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 aniché 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
2 15 43 17 4
2 4 43 17 15
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
2 15 43 / 17 4
2 15 17 43 / 4
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
2 15 43 17 4
2 15 17 43 4
2 15 17 4 43
2 15 4 17 43
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:
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).