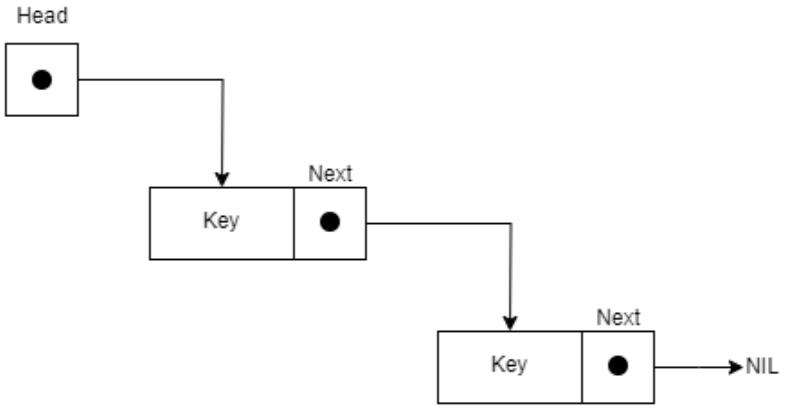
1. Search (S, k)
2. Insert (S, x)
3. Delete (S, x)
4. Maximum (S) / Minimum (S)
5. Predecessor (S, x) / Successor (S, x)

17.2 Implementazione

Metodo	Array	Array ordinato
Search	T(n) = O(n)	T(n) = O(log n)
Insert	T(n) = O(1)	T(n) = O(n)
Delete	T(n) = O(n)	T(n) = O(n)
Max / Min	T(n) = O(n)	T(n) = O(1)
Pred / Succ	T(n) = O(n)	T(n) = O(1)

Liste concatenate

18.1 Struttura



18.2 List search

Pseudocodice:

```
LISTSEARCH (L, k)

      x := L.head
      while x ≠ NIL AND x.key ≠ k do
            x := x.next

      return x
```

Tempi di calcolo:

$T(n) = O(n)$

18.3 List insert

Pseudocodice:

```
LISTINSERT (L, k)

     x.next := L.head
     L.head := x
```

Tempi di calcolo:

$T(n) = O(1)$

18.4 List delete

Pseudocodice:

```
LISTDELETE (L, x)

     if L.head = x then
          L.head := x.next
          x.next := NIL
     else
          y := L.head
          while y.next ≠ x do
               y := y.next

          y.next := x.next
          x.next := NIL
```

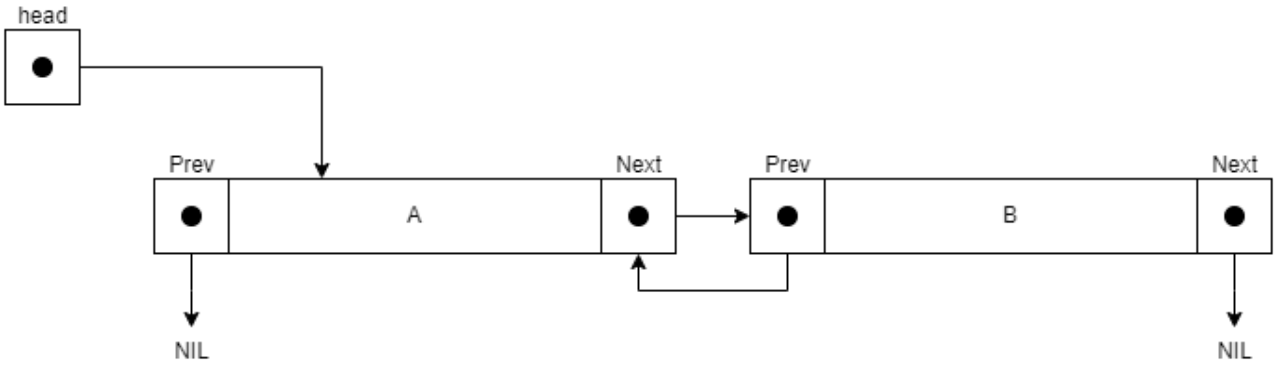
Tempi di calcolo:

Caso migliore: L'elemento da eliminare è la testa della lista
$Tmigl(n) = \theta(1)$

Caso peggiore: L'elemento da eliminare è l'ultimo della lista
$Tpegg(n) = \theta(n)$

Liste doppiamente concatenate

19.1 Struttura



19.2 DL List Insert

Pseudocodice:

```
DlListInsert (L, k)

      x.prev := NIL
      x.next := L.head

      if L.head ≠ NIL then
            L.head.prev := x

      L.head := x
```

Tempi di calcolo:

$T(n) = O(1)$

19.3 DL List Delete

Pseudocodice:

```
DlListDelete (L, k)

      if x.prev ≠ NIL then
            x.prev.next := x.next
      else L.head := x.next

      if x.next ≠ NIL then
            x.prev := x.next := NIL
```

Tempi di calcolo:

$T(n) = O(1)$

Stack / Pile

20.1 Definizione

Gli Stack utilizzano la politica di gestione LIFO (Last in first out)

Procedure:

1. **Push** (insert)
2. **Pop** (delete)
3. **StackEmpty** (ritorna true se la lista è vuota, false altrimenti)
4. **Top** (ritorna l'elemento in cima alla lista senza rimuoverlo)

Attributi:

top → indice che indica dove si dovrà inserire il prossimo elemento

20.2 Tempi di calcolo

	Array	**Lista**
Push	$\theta(1)$	$\theta(1)$
Pop	$\theta(1)$	$\theta(1)$
StackEmpty	$\theta(1)$	$\theta(1)$
Top	$\theta(1)$	$\theta(1)$

20.3 Stack search

Pseudocodice:

```
StackSearch (S, k)

      found := false
      S2 := stack vuoto

      while not StackEmpty(S) AND not found do:

            x := pop (S)
            push (S2, x)

            if x = k then
                  found := TRUE

      while not StackEmpty (S2) do:

            x := pop (S2)
            push (S, x)

      return found
```

Tempi di calcolo:

Caso migliore: L'elemento è in testa
$Tmigl(n) = \theta(1)$

Caso peggiore: L'elemento non è presente
$Tpegg(n) = \theta(n)$

20.4 Stack delete element

Pseudocodice:

```
StackDeleteEl (S, k)

      found := FALSE

      while not StackEmpty(S) AND not found do:
            x := Pop(S)

            if (x = k) then
                  found := TRUE
            else
                  Push (S2, x)

      while not StackEmpty(S2) do:
            Push (S, Pop(S2))
```

Tempi di calcolo:

Caso migliore: k è il primo elemento
$Tmigl(n) = \theta(1)$

Caso peggiore: k non è presente
$Tpegg(n) = \theta(n)$

20.5 Stack sorted insert

Pseudocodice:

```
StackSortedInsert (S, k)

      S2 := stack vuoto

      while not StackEmpty(S2) AND Top(S) < k do:
            x := Pop(S)
            Push (S2, x)

      Push (S, k)

      while not StackEmpty(S2) do:
            Push (S, Pop(S2))
```

Tempi di calcolo:

Caso migliore: k < Top(s)
$Tmigl(n) = \theta(1)$

Caso peggiore: k è il massimo
$Tpegg(n) = \theta(n)$

Queue / Coda

21.1 Definizione

Le Queue utilizzano la politica di gestione FIFO (First in first out)

Procedure:

5. **Enqueue** (insert)
6. **Dequeue** (delete)
7. **QueueEmpty** (ritorna true se la coda è vuota, false altrimenti)
8. **Head** (ritorna l'elemento in testa alla coda senza rimuoverlo)

Attributi:

tail → posizione in cui inserire
head → posizione da cui estrarre

21.2 Tempi di calcolo

	Array	**Lista**
Enqueue	$\theta(1)$	$\theta(1)$
Dequeue	$\theta(1)$	$\theta(1)$
QueueEmpty	$\theta(1)$	$\theta(1)$
Top	$\theta(1)$	$\theta(1)$

21.3 Queue search

Pseudocodice:

```
QueueSearch (Q, k)

      found := FALSE
      Q2 := coda vuota

      while not QueueEmpty(Q) do:

            x := Dequeue(Q)
            Enqueue (Q2, x)

            if x = k then
                  found := TRUE

      while not QueueEmpty(Q2) do:
            Enqueue (Q, Dequeue(Q2))
```

Tempi di calcolo:

Caso migliore e **caso peggiore** coincidono.
$T(n) = \theta(n)$

21.4 Queue sorted insert

Pseudocodice:

```
QueueSortedInsert (Q, k)

      Q2 := coda vuota

      while not QueueEmpty (Q) AND Head(Q) < k do:
            Enqueue (Q2, Dequeue(Q))

      Enqueue (Q2, k)

      while not QueueEmpty(Q) do:
            Enqueue (Q2, Dequeue(Q))

      while not QueueEmpty(Q2) do:
            Enqueue (Q, Dequeue(Q2))
```

Tempi di calcolo:

Caso migliore e **caso peggiore** coincidono (lavorando con le code per riformare la coda iniziale va prima svuotata completamente e poi riempita, per questo caso migliore e peggiore coincidono).
$T(n) = \theta(n)$.

Albero binario di ricerca

22.1 Definizione

è un albero binario tale che per ogni nodo vale la **proprietà degli alberi binari di ricerca** (P.A.B.R).

P.A.B.R :

Sia x un nodo

1. Se y è un discendente del sottoalbero sinistro allora **x.key >= y.key**.
2. Se y è un discendente del sottoalbero destro allora **x.key <= y.key**.

22.2 Inorder visit

Pseudocodice:

```
InorderVisit (x)

     if x ≠ NIL:
          InorderVisit (x.sx)
          print (x.key)
          InorderVisit (x.dx)
```

Tempi di calcolo:

$T(n) = \theta(n)$ → Eseguo un operazione costante (visita del nodo / arco) per n volte

22.3 Preorder visit

Pseudocodice:

```
PreorderVisit (x)

     if x ≠ NIL:
          print (x.key)
          PreorderVisit (x.sx)
          PreorderVisit (x.dx)
```

Tempi di calcolo:

$T(n) = \theta(n)$ → Eseguo un operazione costante (visita del nodo / arco) per n volte

22.4 Postorder visit

Pseudocodice:

```
PostorderVisit (x)

     if x ≠ NIL:
          PostorderVisit (x.sx)
          PostorderVisit (x.dx)
          print (x.key)
```

Tempi di calcolo:

$T(n) = \theta(n)$ → Eseguo un operazione costante (visita del nodo / arco) per n volte

22.5 Bst search

Pseudocodice ricorsivo:

```
BST_Search (x, k)

      if x = NIL then
            return NIL

      if x.key = k then
            return x

      if k < x.key
            return BST_Search (x.sx, k)

      return BST_Search (x.dx, k)
```

Pseudocodice iterativo:

```
BST_Search (x, k)

      x := T.root

      while x ≠ NIL AND x.key ≠ k do
            if k < x.key then
                  x := x.sx
            else
                  x := x.dx

      return x
```

Tempi di calcolo:

Va definito in base all'altezza dell'albero.

Caso migliore: L'elemento che cerco è la key della radice → $T(n) = \Omega(1)$

Caso peggiore : L'elemento che cerco non è presente → $T(n) = O(h)$
con $h = O(n)$, $h = \Omega(log\, n)$

22.6 Bst min / max

Pseudocodice:

```
BST_min (T)
    x := T.root
    while x.sx ≠ NIL do
        x := x.sx
    return x
```

```
BST_max (T)
    x := T.root
    while x.dx ≠ NIL do
        x := x.dx
    return x
```

Tempi di calcolo:

Va definito in base all'altezza dell'albero.

Caso migliore: il min / max è la radice → $T(n) = \Omega(1)$

Caso peggiore : devo scorrere tutto l'albero → $T(n) = O(h)$
con $h = O(n)$, $h = \Omega(log\, n)$

22.7 Bst successor

Pseudocodice:

```
BST_Successor (x)

      if x.dx ≠ NIL then
            return BST_min (x.dx)

      y := x.parent

      while y ≠ NIL AND x = y.dx do
            x := y
            y := y.parent

      return y
```

Tempi di calcolo:

$T(n) = O(h)$ con $h = O(n)$, $h = \Omega(log\, n)$

22.8 Bst insert

Pseudocodice:

```
BST_Insert (T, z)

     x := T.root
     y := NIL

     while x ≠ NIL then
          if z.key < x.key then
               y := x
               x := x.sx
          else
               y := x
               x := x.dx

     z.parent := y

     if y ≠ NIL then
          T.root := z
     else if z.key < y.key then
          y.sx := z
     else
          y.dx := z
```

Tempi di calcolo:

Caso peggiore: $T(n) = O(h)$ con $h = O(n)$, $h = \Omega(log\, n)$

22.9 Bst delete

Pseudocodice:

```
BST_Delete (T, x)

      if x.sx = NIL AND x.dx = NIL then
            if x.parent = NIL then
                  T.root := NIL
            else if x = x.parent.sx then
                  x.parent.sx := NIL
            else
                  x.parent.dx := NIL

      else if x.dx = NIL then
            contrazione_nodo (T, x, x.sx)
      else if x.sx = NIL then
            contrazione_nodo (T, x, x.dx)

      else
            y := BST_min (x.dx)
            scambia x.key con y.key
            BST_delete (T, y)
```

Tempi di calcolo:

Caso peggiore: $T(n) = O(h)$ con $h = O(n)$, $h = \Omega(log\, n)$

22.10 Contrazione nodo

Pseudocodice:

```
contrazione_nodo (T, x, child)
      if x.parent = NIL then
            T.root := child
            child.parent := NIL
      else if x = x.parent.sx then
            x.parent.sx := child
            child.parent := x.parent
      else
            x.parent.dx := child
      child.parent := x.parent
```

Grafi

23.1 Definizione

G = (V,E) → V = insieme finito di elementi (vertici)
E = insieme finito di elementi (archi)

I grafi possono essere:
1. orientati / digrafi / diretti
2. non orientati

Cammino da V a U: $< v0, v1, \dots, vk > t.c\, Vi \in V, v0 = V, vk = U, (Vi, Vi+1) \in E$

Vertici adiacenti: Vertici collegati da un arco

Grado di un vertice: Numero di archi che coinvolgono un nodo:
1. Grado in entrata → archi in entrata
2. Grado in uscita → archi in uscita

23.2 Rappresentazione

1. Liste di adiacenza: array di liste concatenate
    a. Ogni posizione dell'array contiene una lista contenente tutti i vertici adiacenti al vertice osservato
    b. Il numero di nodi presenti in una lista di adiacenza corrisponde al doppio degli archi presente nel grafo non orientato, in caso di grafo orientato il numero di nodi presente nella lista è uguale al numero di archi.

2. Matrice di adiacenza:
    a. Il numero di 1 presenti nella matrice è uguale al doppio degli archi presenti nel grafo non orientato, in caso di grafo orientato il numero di 1 presenti nella matrice è uguale al numero di archi.

23.3 Tempi di calcolo

	Matrice di adiacenza	**Lista di adiacenza**
Spazio occupato	$\theta(\|V\|^2)$	$\theta(\|V\| + \|E\|)$
Ricerca di un elemento	$\theta(1)$	$\theta(grado(U))$
Elencare gli adiacenti	$\theta(\|V\|)$	$\theta(grado(V))$

23.4 Stampa cammino

Pseudocodice:

```
Stampa_cammino (V)

      if v.d := +inf then
            print “no cammino”
      else if v.pred = NIL then
            print v
      else
            Stampa_cammino (v.pred)
```

Tempi di calcolo:

$T(n) = \theta(|V|)$

23.5 Visita in ampiezza (BFS)

Pseudocodice:

```
BFS_Visit (G, s)

      foreach v ∈ G.V
            v.d := +inf
            v.color := BIANCO
            v.pred := NIL

      s.color := GRIGIO
      s.d := 0
      s.pred := NIL
      Q := coda vuota
      Enqueue (Q, s)

      while not Queue_Empty (Q) do
            V := Dequeue (Q)

            foreach u ∈ G.Adj[V]
                  if u.color = BIANCO then
                        u.color := GRIGIO
                        u.d := v.d + 1
                        u.pred := V
                        Enqueue (Q, v)

            v.color := NERO
```

Tempi di calcolo:

$T(n) = \theta(|V| + |E|) \rightarrow$
Il primo foreach viene eseguito tante volte quanti sono i nodi, mentre il while viene eseguito tante volte quanti sono gli archi, per cui il tempo totale è la somma tra la cardinalità dei nodi e degli archi.

23.6 Visita in profondità (DFS)

Pseudocodice:

```
DFS (G)

      foreach v ∈ G.V
            v.color := BIANCO
            v.pred = NIL

      time := 0

      foreach v ∈ G.V
            if v.color = BIANCO then
                  DFS_visit (G, v)


DFS_visit (G,v)

      v.color := GRIGIO
      time := time + 1
      v.d := time

      foreach u ∈ G.Adj[V]
            if u.color = BIANCO then
                  u.pred := v
                  DFS_visit (G, u)

      v.color := NERO
      time := time + 1
      v.f := time
```

Tempi di calcolo:

$T(n) = \theta(|V| + |E|)$ → Lineare nella dimensione dell'input (con implementazione tramite lista di adiacenza)

23.7 Teorema: Struttura di parentesi

$\forall u, v \in V$ **esattamente un caso è vero**:

1. u.d < v.d < v.f < u.f → v è discendente di u
2. v.d < u.d < u.f < v.f → u è discendente di v
3. [u.d,u.f] e [v.d,v.f] sono disgiunti e ne v è discendente di u nè u è discendente di v

23.8 Classificazione archi

1. **TREE EDGE** = (u,v) ∈ E → v è stato scoperto dalla visita di u, v era BIANCO
2. **BACK EDGE** = (u,v) ∈ E → u è discendente di v, v era GRIGIO
3. **FWD EDGE** = (u,v) ∈ E → v è discendente non diretto di u, v era NERO e u.d < v.d < v.f < u.f
4. **CROSS EDGE** = (u,v) ∈ E → se non è uno degli altri tre

Teorema:

G è un DAG se e solo se la DFS non classifica come **back edge** qualche arco di G.

23.9 Ordinamento topologico di un DAG

Pseudocodice:

```
Top_sort (G)

    DFS (G)
    return vertici ordinati in senso decrescente per finishing time
```

Tempo di calcolo:

$T(n) = \theta(|V| + |E|)$ → dovuto al DFS, l'ordinamento richiede tempo $\theta(|V|)$ se implementato correttamente.