# Chapter 1

# Introduzione

## 1.1 Notazione O-Ω-Θ (pag 4)

Per semplificare l'analisi asintotica degli algoritmi sono state introdotte le seguenti notazioni:

$$O(f) = \{g | \exists a > 0 : \exists b > 0 : \forall N \in N : g(N) \leq af(N) + b\}$$
$$\Omega(f) = \{g | \exists c > 0 : \exists \text{ infinite n } : g(n) \geq cf(n)\}$$

In entrambi i casi si tratta di classi di funzioni. Quando $f \in O(g)$ e $f \in \Omega(g)$ al contempo si dice che $f \in \Theta(g)$. Esistono ulteriori definizioni alternative, ad esempio quelle utilizzate nel Blatt 1. Il seguente teorema può essere utile (dimostrazione nel Blatt 1):

**Theorem 1**

Date due funzioni $f, g : N \to R^+$. Se $\lim_{x \to +\infty} \frac{f(n)}{g(n)}$ converge ad una costante $C \geq 0$ allora $f \in O(g)$.

■

## 1.2 Algoritmo di Karatsuba (pag 12)

Un esempio di algoritmo più efficiente per multiplicare due numeri è il seguente:

$65 * 28 = (2 * 6) * 100 + (2 * 6) * 10 + (5 * 8) * 10 + 5 * 8 + (6 - 5) * (8 - 2) * 10 = 1820$

In questo modo abbiamo solo 3 multiplicazioni elementari anziché 4. L'algoritmo può essere generalizzato grazie a divide and conquer dividendo ogni numero in due ed applicando l'algoritmo di base. Analizzando il tempo in base al numero delle multimplicazioni otteniamo che impieda circa $O(n^{1,58})$.

## 1.3 Maximum subarray (pag 20)

Dato un array di numeri il problema consiste nella ricerca del subarray con la somma degli elementi maggiore. Se essa è negativa il risultato è 0. Il metodo più efficiente per ricavare il risultato è il seguente:

```
//A=array da 1, ... n
max=0
scan=0
for (i=1; i<=n; i++){
        scan+=A[i]
        if scan < 0
                scan=0
        if scan > max
                max=scan
}
```

In questo modo il problema viene risolto in tempo lineare.

# Chapter 2

# Sort

Per semplicità ammettiamo che si debba sempre ordinare un array (chiamato $a$) contenente $n$ numeri (interi). In ogni caso con questi algoritmi è possibile ordinare qualsiasi oggetto appartenente ad un universo in cui vige un ordine totale.

## 2.1 Sortieren durch Auswahl (pag 82)

Selection sort consiste nel cercare ogni volta il minimo tra la posizione $i$ e $n$. Una volta trovato esso viene scambiato con l'$i-$tesimo numero.

### Esempio

| | | | | | |
|---|---|---|---|---|---|
| | 15 | 2 | 43 | 17 | 4 |
| $\Rightarrow$ | 2 | 15 | 43 | 17 | 4 |
| $\Rightarrow$ | 2 | 4 | 43 | 17 | 15 |
| $\Rightarrow$ | 2 | 4 | 15 | 17 | 43 |

### Implementazione

```
int i,j, min, temp
for i=1:n-1
       min=i
       for j=(i+1):n
              if a[j]$<$ a[min]
                     min=j
       temp=a[min]
       a[min]=a[i]
       a[i]=temp
```

### Analisi

Dal doppio loop si vede semplicemente che l'algoritmo impiega $\Theta(n^2)$ comparazioni e nel peggior caso (i numeri sono ordinati dal più grande al più piccolo)

O(n) scambi. Da notare che per trovare il minimo sono necessari almeno $n-1$ confronti (Satz 2.1), quindi l'algoritmo non può andare più veloce di così.

## 2.2 Sortieren durch Einfügen (pag 85)

In inglese si chiama insertion sort. Per induzione i numeri fino a $i-1$ sono già ordinati. Il principio consiste di piazzare l'$i-$tesimo elemento al giusto posto, se necessario scalando i restanti a destra di una posizione. Esempio:

| | | | | | | |
|---|---|---|---|---|---|---|
| | 2 | 15 | / | 43 | 17 | 4 |
| $\Rightarrow$ | 2 | 15 | 43 | / | 17 | 4 |
| $\Rightarrow$ | 2 | 15 | 17 | 43 | / | 4 |
| $\Rightarrow$ | 2 | 4 | 15 | 17 | 43 | / |

### Implementazione

```
int i,j,t
for i=2:n
        j=i
        t=a[i]
        while a[j-1] > t
                //scala a destra di una posizione
                a[j]=a[j-1]
                j=j-1
        a[j]=t
```

Si nota subito che se implementato così l'algoritmo può non fermarsi se $t$ è il più piccolo numero. Serve quindi un elemento di stop, ad esempio inserendo $a[0] = t$ prima del while loop.

### Analisi

Nel peggior caso $\Theta(n^2)$ comparazioni ed altrettanti spostamenti. Nel miglior caso $O(n^2)$ comparazioni e spostamenti. Il caso medio rispecchia il peggiore.

## 2.3 Bubblesort (pag 89)

Il principio di questo algoritmo è semplicissimo: ad ogni iterazione viene scambiato l'elemento $a[i]$ con $a[i+1]$ (chiaramente solo se maggiore). In questo modo l'elemento più grande si sposta verso destra. Esempio:

| | | | | | |
|---|---|---|---|---|---|
| | 15 | 2 | 43 | 17 | 4 |
| $\Rightarrow$ | 2 | 15 | 43 | 17 | 4 |
| $\Rightarrow$ | 2 | 15 | 17 | 43 | 4 |
| $\Rightarrow$ | 2 | 15 | 17 | 4 | 43 |
| $\Rightarrow$ | 2 | 15 | 4 | 17 | 43 |
| $\Rightarrow$ | 2 | 4 | 15 | 17 | 43 |

**Implementazione**

```
do
        flag=true
        for i=1:n-1
        if a[i] > a[i+1]
                swap(a[i],a[i+1])
                flag=false
while (!flag)
```

**Analisi**

Nel miglior caso, se l'array è già ordinato, abbiamo $n-1$ paragoni e nessuno scambio. Nel caso medio e peggiore l'algoritmo necessita di $\Theta(n^2)$ scambi e paragoni.

## 2.4 Quicksort (pag 92)

Il principio di quicksort è quello di scegliere un elemento come pivot (nel libro è sempre l'ultimo, ma ne basta uno casuale) e dividere l'array in due. A sinistra del pivot si trovano tutti gli elementi minori, mentre a destra quelli maggiori. Dopodiché viene richiamato l'algoritmo in modo ricorsivo sui due blocchi. L'algoritmo può essere eseguito "in situ", vale a dire che non serve spazio extra per salvare i dati (chiaramente un numero costante di variabili temporanee è permesso). Si usa quindi un solo array e si aggiungono due argomenti al metodo, ad esempio $l$ ed $r$, ad indicare l'intervallo in cui eseguire quicksort. Per dividere i numeri rispettivamente a destra ed a sinistra del pivot si può semplicemente far scorrere verso il centro due puntatori (chiamati $i$ e $j$) che partono rispettivamente alla posizione $l$ e $r$. Quando $a[i] \geq pivot$ e $a[j] \leq pivot$, $a[i]$ ed $a[j]$ vengono scambiati. Alla fine, quando $i = j$, l'elemento $a[i]$ viene scambiato col pivot e viene richiamato l'algoritmo da $l$ ad $i-1$ e da $i+1$ a $r$ (il pivot si trova già alla posizione giusta). Di seguito un esempio:

**Esempio**

array: 5 7 3 1 6 4
quicksort(array, 1, 6)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 5 | 7 | 3 | 1 | 6 | (4) | ← | pivot |
| | ↑ | | | ↑ | | | | |
| | $i$ | | | $j$ | | | | |
| ⇒ | 1 | 7 | 3 | 5 | 6 | (4) | | |
| | | ↑ | ↑ | | | | | |
| | | $i$ | $j$ | | | | | |
| ⇒ | 1 | 3 | 7 | 5 | 6 | (4) | | |
| | | | ↑ | | | | | |
| | | | $i, j$ | | | | | |
| ⇒ | 1 | 3 | (4) | 5 | 6 | 7 | | |

quicksort(array, 1, 2), quicksort(array, 3, 6)

## Implementazione

```
quicksort(int[ ] a, int l, int r)
       if r>l
              i=l-1
              j=r
              v=a[r] //pivot
              while (true)
                     do
                            i+=1
                     until a[i]>=v
                     do
                            j-=1
                     until  a[j]<=v
                     if i>=j
                            break
                     swap(i, j) //scambia a[i] con a[j]
              swap(i,r)
              quicksort(a, l, i-1)
              quicksort(a, i+1, r)
```

## Analisi

Nel peggior caso, ad esempio se la sequenza è già ordinata o più in generale se il pivot è sempre l'elemento maggiore o minore, quicksort impiega $O(n^2)$ comparazioni, poiché ad ogni ricorsione c'è un solo elemento in meno. In questo caso avverrebbero $O(n)$ spostamenti. Nel miglior caso l'algoritmo impiega un tempo di $\Theta(n \log(n))$, poiché l'albero delle ricorsioni ha un'altezza logaritmica. Lo stesso ragionamento vale per il numero di spostamenti. A pagina 99 c'è una lunga dimostrazione per induzione che mostra che anche il caso medio ha lo stesso tempo del miglior caso. Nonostante l'algoritmo sia in situ, anche le ricorsioni occupano un determinato spazio nella memoria, vale a dire $\Omega(n)$ chiamate. Rendendo l'algoritmo semi-iterativo si possono ottenere solo $O(\log(n))$ chiamate. Si deve semplicemente richiamare in modo ricorsivo quicksort sul più piccolo sub-array, mentre si completa l'altra parte in modo iterativo (algoritmo illustrato a pagina 100). Alla pagina successiva è invece mostrato come fare a rendere quicksort totalmente iterativo (personalmente non ho capito molto).

## Varianti di quicksort

Per evitare un tempo di $O(n^2)$ nel caso di un array già ordinato sono state pensate alcune varianti dell'algoritmo, le quali concernono solamente la scelta del pivot. La prima è la cosiddetta 3-Median-Strategie (median of three values strategy), che consiste semplicemente nel prendere tre elementi campione, rispettivamente a destra, sinistra e nel mezzo, e scegliere come pivot il valore medio tra questi. L'altra strategia si chiama Zufalls-Strategie (randomized quicksort) e consiste nel scegliere il pivot casualmente. Chiaramente può ancora avvenire un caso in cui necessita di un tempo quadratico, ma non esiste più una successione di numeri per cui questo accade. Una variante per rendere l'algoritmo glatt (smooth), cioè che impiega in media $O(n \log(n))$ comparazioni

e solo $O(n)$ quando l'input consiste in n elementi uguali, è la seguente. Praticamente gli elementi che sono uguali al pivot vengono spostati all'estrema destra o sinistra, dopodiché i due blocchi vengono riportati al centro e l'algoritmo prosegue come di consueto. L'implementazione ed una spiegazione più dettagliata possono essere trovate a pagina 104.

## 2.5 Heapsort (pag 106)

Heapsort si basa su insertion sort, ma invece di cercare il minimo/massimo ogni volta, che costa $O(n)$ paragoni, ci si affida ad un Heap (un tipo di albero binario). L'algoritmo è semplice ma efficace: finché l'albero non è vuoto si allontana la radice, nonché l'elemento massimo, e lo si scambia con l'ultimo elemento. Dopodiché tramite un indice si riduce l'albero di un elemento e si restaurano (tramite un metodo chiamato versickern) le sue proprietà (richiede un tempo di $O(\log(n))$). Rimane solo la parte iniziale, cioè la costruzione di un Heap a partire da un array qualsiaisi. Un metodo efficiente consiste nel eseguire il metodo "versickern" sulla sequenza delle chiavi $k_{\lfloor \frac{n}{2} \rfloor}, k_{\lfloor \frac{n}{2} \rfloor - 1}, ..., k_1$. Esempio:

$$
\begin{array}{cccccccc}
 & 2 & 1 & 5 & 3 & 4 & 8 & 7 & 6 \\
\Rightarrow & 2 & 1 & 5 & 6 & 4 & 8 & 7 & 3 \\
\Rightarrow & 2 & 1 & 8 & 6 & 4 & 5 & 7 & 3 \\
\Rightarrow & 2 & 6 & 8 & 3 & 4 & 5 & 7 & 1 \\
\Rightarrow & 8 & 6 & 7 & 3 & 4 & 5 & 2 & 1
\end{array}
$$

### Implementazione

```
//costruiamo l'heap
for (i=n/2; i>=1; i--)
       versickere(a,i,n) //array a da i a N
//heapsort
for (i=n      ; i>=2; i--)
       swap(a[1],a[i])
       versickere(a,1,i-1)
```

Dove versickere è:

```
versickere(int[] a, int i, int m)
       int j
       while 2*i<=m //a[i] ha un figlio a sinistra
              j=2*i
              if j<m //a[i] ha anche un figlio a destra
                     if a[j]<a[j+1]
                            j=j+1
                     //ora a[j] e' il figlio con un valore maggiore
                     if a[i]<a[j]
                            swap(a[i],a[j])
                            i=j //va avanti a restaurare
                     else
                            i=m //fine
```

### Analisi

La costruzione dell'heap avviene in tempo lineare. Il resto dell'algoritmo prevede $n$ volte il metodo versickere, quindi in complesso l'algoritmo richiede $O(n \log(n))$ comparazioni e spostamenti nel peggior caso. Da notare che non occupa spazio extra, quindi è completamente "in situ". L'unico problema è che non è stabile, vale a dire che la posizione relativa di chiavi con lo stesso valore inizialmente adiacenti può cambiare.

## 2.6 Mergesort (pag 112)

Anche mergesort si basa su divide and conquer. Si basa sul fatto che due blocchi dell'array sono già ordinati, quindi possono semplicemente essere uniti in tempo lineare, lasciando scorrere due indici nelle rispettive parti e salvando i valori in ordine crescente in un array temporaneo.

### Esempio

$A = \{2, 1, 3, 9, 5, 4\}$. Esso viene dunque diviso in due, vale a dire in $A_1 = \{2, 1, 3\}$ e $A_2 = \{9, 5, 4\}$. Per ricorsione essi vengono ordinati, quindi rimane da unire $A_1 = \{1, 2, 3\}$ e $A_2 = \{4, 5, 9\}$. Il passo merge prevede l'unione dei due insiemi, quindi si ottiene il risultato finale.

### Implementazione

```
mergesort (int[] a, int l, int r)
        if l<r //altrimenti caso base: array rimane invariato
                m=(l+r)/2 //posizione media
                mergesort (a, l, m)
                mergesort (a, m+1, r)
                merge (a, l, m, r)

merge (int[] a, int l,m,r)
        i=l
        j=m+1
        k=l
        while (i<=m && j<=r)
                if a[i]<a[j]
                        b[k]=a[i]
                        i++
                else
                        b[k]=a[j]
                        j++
                k++
        if i>m //prima successione esautita
                for h=j:r
                        b[k+h-j]=a[h]
        else
                for h=i:m
                        b[k+h-i]=a[h]
        for h=l:r
```

```
a[h]=b[h] //salva nuovamente nell'array originale
```

## Analisi

Merge impiega sempre $O(n)$ spostamenti e comparazioni. Dato che l'array viene sempre diviso in due, il metodo merge viene chiamato $\log(n)$ volte. In totale l'algoritmo impiega sempre $\Theta(n \log(n))$. L'unico problema è che serve sempre $O(n)$ spazio extra per salvare l'array di supporto.

## Mergesort iterativo

L'algoritmo può anche essere reso totalmente iterativo. Esso si chiama "reines 2-Wege-Mergesort" (straight 2-way merge sort). L'idea è quella di unire tramite il metodo merge intervalli sempre più grandi (esattamente il doppio) fino ad avere l'intero array completamente ordinato. Esempio:

$$\begin{array}{rcccccc} & 2 & 1 & 3 & 9 & 6 & 5 \\ \Rightarrow & 1 & 2| & 3 & 9| & 5 & 6| \\ \Rightarrow & 1 & 2 & 3 & 9| & 5 & 6 \\ \Rightarrow & 1 & 2 & 3 & 5 & 6 & 9| \end{array}$$

```
straightmergesort(int[] a, int l,r)
       int size, ll, mm, rr
       size=1
       while size<r-l+1
              rr=l-1 //gli elementi fino ad a[rr] incluso sono a posto
              while rr+size<r //fino che tutti non sono elaborati
                     ll=rr+1 //bordo sinistro successione 1
                     mm=ll+size-1 //bordo desto successione 1
                     if mm+size<=r
                            rr=mm+size
                     else
                            rr=r
                     merge (a, ll, mm, rr)
              size=2*size
```

Le performance sono identiche a prima, ma il metodo è completamente iterativo.

## Mergesort naturale

L'unica differenza da straight 2-way merge sort è che al posto di unire successioni di lunghezza arbitraria sfruttiamo successioni già ordinate alla partenza. Ad esempio:

$$\begin{array}{rcccccc} & 2 & 1 & 3 & 6 & 9 & 5 \\ \Rightarrow & 2| & 1 & 3 & 6 & 9| & 5 \\ \Rightarrow & 1 & 2 & 3 & 6 & 9| & 6| \\ \Rightarrow & 1 & 2 & 3 & 5 & 6 & 9| \end{array}$$

A livello di implementazione dobbiamo soltanto aggiungere la ricerca delle successioni già ordinate.

```
straightmergesort(int[] a, int l,r)
	int ll, mm, rr
	do
		rr=l-1
		while rr<r
			ll=rr+1
			mm=ll
			while (mm<r && a[mm+1]>=a[mm])
				mm++
			if mm<r
				while (rr<r && a[rr+1]>=a[rr])
					rr++
				merge (a, ll, mm, rr)
			else
				rr=mm
	until ll=l
```

In questo modo cambiano anche le performance dell'algoritmo. Compare difatti un best case, ad esempio quando l'input è già ordinato. In questo caso avremmo $\Theta(n)$ comparazioni e 0 spostamenti. Il caso medio ed il peggior caso rispecchiano invece straight merge sort, nonché le tempistiche della prima analisi.

## 2.7 Radixsort (pag 121)

Radixsort si basa sulla comparazione delle cifre dei rispettivi numeri. Noi tratteremo solo la forma duale, ma il tutto funziona con qualsiasi forma, compresa quella decimale. Ammettiamo sempre che esista una funzione $z(i,k)$ che ritorna l'$i-$tesima cifra a partire dalla meno significativa del numero $k$. Ad esempio $z(0,517) = 7$ e $z(2,517) = 5$.

### 2.7.1 Radix-exchange sort

L'algoritmo segue un principio abbastanza simile a quicksort: tutti gli elementi il cui $i-$tesimo bit è 0 vanno a sinistra, mentre se è uno vanno a destra. Il modo più semplice è quello, come in quicksort, di far scorrere due puntatori verso il centro e scambiare i numeri che si trovano al posto sbagliato. Dopodiché si richiama l'algoritmo sull' $(i-1)-$tesimo bit e sulle due parti dell'array.

**Esempio**

| | | | | | |
|---|---|---|---|---|---|
| | 1011 | 0010 | 1101 | 0011 | \|3 |
| $\Rightarrow$ | 0011 | 0010; | 1101 | 1011; | \|2 |
| $\Rightarrow$ | 0011 | 0010; | 1011; | 1101; | \|1 |
| $\Rightarrow$ | 0011 | 0010; | 1011; | 1101; | \|0 |
| $\Rightarrow$ | 0010 | 0011 | 1011 | 1101 | |

**Implementazione**

```
radixexchangesort(int[] a, int l,r,b)
	int i,j
```

```
if r>l
        i=l-1
        j=r+1
        while (true)
                do
                        i++
                until
                        z(b, a[i])=1 || i>=j
                do
                        j--
                until
                        z(b, a[j])=0 || i>=j
                if i>=j
                        break
                swap (a[i], a[j])
        if b>0
                radixexchangesort(a, l, i-1, b-1)
                radixexchangesort(a, i, r, b-1)
```

## Analisi

L'algoritmo viene richiamato al massimo $b$ volte ed ogni volta impiega $O(n)$ paragoni e movimenti. In totale impiega dunque $O(bn)$ (tempo pseudo polinomiale). Se $b > \log(n)$ radixsort non è l'algoritmo adeguato al problema.

### 2.7.2 Fachverteilung

Questo algoritmo viene spesso anche chiamato bucket sort, poiché vengono utilizzati dei bucket per ordinare i numeri. L'idea di base è semplice: per ordinare una serie di numeri con m cifre si inizia ad ordinarli dividendoli in bucket rappresentanti la cifra, sempre da destra a sinistra, dal basso all'alto. Dopo uno step bisogna "raccogliere" nuovamente la sequenza dai bucket e rieseguire il procedimento con la cifra successiva. Dopo m step la sequenza è ordinata. L'algoritmo può praticamente essere diviso in due parti, dividere e raccogliere.

#### Esempio

Sequenza originale: 40, 13, 22, 54, 15, 28.Primo step:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 40 | | 22 | 13 | 54 | 15 | | | 28 | |
| $F0$ | $F1$ | $F2$ | $F3$ | $F4$ | $F5$ | $F6$ | $F7$ | $F8$ | $F9$ |

La sequenza provvisoria diventa 40, 22, 13, 54, 15, 28. Secondo step:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 15 | 28 | | | | | | | |
| | 13 | 22 | | 40 | 54 | | | | |
| $F0$ | $F1$ | $F2$ | $F3$ | $F4$ | $F5$ | $F6$ | $F7$ | $F8$ | $F9$ |

Il risultato finale, letto da sinistra a destra e dal basso all'alto è quindi: 13, 15, 22, 28, 40, 54.

### Implementazione

Un approcio naiv sarebbe quello di creare m bucket di dimensione n, in modo che nel peggior caso tutti i valori starebbero in un solo bucket. Questo però comporterebbe a O(mn) spazio extra, quando realmente ne serve solo $\Theta(n)$ (ci sono sempre esattamente n elementi). Ci sono due possibili soluzioni al problema. La prima è di usare liste lineari dalla dimensione non fissa, la seconda è quella di sondare quanti elementi andranno in ciascun bucket e dividere un array di dimensione fissa in m scompartimenti tramite degli indici di divisione (Verteilungszahlen). Analizziamo dapprima la prima possibilità:

```
radixsort_list(int[] a)
	List<Integer>[] l=new List<Integer>[m]
	int i,j,t
	for j=0:m-1
		l[j]=new List<Integer>() //inizializza la lista
	for t=1:l-1
		//fase di divisione
		for i=1:n
			j=z(t,a[i])
			l[j].pushtail(a[i]) //aggiunge all'ultima posizione
		//fase di raccolta
		i=0
		for j=0:m-1
			while (!l[j].isEmpty())
				a[i]=l[j].pophead()
				i++
```

Adesso la possibilità utilizzando un sondaggio prima della fase di divisione:

```
radixsort_sond(int[] a)
	int[] b //array di dimensione n, numerato da 1 a n
	int[] c //array di dimensione m, Verteilungszahlen, da 0 a m-1
	int i,j,t
	for t=0:l-1
		//sondaggio
		for i=0:m-1
			c[i]=0 //inizializzazione
		for i=1:n
			j=z(t,a[i])
			c[j]++
		//futuri indici dell'array b
		c[m-1]=n+1-c[m-1]
		for i=2:m
			c[m-i]=c[m-i+1]-c[m-i]
		//fase di divisione
		for i=1:n
			j=z(t,a[i])
			b[c[j]]=a[i]
			c[j]++
		//fase di raccolta
		for i=1:n
			a[i]=b[i]
```

### Analisi

Entrambe le implementazioni richiedono $O(l(m+n))$ passaggi e memoria di $O(n+m)$.

## 2.8 Untere Schranke (pag 153)

Per ora tutti gli algoritmi che usano comparazioni (praticamente tutti tranne radix sort) impiegano almeno $\Omega(n\log(n))$.

**Theorem 2**

Ogni algoritmo di ordinamento impiega nel caso medio e nel caso peggiore almeno $\Omega(n\log(n))$ paragoni per ordinare n chiavi.

**Dimostrazione** Per dimostrare questo teorema valutiamo tutte le possibili n! permutazioni di n chiavi e le inseriamo in un albero binario. Inizialmente l'algoritmo generico non conosce nulla sulla permutazione corrente, ma man mano che compara le chiavi si sposta tra i rami fino a raggiungere la foglia contenente la giusta permutazione. Praticamente un nodo i:j rappresenta una comparazione. Esso ha due figli: a sinistra se $a[i] < a[j]$ e a destra se $a[i] \geq a[j]$. Dato che ci sono n! permutazioni possibili devono per forza esserci n! foglie. Ora non ci rimane che dimostrare che l'altezza massima e media di un albero binario con k foglie sia di almeno $log_2(k)$. In questo caso avremmo:

$$log(k!) \geq \log((\frac{n}{2})^{\frac{n}{2}}) = \frac{n}{2}\log(\frac{n}{2}) \in \Omega(n\log(n))$$

■

**Theorem 2**

L'altezza massima e media di un albero binario con k chiavi è di almeno $log_2(k)$.

**Dimostrazione** La prima parte è triviale, in quanto un albero di altezza t può avere al massimo $2^t$ foglie. La seconda parte viene dimostrata per assurdo. Sia T il più piccolo albero per cui l'ipotesi non regge. T ha k foglie, dunque $k \geq 2$ e T deve avere un Teilbaum $T_1$ con $k_1$ foglie a sinistra e $T_2$ con $k_2$ folgie a destra. Chiaramente $k_1 + k_2 = k$. Dato che sia $T_1$ sia $T_2$ sono più piccoli di T, allora le loro altezze medie devono essere maggiori di $log_2(k_1)$ e rispettivamente $log_2(k_2)$. Ovviamente l'altezza media di T deve essere maggiore di uno rispetto ai due Teilbäume. Quindi:

$$\begin{aligned} altezzamedia(T) &= \frac{k_1}{k}(\text{altezza media}(T_1)+1) + \frac{k_2}{k}(altezzamedia(T_2)+1) \\ &\geq \frac{k_1}{k}(log_2(k_1)+1) + \frac{k_2}{k}(log_2(k_2)+1) \\ &= \frac{1}{k}(log_2(2k_1) + log_2(2k_2)) = f(k_1,k_2) \end{aligned} \tag{2.1}$$

Secondo la condizione che $k_1 + k_2 = k$, f assume valore minimo quando $k_1 = k_2 = \frac{k}{2}$ quindi:

$$\text{altezza media}(T) \geq \frac{1}{k}(\frac{k}{2}log_2(k) + \frac{k}{2}log_2(k)) = log_2(k)$$

Si incappa dunque in una contraddizione.

■

# Chapter 3

# Search

## 3.1 Auswahlproblem (pag 168)

Il primo approcio nel cercare di trovare l'i-tesimo elemento più piccolo è quello di ordinare dapprima la sequenza per poi eliminare i primi i-1 elementi. In questo caso si ottiene un tempo di $O(n\log(n))$. Seguendo lo stesso principio di quicksort è però possibile arrivare alla soluzione in tempo lineare. Per l'implementazione faremo uso del metodo, simile a quello usato in quicksort:

```
int dividi(int l,r, pivot)
       /* divide a[l], ... , a[r] in due gruppi
          a[l], ..., a[m-1] sono < pivot
          a[m], ..., a[m] sono >= pivot
          restituisce il valore di m */
```

Una volta trovato m abbiamo tre possibilità: la chiave si trova nel primo gruppo o nel secondo o l'abbiamo trovata.

```
trova (int l,r,i)
       int m,v
       if r>l
              scegliere pivot v
              m=teile(l,r,v)
              if i<=m-l
                     trova (l, m-1, i)
              else
                     trova (m,r,i-m+l
       else
              //r=l, qundi i=1
              //l'elemento si trova nella posizione a[l]
```

Se scegliamo male il pivot l'algoritmo impiega $\Omega(n^2)$ passi. Se invece il pivot divide l'array in qn e (1-q)n elementi, con 0.5¡q¡1 impiega:

$$\begin{aligned} T(n) &= T(qn) + cn \\ &\leq cn\sum_{i=0}^{+\infty} q^i = cn\frac{1}{1-q} \in O(n) \end{aligned} \tag{3.1}$$

Per scegliere il pivot adeguato si può usare l'algoritmo proposto da Blum, chiamato "median of median strategy".

1. (caso base) se n¡costante calcola l'i-tesimo più piccolo elemento direttamente.
2. dividi gli n elementi in $\lfloor \frac{n}{5} \rfloor$ gruppi di cinque elementi e un gruppo con i restanti.
3. ordina questi gruppi (in tempo costante) e per ogni gruppo trova la mediana. Se il gruppo ha un numero pari di elementi scegli il più grosso degli elementi medi.
4. applica l'algoritmo ricorsivamente alle $\lceil \frac{n}{5} \rceil$ mediane per trovare la mediana delle mediane.
5. utilizza quest'elemento come pivot e procedi normalmente.

Questa strategia assicura che ci sono almeno

$$3(\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2) \geq \frac{3n}{10} - 6$$

elementi più piccoli di v. Segue quindi che l'algoritmo deve essere richiamato al massimo $\lceil \frac{7n}{10} + 6 \rceil$ volte ricorsivamente. Il tempo impiegato è dunque:

$$T(n) \leq T\lceil \frac{n}{5} \rceil + T\lceil \frac{7n}{10} + 6 \rceil + an \leq ... \leq cn \in O(n)$$

## 3.2 Sequenzielle Suche (pag 173)

Ammettiamo che le n chiavi siano salvate in un array o in una lista. Il modo più semplice per sondare la sequenza è quello di controllare le n chiavi fino ad a trovare l'elemento desiderato. Per fare in modo che l'algoritmo si fermi iniziamo dal fondo e mettiamo uno stopper alla posizione 0.

```
sequentialsearch (int k)
        int i
        a[0]=k //stopper
        i=n+1
        do
                i--
        while a[i] != k
        if i != 0
                //a[i] elemento ricercato
        else
                //non esiste nessun elemento k
```

Chiaramente l'algoritmo impiega 0 passaggi nel miglior caso e n nel peggiore. Nel caso medio:

$$\frac{1}{n} \sum_{i=1}^{n} i = \frac{n+1}{2} \in O(n)$$

## 3.3 Binäre Suche (pag 174)

La ricerca binaria si basa su divide and conquer. Ammettendo di avere un array ordinato si può descrivere l'algoritmo come segue:

1. se la sequenza è vuota la ricerca finisce senza successo, altrimenti guarda l'elemento a[m] (posizione media).
2. se k $<$ a[m] applica sulla sequenza a[1], ..., a[m-1] lo stesso procedimento.
3. se k $\geq$ a[m] applica sulla sequenza a[m], ..., a[n] lo stesso procedimento.
4. se k $=$ a[m] l'elemento è stato trovato.

```
binsearch(int l,r,k)
        int m=(l+r)/2
        if l > r
                //ricerca conclusa senza successo
        else
                if k < a[m]
                        binsearch(l, m-1, k)
                else if k > a[m]
                        binsearch(m+1, r, k)
                else
                        //a[m]=k, elemento trovato
```

Un'altra variante iterativa è la seguente:

```
binsearch(k)
        int m,l,r
        l=1
        r=n
        do
                m=(l+r)/2
                if k < a[m]
                        r=m-1
                else
                        l=m+1
        until
                k=a[m] || l>r
        if k=a[m]
                binsearch=m
        else
                binsearch=0
```

In questo modo una ricerca non impiega più che $\lceil log_2(n+1) \rceil \in O(n)$ passaggi.

## 3.4 Interpolationsuche (pag 179)

Nella ricerca binaria scegliamo ogni volta l'elemento a metà dell'array come pivot. Non sempre si tratta di una buona idea. Ad esempio se cerchiamo

nell'elenco telefonico "Bernasconi" non lo apriamo al centro. L'unica differenza dalla ricerca binaria è dunque la scelta della m:

$$m = \lceil l + \frac{k - a[l]}{a[r] - a[l]}(r - l) \rceil$$

Nel miglior caso impieghiamo un numero di paragoni costante, ma nel caso medio $O(log_2(log_2(n))$ comparazioni. Perdiamo però a causa della maggiore complessità delle operazioni aritmetiche e nel caso sfortunato, in cui impiega O(n) paragoni.