Introduzione agli algoritmi

Leonardo Ganzaroli

Indice

Introduzione

Questi appunti sono derivanti principalmente dalle dispense del corso di *Introduzione agli algoritmi* che ho svolto durante la laurea Triennale di informatica all'università "La Sapienza".

1 Nozioni di base

1.1 Alcune definizioni

Definizione L'informatica è la scienza che consente di ordinare, trattare e trasmettere l'informazione attraverso l'elaborazione elettronica.

Una definizione alternativa è *L'informatica è la scienza degli algoritmi che descrivono e trasformano l'informazione: la loro teoria, analisi, progetto, efficienza, realizzazione e applicazione.*

Definizione Una struttura dati è un metodo per organizzare dati che prescinde dai dati stessi.

Definizione Un algoritmo è una sequenza di comandi elementari ed univoci che terminano in un tempo finito ed operano su strutture dati.

Definizione L'efficienza di un algoritmo è la quantificazione delle sue esigenze in termini di spazio e tempo.

Definizione Il problem solving è un'attività atta a raggiungere una soluzione partendo da una situazione iniziale, in questo contesto è limitata ai problemi computazionali.

Definizione Un problema computazionale è un problema che richiede di descrivere in modo automatico la relazione tra un insieme di valori di input e un insieme di valori di output.

Definizione Un algoritmo si dice corretto se per ogni istanza di un certo problema esso termina producendo sempre l'output corretto.

1.2 Modello del calcolatore

Per poter calcolare i vari costi di un algoritmo è necessario usare un modello astratto di calcolatore, si può modellare con 4 elementi:

- Processore
- Memoria centrale
- Memoria secondaria
- Dispositivi I/O

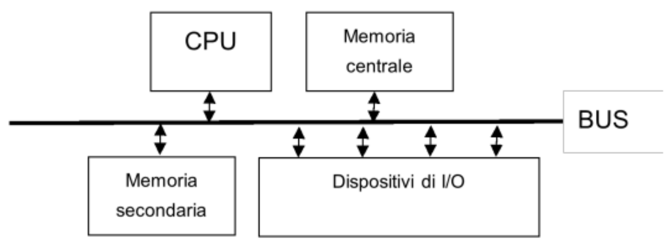


Figura 1: Connessione tra gli elementi

1.2.1 Memoria centrale/primaria

In questo caso ci si concentra sulla RAM (Random access memory) che può essere vista come una lunga sequenza di componenti elementari detti bit che possono assumere solo i valori 0 e 1.

Definizione Un gruppo di 8 bits è detto byte.

Definizione Un registro/parola di memoria è un aggregato di bytes, nei calcolatori moderni sono normalmente 4 o 8.

Inoltre:

- Il processore può operare su un registro (sia in lettura che scrittura) in una sola operazione
- Ogni parola ha un indirizzo
- Il tempo per svolgere un'operazione è lo stesso indipendentemente dall'indirizzo
- Un indirizzo è un numero intero

Definizione Lo spazio di indirizzamento è il numero di bit usati per rappresentare gli indirizzi.

1.2.2 Memoria secondaria

La memoria secondaria ha le seguenti caratteristiche:

- Conserva il contenuto
- È più lenta di quella centrale
- È più grande di quella centrale
- È più economica di quella centrale

1.2.3 Random Access Machine

Questo modello teorico astratto è caratterizzato da una memoria ad accesso casuale, un solo processore ed un insieme di istruzioni eseguite in tempo costante che permettono di fare:

- I/O
- Operazioni aritmetiche
- Accesso e modifica del contenuto della memoria
- Salti

1.3 Criterio della misura

Definizione Il costo computazionale di un algoritmo è il suo tempo di esecuzione e/o le sue necessità in termini di memoria.

Costo uniforme (Quello usato)

Si parla di costo uniforme se si assume che il costo di esecuzione dipende dalla dimensione degli operandi, ogni operazione è un singolo passo con costo 1.

Costo logaritmico

Si parla di costo logaritmico se si assume che il costo delle operazioni elementari è in funzione della dimensione degli operandi ($n \to \log n$).

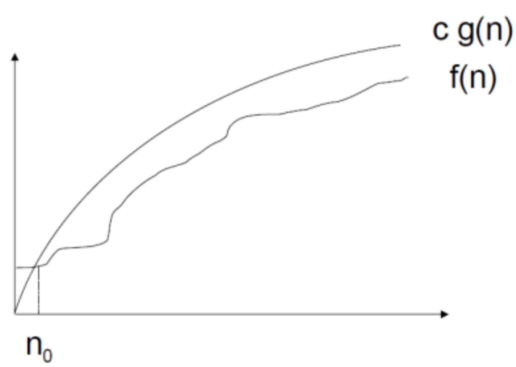
2 Algoritmi

2.1 Notazione asintotica

Definizione Per efficienza asintotica degli algoritmi si intende la valutazione del loro costo quando l'input è sufficientemente grande.

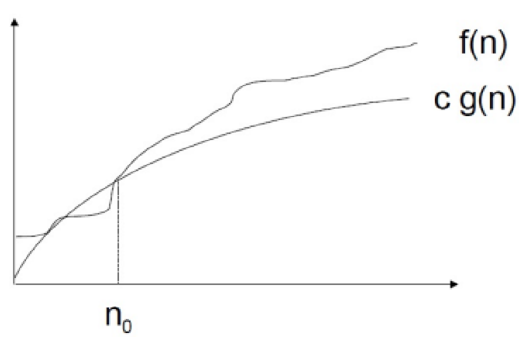
Definizione Date $f(n), g(n) \geq 0$. Si dice che $f(n)$ è un $O(g(n))$ se:

$$\exists c, n_0 \mid \forall\ n \geq n_0 \;\; 0 \leq f(n) \leq c * g(n)$$



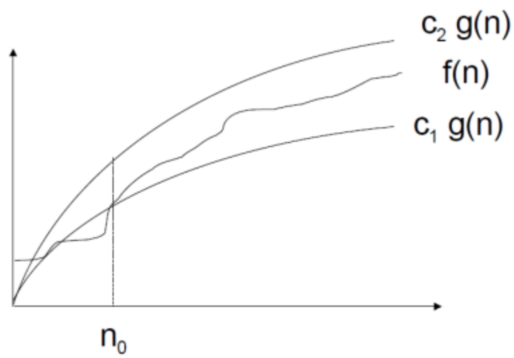
Definizione Date $f(n), g(n) \geq 0$. Si dice che $f(n)$ è un $\Omega(g(n))$ se:

$$\exists c, n_0 \mid \forall\ n \geq n_0 \;\; f(n) \geq c * g(n)$$



Definizione Date $f(n), g(n) \geq 0$. Si dice che $f(n)$ è un $\Theta(g(n))$ se:

$$\exists c_1, c_2, n_0 \mid \forall\ n \geq n_0\ \ c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$



Quindi $f(n)$ deve essere sia $\Omega(g(n))$ che $O(g(n))$.

Esempio:

- $n^2 + 4n$ è $O(n^2)$
  Infatti se $c \geq 5$ si ha che $\forall\ n\ \ n^2 + 4n \leq c * n^2$.
- $2n^2 + 3$ è $\Omega(n^2)$
  Infatti se $c \leq 2$ si ha che $\forall\ n\ \ 2n^2 + 3 \geq c * n^2$.
- $(n + 10)^3$ è $\Theta(n^3)$
  Per Ω basta prendere $n_0 = c = 1$, per O invece $n_0 = 10, c = 8$.

Calcolo alternativo

Un altro modo per determinare la notazione di una funzione è tramite i limiti, infatti si può usare il risultato di $\lim_{n\to\infty} \frac{f(n)}{g(n)}$:

- $> 0 \to f(n) = \Theta(g(n))$
- $= \infty \to f(n) = \Omega(g(n))$
- $= 0 \to f(n) = O(g(n))$

2.1.1 Algebra

Si possono seguire delle semplici regole che permettono di semplificare il calcolo del costo computazionale:

- Costanti moltiplicative

  $\forall\ k > 0, f(n) \geq 0$:

  - Se $f(n)$ è $O(g(n))$ allora $k * f(n)$ è $O(g(n))$
  - Se $f(n)$ è $\Omega(g(n))$ allora $k * f(n)$ è $\Omega(g(n))$
  - Se $f(n)$ è $\Theta(g(n))$ allora $k * f(n)$ è $\Theta(g(n))$

- Commutatività somma

  $\forall\ f(n), d(n) > 0$:

  - Se $f(n)$ è $O(g(n))$ e $d(n)$ è $O(h(n))$ allora $f(n) + d(n) = O(g(n) + h(n)) = O(max(g(n), h(n)))$
  - Se $f(n)$ è $\Omega(g(n))$ e $d(n)$ è $\Omega(h(n))$ allora $f(n) + d(n) = \Omega(g(n) + h(n)) = \Omega(max(g(n), h(n)))$
  - Se $f(n)$ è $\Theta(g(n))$ e $d(n)$ è $\Theta(h(n))$ allora $f(n) + d(n) = \Theta(g(n) + h(n)) = \Theta(max(g(n), h(n)))$

- Commutatività prodotto

  $\forall\ f(n), d(n) > 0$:

  - Se $f(n)$ è $O(g(n))$ e $d(n)$ è $O(h(n))$ allora $f(n)*d(n) = O(g(n)*h(n))$
  - Se $f(n)$ è $\Omega(g(n))$ e $d(n)$ è $\Omega(h(n))$ allora $f(n)*d(n) = \Omega(g(n)*h(n))$
  - Se $f(n)$ è $\Theta(g(n))$ e $d(n)$ è $\Theta(h(n))$ allora $f(n)*d(n) = \Theta(g(n)*h(n))$

Esempio:

$$3n2^n + 4n^4 = \Theta(n)\Theta(2^n) + \Theta(n^4) = \Theta(n2^n) + \Theta(n^4) = \Theta(n2^n)$$

2.2 Valutazione del costo

Definizione Lo pseudocodice è un linguaggio di programmazione informale, esso ha le seguenti caratteristiche:

- Contiene tutti i costrutti di controllo classici
- Usa il linguaggio naturale per specificare le operazioni
- Ignora la gestione degli errori

Alcune regole:

- Le istruzioni elementari hanno costo $\Theta(1)$
- L'istruzione *if* ha costo pari alla somma di:
  - Costo della verifica della condizione
  - Massimo costo dei due rami
- I cicli hanno costo pari alla somma di:
  - Costo della verifica della condizione
  - Somma dei costi massimi di ogni iterazione
- Il costo totale è la somma dei costi di tutte le istruzioni

In alcuni casi non sarà possibile avere un unico risultato preciso, in quel caso si identificano:

- Caso migliore
- caso peggiore
- Caso medio (spesso difficile da calcolare)

Esempio:

Algorithm 1 Trovare l'elemento massimo di un array

```
Trova_max(A):
n=len(A)-1                          ▷ Θ(1)
max=A[0]                            ▷ Θ(1)
for i ∈ [1, n] do                   ▷ (n − 1) iterazioni + Θ(1)
    if A[i]>max then                ▷ Θ(1)
        max=A[i]                    ▷ Θ(1)
    end if
end for
return max                          ▷ Θ(1)
```

Il costo è $T(n) = \Theta(1) + [(n-1) * \Theta(1) + \Theta(1)] + \Theta(1) = \Theta(n)$

2.3 Ricorsione

Definizione Un algoritmo è detto ricorsivo quando si esprime in termini di se stesso.

Definizione Una funzione è detta ricorsiva quando nel suo corpo è presente una chiamata alla funzione stessa.

Definizione Il caso base è quello che permette di terminare la ricorsione, ogni funzione ne deve avere almeno 1.

Il fattoriale è una funzione ricorsiva definita come:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n * (n-1)! & \text{altrimenti} \end{cases}$$

Trasformandola in algoritmo:

Algorithm 2 Fattoriale (ricorsivo)

```
Fatt(x):
if x==0 then                                  ▷ Caso base
    return 1
end if
return x*Fatt(x-1)
```

Nel caso appena visto la ricorsione è definita *diretta*, si dice *indiretta* quando una funzione A chiama una funzione B e a sua volta B chiama A.

Un certo problema che si può risolvere tramite ricorsione è risolvibile anche in modo iterativo e viceversa, ovviamente quale approccio usare dipende dal caso specifico ma conviene sempre scegliere la soluzione che risulta più semplice e chiara. L'unico caso in cui conviene sempre usare l'approccio iterativo è quando bisogna tener conto dell'efficienza, infatti ogni chiamata di funzione occuperà una certa quantità di memoria e questo diventa un problema se le chiamate sono molte.

2.3.1 Equazioni di ricorrenza

Per calcolare il costo delle funzioni ricorsive bisogna riscrivere l'equazione trasformandola in una equazione di ricorrenza che ha la forma:

- Formulazione ricorsiva
- Caso base

La funzione fattoriale vista prima diventa:

$T(n) = T(n-1) + \Theta(1)$ chiamata ricorsiva + costo moltipicazione
$T(0) = \Theta(1)$ caso base

Per trovare il costo effettivo esistono diversi metodi:

- Di sostituzione
- Iterativo
- Dell'albero
- Principale

Metodo di sostituzione

L'idea di base è:

- Ipotizza una soluzione
- Verificala tramite induzione

Provo a trovare il costo della funzione del fattoriale ricorsiva:

- Ipotizzo che:
  - $T(n) = T(n-1) + c$ per qualche $c > 0$
  - $T(0) = d$ per qualche $d > 0$
- Provo con $T(n) = O(n) \rightarrow T(n) \leq kn$

Caso base:

$$T(0) \leq k \iff k \geq d$$

Passo induttivo:

Assumendo che $\forall\ r < n \quad T(r) \leq kr$:

$$(T(n) \leq k(n-1) + c = kn - k + c \leq kn) \iff k \geq c$$

Ovviamente un valore k maggiore sia di c che di d esiste sempre, quindi $T(n)$ è $O(n)$ ed in modo analogo si verifica che $T(n)$ è $\Omega(n)$, quindi $T(n)$ è $\Theta(n)$.

Metodo iterativo

L'idea di base è quella di sviluppare l'equazione ed esprimerla come una somma di termini dipendenti da n e dal caso base.

Usando nuovamente la formula del fattoriale:

$$\begin{aligned} T(n) &= T(n-1) + \Theta(1) \\ &= T(n-2) + \Theta(1) + \Theta(1) \\ &= T(n-3) + \Theta(1) + \Theta(1) + \Theta(1) \end{aligned}$$

Continuando si arriva a $T(n) = n\Theta(1) = \Theta(n)$.

Metodo dell'albero

Si costruisce l'albero di ricorrenza in modo da valutare lo sviluppo del costo graficamente.

Data $2T(\frac{n}{2}) + \Theta(n^2), T(1) = \Theta(1)$:

1. radice: $\Theta(n^2)$
2. $2((\frac{n}{2})^2) = \Theta(\frac{n^2}{2})$
3. $4((\frac{n}{4})^2) = \Theta(\frac{n^2}{4})$

L'i-esimo livello è $2^{i-1}\Theta((\frac{n}{2^{i-1}})^2) = \Theta(\frac{n^2}{2^{i-1}})$, il valore massimo di i deve essere tale che $\frac{n}{2^{i-1}} = 1$, ossia $i - 1 = \log n \rightarrow i = \log n + 1$.

Il totale è $\sum_{i=1}^{\log n+1} \Theta(\frac{n^2}{2^{i-1}}) = n^2 \sum_{j=0}^{\log n} \Theta(\frac{1}{2^j}) = \Theta(n^2)$

Metodo del teorema principale

Questo è il metodo più utile e fornisce la soluzione delle equazioni con forma:

$$T(n) = aT(\frac{n}{b}) + f(n)$$
$$T(1) = \Theta(1)$$

Con $a \geq 1, b > 1$ e $f(n)$ funzione asintoticamente positiva.

Ci sono 3 casi:

- Se $f(n) = O(n^{\log_b a-\epsilon})$ per qualche $\epsilon > 0$ allora $T(n) = \Theta(n^{\log_b a})$
- Se $f(n) = \Theta(n^{\log_b a})$ allora $T(n) = \Theta(n^{\log_b a} \log n)$
- Se $f(n) = \Omega(n^{\log_b a+\epsilon})$ per qualche $\epsilon > 0$, $f(\frac{n}{b}) \leq c * f(n)$ per qualche $c < 1$ ed n abbastanza grande allora $T(n) = \Theta(f(n))$

Esempio:

- $9T(\frac{n}{3}) + \Theta(n)$

  Primo caso, $n^{\log_3 9} = n^2 \rightarrow \Theta(n^2)$.

- $T(\frac{2n}{3}) + \Theta(1)$

  Secondo caso, $n^{\log_{\frac{3}{2}} 1} = 1 \rightarrow \Theta(\log n)$.

- $T(3\frac{n}{4}) + \Theta(n \log n)$

  Si ha $n^{\log_4 3} \approx n^{0,7}$, aggiungendo $\epsilon = 0,2$ si è nel terzo caso. Ponendo $c = \frac{3}{4}$ si ottiene $\frac{3n}{4} \log \frac{n}{4} \leq \frac{3n}{4} \log n$ che risulta vero, quindi $T(n) = \Theta(n \log n)$.

2.4 Alg. di ricerca

Uno dei problemi più diffusi è quello della ricerca di un elemento in un insieme di dati (in questo caso un array), esistono 2 algoritmi diversi che assolvono a questo compito.

2.4.1 Sequenziale

La ricerca sequenziale è quella più semplice e l'unica alternativa se l'array non è ordinato, si controlla ogni elemento presente:

Algorithm 3 Ricerca sequenziale

```
Trova(A,x):
n=len(A)-1                          ▷ Θ(1)
for i ∈ [0, n] do                   ▷ Θ(1) + al più n iterazioni
    if A[i]==x then                 ▷ Θ(1)
        return i                    ▷ Θ(1)
    end if
end for
return -1                           ▷ Θ(1)
```

Costi:

- Caso peggiore $O(n)$
- Caso migliore $O(1)$
- Costo medio:

  Ipotizzando che un elemento x si possa trovare in ogni posizione con la stessa probabilità $(\frac{1}{n})$, il numero medio di iterazioni sarà $\sum_{i=1}^{n} i * \frac{1}{n} = \frac{n+1}{2}$ che diventa $O(n)$.

2.4.2 Binaria

Nel caso in cui l'array sia ordinato si può cercare in modo simile a come si cerca una parola nel dizionario, si controlla l'elemento centrale e in base al suo valore si continua la ricerca nel sottoarray sinistro o destro.

Algorithm 4 Ricerca binaria

```
Trova(A,x):
a,b=0,len(A)
m=(a+b)/2
while A[m]!=x do
    if A[m]>x then
        b=m-1
    else
        a=m+1
    end if
    if a>b then
        return -1
    end if
    m=(a+b)/2
end while
return m
```

In questo caso ad ogni iterazione si vanno a dimezzare gli elementi su cui lavorare, questo porta il numero di iterazioni necessarie ad avere una crescita logaritmica ($\log n$).

Costi:

- Caso peggiore $O(\log n)$
- Caso migliore $O(1)$

Come visto prima si può calcolare il caso medio, ipotizzo che:

- x è presente
- x si può trovare in ogni posizione con la stessa probabilità $(\frac{1}{n})$
- len(Array) = potenza del 2 (per semplicità di calcolo)

Nell'i-esima iterazione sono raggiungibili $n(i) = 2^{i-1}$ elementi, quindi si eseguono i iterazioni solo se x è uno degli elementi raggiunti in quella iterazione e la probabilità che sia in quegli elementi è $\frac{n(i)}{n}$.

Il numero di iterazioni è $\frac{1}{n}\sum_{i=1}^{\log n} i2^{i-1} = \log n - 1 + \frac{1}{n}$ che diventa $O(\log n)$.

2.5 Alg. di ordinamento

Un altro problema molto diffuso è quello dell'ordinamento di un insieme di dati (anche in questo caso si considera un array/vettore) rispetto ad una certa relazione d'ordine sullo stesso.

La maggior parte degli algoritmi che verranno presi in esame sono basati su:

- Scambio tra 2 elementi
- Confronto tra 2 elementi

Definizione I dati satellite sono eventuali dati aggiuntivi collegati ad un elemento.

2.5.1 Semplici

Insertion Sort

Questo algoritmo si basa sul prendere un elemento e spostarlo a sinistra fino a trovargli una posizione adatta:

Algorithm 5 Insertion Sort

```
Ins_sort(A):
for j ∈ [1, len(A) − 1] do
    x=A[j]
    i=j-1
    while (i ≥ 0) & (A[i] > x) do
        A[i+1]=A[i]
        i−−
    end while
    A[i+1]=x
end for
```

Costi:

- Caso migliore = $O(n)$

  Gli elementi sono ordinati ed il secondo ciclo non viene eseguito.

- Caso peggiore = $O(n^2)$

  Gli elementi sono ordinati in senso inverso, in questo caso ogni elemento va spostato lungo tutto l'array e questo porta ogni iterazione ad un costo $n*n$.

Selection Sort

Questo algoritmo si basa sul trovare ad ogni iterazione il minimo/massimo elemento nell'array ancora disordinato e spostarlo in prima/ultima posizione:

Algorithm 6 Selection Sort

```
Sel_sort(A):
for i ∈ [0, len(A) − 2] do
    min=i
    for j ∈ [i + 1, len(A) − 1] do
        if A[j]<A[m] then
            m=j
        end if
    end for
    Scambia A[i] e A[m]
end for
```

Questo algoritmo esegue entrambi i cicli ad ogni iterazione indipendentemente dalla distribuzione dei dati, questo lo porta ad avere il costo unico $\Theta(n^2)$.

Bubble Sort

Questo algoritmo confronta le coppie adiacenti ed eventualmente ne scambia gli elementi finché non sono tutte ordinate:

Algorithm 7 Bubble Sort

```
Bub_sort(A):
for i ∈ [0, len(A) − 2] do
    for j ∈ [i + 1, len(A) − 1] do
        if A[j]<A[i] then
            Scambia A[i] e A[j]
        end if
    end for
end for
```

Come per l'algoritmo precedente anche qui i 2 cicli vengono eseguiti in qualunque caso, il costo è lo stesso.

2.5.2 Efficienti

Definizione L'albero di decisione rappresenta graficamente le possibili strade che un algoritmo basato su ordinamento può percorrere.

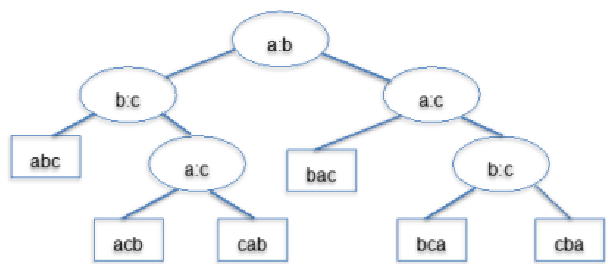


Figura 2: Albero con 3 elementi

Passando ad un livello più generale è vero che:

- Un array lungo n ha $n!$ possibili ordinamenti
- Un albero alto h ha al massimo 2^h foglie

h deve essere tale che $2^h \geq n! \Rightarrow h \geq \log n! = \Theta(n \log n)$, quindi il costo minimo di un algoritmo basato sugli ordinamenti è $\Omega(n \log n)$.

Mergesort

Questo algoritmo è basato sulla tecnica *divide et impera*, ossia divide il problema in sottoproblemi e li risolve ricorsivamente (ogni chiamata riceve metà dell'array ed ordina i sottoarray):

Algorithm 8 Mergesort

```
Mer_sort(A,index,index2):
if index<index2 then
    m=(index+index2)/2
    Mer_sort(A,index,m)
    Mer_sort(A,m+1,index2)
    Fondi(A,index,m,index2)
end if

Fondi(A,index,m,index2):                ▷ Combina i 2 sottoarray in uno
i,j=index,m+1
B=[]
while (i ≤ m) & (j ≤ index2) do
    if A[i]≤A[j] then
        B.append(A[i])
        i++
    else
        B.append(A[j])
        j++
    end if
end while
while i≤m do
    B.append(A[i])
    i++
end while
while j≤index2 do
    B.append(A[j])
    j++
end while
for i ∈ [0, len(B) − 1] do
    A[index+i]=B[i]
end for
```

La funzione Fondi ha costo $\Theta(n)$ dato che tutti i cicli scorrono al più l'intero array, quindi $O(n)+O(n)+O(n)+\Theta(n)$, la funzione principale si può esprimere con l'equazione di ricorrenza:

$$T(n) = 2T(\frac{n}{2}) + \Theta(n)$$
$$T(1) = \Theta(1)$$

Che una volta risolta diventa $\Theta(n \log n)$.

Merge-Insertion

Quando la dimensione dei sottoproblemi diventa abbastanza piccola l'Insertion Sort risulta più veloce del Mergesort, combinandoli si ottiene:

Algorithm 9 Merge_Insertion

```
MerIns_sort(A,index,index2,k,dim):
if dim>k then
    m=(index+index2)/2
    MerIns_sort(A,index,m,k,m-index+1)
    MerIns_sort(A,m+1,index2,k,index2-index)
    Fondi(A,index,m,index2)
else
    Ins_Sort(index,index2)
end if
```

L'equazione è:

$$T(n) = 2T(\frac{n}{2}) + \Theta(n)$$
$$T(k) = \Theta(k^2)$$

Se $k = O(\log n)$ il costo dell'algoritmo diventa $\Theta(n \log n)$.

Quicksort

Anche questo algoritmo sfrutta il *divide et impera*, la differenza sta nell'uso di un *pivot* per dividere in sottoarray:

Algorithm 10 Quicksort

```
Quick_sort(A,index,index2):
if index<index2 then
    m=Partiziona(A,index,index2)
    Quick_sort(A,index,m-1)
    Quick_sort(A,m+1,index2)
end if

Partiziona(A,index,index2):                    ▷ "Crea" i sottoarray
pivot=A[index]                    ▷ Come si sceglie il pivot non è importante
i=index+1
for j ∈ [1 + index,index2 + 1] do
    if A[j]<pivot then
        scambia A[i]e A[j]
        i++
    end if
end for
scambia A[i-1] e A[index]
return i-1
```

Si nota facilmente che Partiziona ha costo $\Theta(n)$, con questa informazione si può scrivere l'equazione di ricorrenza:

$$T(n) = T(k) + T(n-k-1) + \Theta(n) \text{ con } 0 \leq k \leq n-1$$
$$T(1) = \Theta(1)$$

Valutando i 3 possibili casi ottengo i costi:

- Migliore: Sottoproblemi sempre bilanciati

$$2T(\frac{n-1}{2}) + \Theta(n) = \Theta(\log n)$$

- Peggiore: Un sottoproblema è sempre nullo

$$T(n-1) + \Theta(n) = \Theta(n^2)$$

- Medio: Il pivot suddivide gli elementi con egual probabilità

$$\frac{1}{n-1}\left[\sum_{k=0}^{n-1}(T(k) - T(n-k))\right] + \Theta(n) = \Theta(n \log n)$$

Heapsort

Questo algoritmo trasforma l'array in un Max-Heap e sfrutta le sue caratteristiche, ad ogni iterazione scambia la radice con l'ultima foglia e risistema l'heap escludendo ad ogni iterazione l'ultima foglia:

Algorithm 11 Heapsort

```
Heap_sort(A):
Trasforma A in heap                                  ▷ O(n log n)
for x ∈ [len(A) − 1, 1] do
    Scambia A[0] e A[x]
    Heapify(A,0,x)
end for

Heapify(A,i,size):                                   ▷ Aggiusta l'heap
l,r,max= 2i + 1, 2i + 2, i
if (l<size) & (A[l]>A[i]) then
    max = l
end if
if (r≤size) & (A[r]>A[max]) then
    max = r
end if
if max!=i then
    Scambia A[max] e A[i]
    Heapify(A,max,size)
end if
```

Il costo totale è $T(n) = O(n) + O((n-1)\log n) = O(n \log n)$

2.5.3 Lineari

I 2 algoritmi che verranno presi in considerazione hanno un costo lineare perché non sono basati su confronti.

Counting Sort

Ipotizzando che l'array contenga solamente numeri interi compresi in un certo range $[0, k]$, creo un array ausiliario di lunghezza k in cui conto le occorrenze di ogni numero e vado poi a sovrascrivere l'array originale:

Algorithm 12 Counting Sort

```
Count_sort(A):
C=Array di lunghezza max(A)+1
for i ∈ [0, len(A) − 1] do
    C[A[i]]++
end for
j=0
for i ∈ [0, max(A)] do
    while C[i]> 0 do
        A[j]=i
        j++
        C[i]−−
    end while
end for
```

Il costo è dato dalla somma dei due cicli $\Theta(n + k)$, se $k = O(n) \Rightarrow \Theta(n)$.

Una cosa da tenere in considerazione è la dimensione dell'array ausiliario, infatti in casi come $A = \{4, 6, 1, 2, 55555\}$ C occuperà inutilmente una grande quantità di memoria.

Con dati satellite

Dato che l'array originale viene sovrascritto bisogna implementare delle modifiche per preservare eventuali dati satellite:

Algorithm 13 Counting Sort con dati satellite

```
Count_sort2.0(A):
C=Array di lunghezza max(A)+1
B=Array di lunghezza len(A)
for i ∈ [0, len(A) − 1] do
    C[A[i]]++
end for
for i ∈ [1, max(A)] do
    C[i]+=C[i-1]
end for
for i ∈ [len(A), −1] do
    B[C[A[j]]]=A[j]
    C[A[j]]−−
end for
return B
```

Bucket Sort

In questo caso presumo che gli elementi siano equamente distribuiti nell'intervallo $[1, k]$, divido l'intervallo in sottointervalli detti *bucket* di ampiezza uguale che verranno ordinati con un altro algoritmo e ricombinati alla fine:

Algorithm 14 Bucket Sort

```
Buck_sort(A):
Crea i bucket in base a max(A)                    ▷ Bucket=Lista
for i ∈ [0, len(A) − 1] do
    Inserisci A[i] nell'apposito bucket
end for
for i ∈ [0, len(A) − 1] do
    Ordina l'i-esimo bucket
end for
Combina i bucket (seguendo l'ordine B1, B2, . . .) in un'unica lista
Copia la lista in A
```

Il costo dipende da:

- Distribuzione dei numeri nei bucket
- Numero e lunghezza dei bucket
- Algoritmo di ordinamento usato

In ogni caso il costo medio è $O(n + k + \frac{n^2}{k})$ che diventa lineare se $k = n$.

3 Strutture dati

Una struttura dati memorizza e manipola insiemi dinamici di dati varianti nel tempo, ogni elemento (o nodo) può poi essere composto da molteplici dati elementari, comunemente sono composti da:

- Chiave: usata per distinguere gli elementi
- Dati satellite: altri dati non usati direttamente

Le tipiche operazioni che si possono svolgere sono:

- Ricerca di un elemento
- Ricerca del minimo/massimo
- Ricerca dell'elemento precedente/successivo
- Inserimento di un nuovo elemento
- Cancellazione di un elemento

3.1 Array

Un array ha le seguenti caratteristiche:

- Ogni elemento è omogeneo
- Ha l'accesso casuale
- Ha dimensione fissa

Array	Ricerca	Min/Max	Prec/Succ	Inserimento	Cancellazione
generico	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
ordinato	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$

3.2 Lista

Definizione Un puntatore è una variabile che contiene l'indirizzo in memoria di un'altra variabile.

Definizione Una lista è una struttura dati che organizza i suoi elementi in sequenza, le sue proprietà sono:

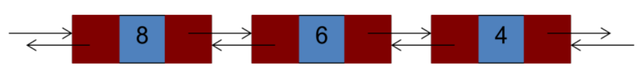
- L'accesso è solo sequenziale
- L'accesso avviene all'inizio o alla fine della lista

3.2.1 Semplice

In questo caso un nodo conterrà un puntatore all'elemento successivo:

3.2.2 Doppia

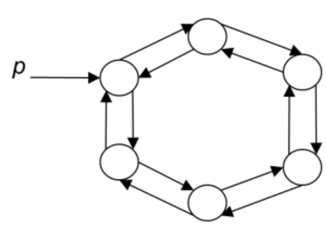
Per diminuire il costo della cancellazione si può inserire nel nodo anche un puntatore all'elemento precedente:



Lista	Ricerca	Min/Max	Prec/Succ	Inserimento	Cancellazione
semplice	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(n)$
doppia	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

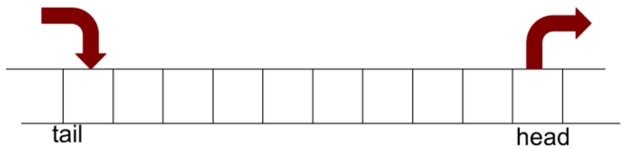
Circolare

Un'implementazione particolare è quella in cui l'ultimo elemento viene fatto puntare al primo creando così un cerchio:



3.3 Coda

Definizione La coda è una struttura con comportamento *FIFO*, ossia gli elementi vengono prelevati (operazione Dequeue) nell'ordine con cui sono stati inseriti (operazione Enqueue), ha una struttura:



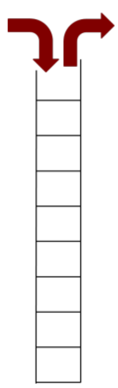
Coda	Enqueue	Dequeue
	$\Theta(1)$	$\Theta(1)$

3.3.1 Con priorità

Una variante è quella in cui la posizione di un elemento non dipende dall'istante di inserimento ma da un altro valore detto di priorità (contenuto nel nodo). Un potenziale problema di questa variante è quello della *starvation* in cui un elemento non verrà mai estratto se viene scavalcato in continuazione da nuovi elementi con più priorità.

3.4 Pila

Definizione La pila è una struttura con comportamento *LIFO*, ossia gli elementi vengono prelevati (operazione Pop) nell'ordine inverso con cui sono stati inseriti (operazione Push), ha una struttura:



Pila	Pop	Push
	$\Theta(1)$	$\Theta(1)$

3.5 Grafo

Definizione Un grafo è una coppia di insiemi (V, E) tali che:

- V è un insieme di nodi
- $E \subseteq V \times V$ è un insieme di archi tra i nodi

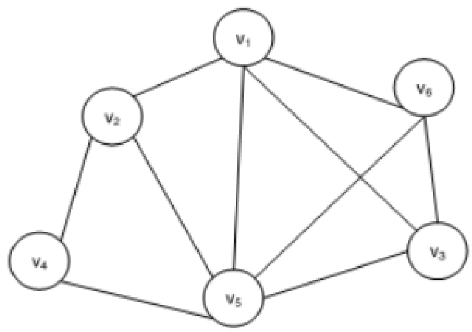


Figura 3: Esempio di Grafo

Definizione Una passeggiata è una sequenza di nodi $(v_1, v_2, \ldots, v_k) \mid \forall\ i \;\; 1 \leq i \leq k-1 \;\; \exists (v_i, v_{i+1}) \in E$.

Definizione Un ciclo è una passeggiata che inizia e finisce sullo stesso nodo ed esso è l'unico ripetuto.

Definizione Un grafo è detto aciclico se non contiene cicli.

Definizione Un cammino è una passeggiata senza archi e vertici ripetuti.

Definizione Un grafo è detto connesso se esiste un cammino tra ogni coppia di nodi.

3.6 Albero

Definizione Un albero è un grafo connesso e aciclico.

Definizione Un albero radicato è un albero in cui è presente un elemento chiamato *radice*, si rappresenta graficamente mettendo la radice in alto e rappresentando i cammini verso il basso organizzandoli a livelli:

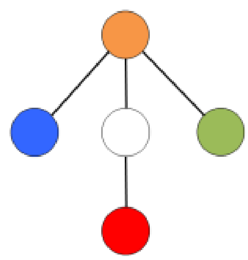


Figura 4: Esempio di albero

Definizione Il padre di un nodo x è il nodo che si incontra prima di lui nel cammino dalla radice, viceversa si dice figlio di x.

Definizione I nodi con lo stesso padre si chiamano fratelli.

Definizione L'antenato di un nodo x è qualsiasi nodo incontrato sul cammino per raggiungere x.

Definizione I discendenti di un nodo x sono tutti i nodi che hanno come antenato x.

Definizione Un nodo senza figli si chiama foglia.

Definizione L'altezza di un albero radicato è la lunghezza del cammino più lungo dalla radice ad una foglia.

Definizione Un albero radicato è detto ordinato se tutti i figli di ogni nodo hanno un qualche ordine.

3.6.1 Binario

Definizione Un albero binario è un albero radicato e ordinato in cui ogni nodo ha al massimo 2 figli definiti sinistro e destro.

Definizione Un albero binario è detto completo se ogni livello ha il massimo numero possibile di nodi.

Definizione Un albero binario è detto quasi completo se tutti i livelli sono pieni ma l'ultimo solo in parte (da sinistra a destra).

Rappresentazione in memoria

Ci sono 3 possibilità per memorizzare un albero binario:

1. **Con record e puntatori**

   Un nodo è formato dal campo chiave e 2 puntatori ai figli.

2. **Posizionale**

   Si usa un array con la radice in posizione 0, i figli di un nodo all'indice i saranno in posizione $2i, 2i - 1$.

3. **Vettore dei padri**

   Si usa un vettore in cui l'indice i corrisponde al nodo i e contiene l'indice del padre di i.

Visita dei nodi

Accedere a tutti i nodi risulta leggermente più complicato delle altre strutture, i possibili modi per visitare tutti i nodi sono 3:

1. **Preorder**

   Prima visito il nodo e poi i sottoalberi.

2. **Inorder**

   Visito il sottoalbero sinistro, il nodo e poi il sottoalbero destro.

3. **Postorder**

   Il nodo è visitato dopo le visite ai sottoalberi.

Nel caso in cui si usino i puntatori l'opzione migliore è usare una funzione ricorsiva, indipendentemente dal tipo di visita il costo è $\Theta(n)$.

3.6.2 Heap

Definizione Un Max-Heap è un albero binario completo (o quasi) con la seguente caratteristica: *ogni chiave di un nodo è più grande delle chiavi dei suoi discendenti*, ovviamente esiste anche il Min-Heap con la caratteristica opposta.

Data la loro struttura risulta evidente che trovare il massimo/minimo ha costo $\Theta(1)$ essendo esso la radice.

3.6.3 ABR

Definizione Un albero binario di ricerca è un albero binario con la seguente caratteristica:

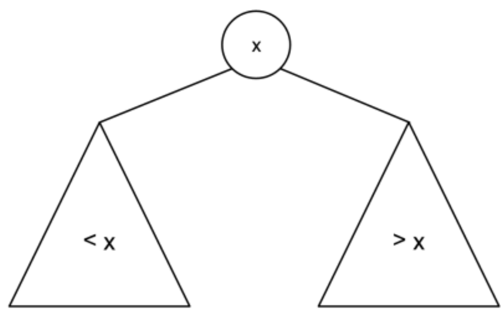


Figura 5: Struttura ABR

Il costo delle operazioni dipende dal bilanciamento dell'albero:

- Caso peggiore:

  Se l'albero è degenere (graficamente si immagini una diagonale) potrebbe essere necessario scorrerlo interamente, quindi $O(n)$.

- Caso migliore:

  Se l'albero è completo diventa simile ad una ricerca binaria, il costo è $\Omega(\log n)$.

3.6.4 Rosso-nero

Definizione Una foglia fittizia è una foglia senza valore che viene eventualmente aggiunta ad un nodo per fargli avere 2 figli.

Definizione Un albero-RB è un ABR con le seguenti caratteristiche:

- I nodi hanno un campo aggiuntivo che contiene il loro colore (Rosso o nero)
- Un nodo rosso ha entrambi i figli neri
- Ogni foglia fittizia è nera
- La radice è nera
- Ogni cammino da un nodo ad una sua foglia discendente contiene lo stesso numero di nodi neri, il numero di nodi neri si indica con *b-altezza*

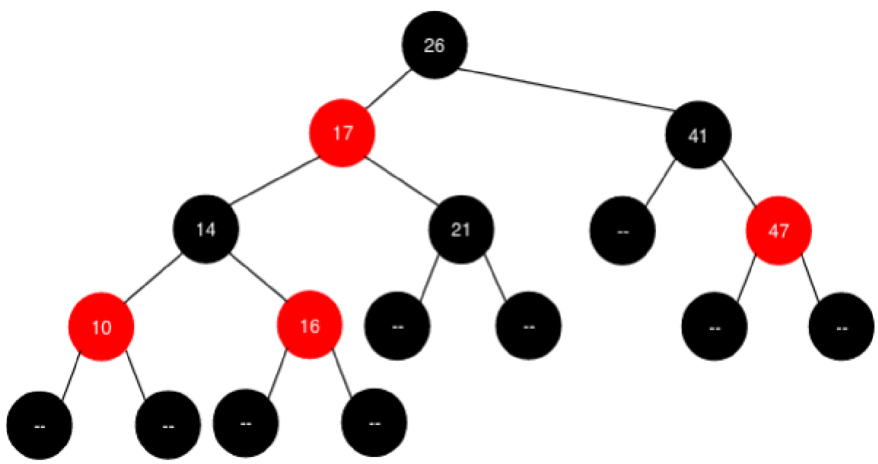


Figura 6: Esempio di albero RB

Queste caratteristiche permettono di avere un albero la cui altezza è sempre:

$$h \leq 2\log(n+1) \text{ con } n \text{ nodi interni}$$

Rotazione

Questo albero ha una particolare operazione detta *rotazione* (a DX o SX) che gli permette di mantenere le caratteristiche dopo un inserimento o cancellazione in $O(\log n)$.

Nello specifico la rotazione a SX di un nodo x consiste in:

1. Il sottoalbero sinistro del figlio destro di x diventa il sottoalbero destro di x
2. x diventa il figlio sinistro del suo figlio destro

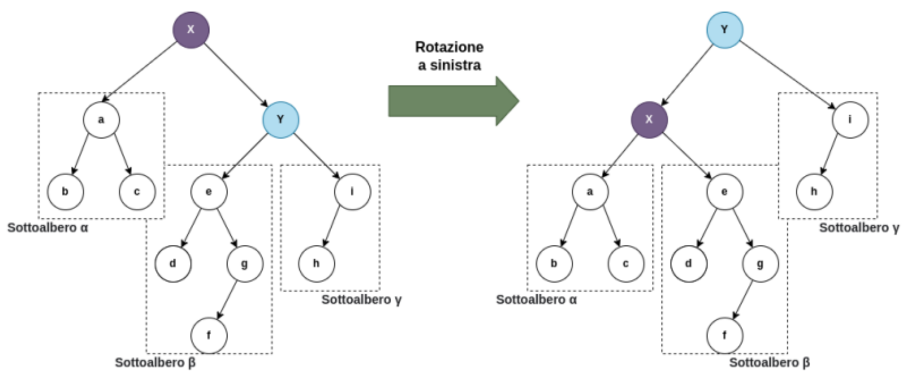


Figura 7: Esempio di rotazione a SX

3.7 Dizionario

Definizione Un dizionario è una struttura dati che permette di gestire un insieme dinamico di dati (normalmente ordinato) con 3 sole operazioni:

- Inserisci
- Cancella
- Cerca

Da qui in poi:

- U = insieme dei valori delle chiavi
- n = numero di elementi da memorizzare
- m = numero di posizioni disponibili

3.7.1 Indirizzamento diretto

Ipotizzando $n \leq |U| = m$ basta un array con m posizioni che permette di avere le operazioni con costo $\Theta(1)$.

Nella realtà però non è un metodo utilizzabile perché:

1. U potrebbe essere enorme
2. Le chiavi effettivamente usate potrebbe essere poche e ciò porta ad uno spreco di memoria

3.7.2 Hash

Per risolvere il problema del metodo precedente viene fatto uso di una funzione detta *hash* che fornisce la posizione dove inserire l'elemento in base alla sua chiave.

Le 3 funzioni più comuni sono:

- Scansione lineare:

$$h(k, i) = (h'(k) + i) \mod m \text{ con } i \in [0, m-1]$$

- Scansione quadratica:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m \text{ con } i \in [0, m-1]$$

- Hashing doppio:

$$h(k, i) = (h_1(k) + h_2(k)i) \mod m \text{ con } i \in [0, m-1]$$

Anche questo metodo ha un problema, bisogna trovare una funzione per cui un'eventuale collisione (la funzione dà una posizione già occupata) avvenga con la probabilità più bassa possibile.

Risoluzione delle collisioni

Ci sono 2 metodi per affrontare il problema:

1. Liste di trabocco

   Essenzialmente viene associata una lista ad ogni possibile output della funzione, mediamente il costo di ricerca/cancellazione diventa $\Theta(1 + \frac{n}{m})$.

2. Indirizzamento aperto

   Nella funzione si tiene conto anche del numero di collisioni incontrate $h(k, 0), h(k, 1), \ldots$, bisogna però gestire la cancellazione che lasciando una casella vuota può portare a risultati errati nella ricerca.