ASD - Algoritmi e Strutture Dati

Elia Ronchetti
@ulerich

2022/2023

Indice

Capitolo 1

Introduzione algoritmi

1.1 Che cos'è un algoritmo?

Un algoritmo è

- Una sequenza di istruzioni elementari
- Agisce su un input per produrre un output
- Risolve un problema computazionale

Un algoritmo deve essere corretto e efficiente.

Corretto Significa che deve funzionare per qualsiasi input valido

Efficiente Deve occupare il minor spazio possibile ed impiegare il minor tempo possibile.

L'efficienza di un algoritmo si misura in termini di spazio e tempo

1.2 Analisi di un algoritmo

Per analizzare l'efficienza di un algoritmo si calcola il numero di istruzioni eseguite, ma esso non è univoco, varia in base all'input ricevuto, è quindi necessario individuare il **caso migliore** e il **caso peggiore**, essi si analizzano a parità di dimensioni, per questo non dipendono da essa. Dire che il caso migliore è quando l'array è vuoto non ha senso ai fini dell'analisi.

Per avere un'idea dei tempi di esecuzione è necessario calcolare il **Caso Medio**

NON è la media tra caso peggiore e caso migliore!

1.3 Regole sullo Pseudocodice

Gli algoritmi saranno scritti in Pseudocodice secondo le seguenti regole

- Il codice sarà simil C/Java
- Cicli: for, while, do-while
- Condizioni: if, else
- Indentazione + begin/end
- Commenti /*.....*/
- Assegnamenti A = 5, A := 5, A $\leftarrow$ 5
- Test del valore A == 5
- Variabili: locali
- Array A[i] $\rightarrow$ i $\rightarrow$ 1 ... n
- Gli Array partono da 1
- Dati sono considerati oggetti con attrivuti (come length(A) per gli array)
- Puntatori: liste dinamiche
- Funzioni/Procedure - I parametri sono passati per valore (non per indirizzo)
- Operatori booleani AND e OR sono cortocircuitati, quindi se ho a AND b valuterò prima a e se è falso non valuterò b, per OR invece se a è vero non valuterò b

Macchina RAM (Random Access Machine) La macchina su cui verranno eseguiti gli algoritmi sarà considerata RAM e quindi con le seguenti Caratteristiche

- Memoria ad accesso diretto
- No limiti memoria
- Sistema monoprocessore

1.4 Esame

L'esame sarà uno scritto con esercizi e domande di teoria. I parziali sono tendenzialmente riservati al primo anno, ma è possibile scrivere una email al prof 2 settimane prima del parziale e chiedere di poterlo sostenere anche se si è di un altro anno, sarà a sua discrezione concedere o meno questa opportunità. Si possono recuperare i parziali, è possibile anche tentare un recupero per migliorare un voto già positivo, accettando il rischio di che se il voto preso nell'esame di recupero è minore di quello originale si dovrà accettare quel voto.

1.5 Definizioni di base

Algoritmo Corretto Un algoritmo si definisce corretto se per tutti gli input si ottiene il risultato desiderato, l'algoritmo è corretto solo se garantisce la correttezza del risultato.

Algoritmo efficiente Minor utilizzo di **Spazio** e **Tempo**.

Determinare l'efficienza di un algoritmo

Il primo passo è determinare il numero di istruzioni eseguite dall'algoritmo dato che così facendo non dipendo dalla potenza dell'hardware e dall'input.

Operazioni valutazione algoritmo

1. Conto le istruzioni **eseguite**

2. Determinare T peggiore - T migliore - T medio (la media non è fra T peggiore e T migliore)

Il tempo non sarà una quantità in secondi, ma dipenderanno da un parametro n T(n).
Quando devo scegliere un algoritmo mi baso sulla funzione n, dato che al crescere dell'input la funzione crescerà in modo lineare, quadratico, cubico, ecc. e questo mi mostrerà come cresce il tempo in funzione di n.
A parità di n controllo il fattore moltiplicativo.

Esempio I polinimoniali hanno tempi di esecuzione accettabili, mentre i tempi esponenziali sono intrattabili, il problema è che esistono algoritmi esatti, ma sono esponenziali, per questo sono inutili dato che non con grandi input non si fermano mai.
Determinare il **Caso peggiore** serve per capire quanto tempo devo aspettare al massimo, quindi dopo quanto tempo avrò sicuramente un risultato, il **Caso minore**, determina il tempo minimo che devo aspettare, il **Caso medio** determina mediamente quanto tempo devo aspettare (non è la media fra T peggiore e T migliore).

1.6 Ricerca Sequenziale

```
int RicSeq(int k, int v[])
 1  i=1
    while v[i] != k AND i <= length(v)
        i++
 1  if i <= length(v)
1/0     return(i)
    else
1/0     return(-1)
```

In questo semplice algoritmo per la ricerca sequenziale di un numero all'interno di un array ci concentreremo sulla ricerca del caso peggiore e quello migliore.
Ricordiamo che il caso peggiore e migliore si determina a parità di dimensione dell'input, non ha senso dire che il caso migliore è il vettore vuoto.
I numeri a fianco indicano il numero di volte per cui ogni operazione è eseguita. Else non viene considerato dato che non viene effettuato un controllo, quando l'if è falso rimanda alle istruzioni sotto l'else (a livello di linguaggio macchina è un'etichetta che indica al codice dove effettuare la jump nel caso in cui la condizione dell'if non fosse vera).

1.6.1 Cosa devo identificare

Devo identificare:

- Quando si verifica (in che condizioni)
- Il tempo di esecuzione

Quando si verifica

Migliore Indicato come **t**. Posso pensare (sempre intuitivamente) che il caso migliore è quando il numero si trova nella prima posizione del vettore. Effettivamente in questo caso il ciclo while non viene mai attuato (a parte la verifica della condizione)

Peggiore In questo caso devo identificare per quale input la ricerca sequenziale performa peggio, intuitivamente posso pensare che il caso peggiore è quando il numero non è presente nel vettore, dato che devo scorrere tutto il vettore.
Analizzando l'esecuzione di tutte le istruzioni posso confermare la mia ipotesi.

Importante Non basta fare un esempio, per identificare il caso migliore e peggiore, devo descrivere (anche in forma verbale) per quali input si verifica, in questo caso abbiamo detto infatti, quando il numero non è presente oppure quando il numero si trova nella prima posizione.

Analisi tempo di esecuzione Il tempo di esecuzione dipende dal numero di operazioni eseguite, i tempi di esecuzioni maggiori si concentrano spesso nei cicli (for, while, do-while, ecc.), ma è comunque importante valutare tutte le istruzioni.

Tempo di esecuzione caso migliore Indicato con $t(n)$ In questo caso vengono eseguite solamente 4 istruzioni

```
int RicSeq(int k, int v[])
 1  i=1
 1  while v[i] != k AND i <= length(v)
        i++
 1  if i <= length(v)
 1      return(i)
   else
 0      return(-1)
```

Le quattro istruzioni con dei commenti

- i=1
- while (controlla una volta sola dato che trova subito che k = v[i])
- if i $\leq$ length(v)
- return(i)

Si tratta di una funzione costante $t(n) = 4$, solitamente le funzioni riguardanti i tempi di esecuzione sono in funzione di n, ma in questo specifico caso la funzione non dipende dalla dimensione dell'input, infatti se il numero è all'inizio del vettore a prescindere dalle dimensioni dell'array non dovrò mai scorrere il vettore (vedremo che non è praticamente mai così per il caso peggiore)

Analisi esecuzione caso peggiore Indicato con $T(n)$

```
int RicSeq(int k, int v[])
 1   i=1
n+1  while v[i] != k AND i <= length(v)
 n         i++
 1   if i <= length(v)
 0        return(i)
     else
 1        return(-1)
```

Il ciclo while viene eseguito n+1 volte dato che ogni volta che incremento i++, devo verificare di non essere uscito dall'array, quindi verrà eseguito sempre una volta in più rispetto all'incremento.
Questo significa che $T(n) = 3 + 1 + 2n = 2n + 4$ dove n è la dimensione dell'input, in questo caso la dimensione dell'array.
Questa funzione indica il numero di istruzioni eseguite (non è propriamente un tempo).

1.6.2 Caso medio

Indicato come $T_m(n)$, è il tempo medio di esecuzione, non è la media fra il caso migliore e peggiore.
In questo caso dobbiamo chiederci, cosa ci aspettiamo che succeda mediamente?
In questo caso mi aspetto che il caso medio sia che k si trovi in posizione $\frac{n}{2}$, quindi che sia a metà. Mi aspetto quindi che il caso medio sia il caso peggiore diviso due.
In generale il caso medio è un po' più complesso dato che richiederebbe di conoscere la distribuzione di probabilità dell'input.
$T_m(n) = 2\frac{n}{2} + 4 = n + 4$

Possibile obiezione Sto mettendo sullo stesso piano le istruzioni if e i while che magari hanno più condizioni da verificare, effettivamente il while impiega leggermente più tempo, posso scrivere una funzione indicando con c $(c_1, c_2, c_3, \dots)$ il tempo di esecuzione che può avere ogni istruzioni, il problema è che la funzione diventa enorme e poco comprensibile, alla fine a me interessa capire l'ordine di grandezza di esecuzione per poter capire se l'algoritmo è efficiente e per poterlo confrontare con altri algoritmi, alla fine se confronto un algoritmo che ha come caso medio $1000n$ con uno che ha come caso medio n^2 sceglierò comunque il primo, quindi non mi interessa se alcune funzioni ci mettono un po' di più di altre.

1.7 Ricerca binaria (o dicotomica)

La ricerca binaria si basa sull'assunzione che l'array dato in input sia ordinato.
L'elemento da cercare viene confrontato ripetutamente con l'elemento al centro della struttura dati e, in base al risultato del confronto, viene ridotta la porzione di dati in cui si effettua la successiva ricerca. Questo processo viene ripetuto fino a quando l'elemento desiderato viene trovato o fino a quando la porzione di dati da esaminare diventa vuota.

```
int Ricerca_Binaria(int k, int v[])
    sx = 1;
    dx = length(v);
    m = (sx+dx) div 2; //divisione intera
    while (v[m] != k AND sx <= dx)
        if v[m] > k
            dx = m-1;
        else
            sx = m+1;
        m = (sx+dx) div 2;
    if sx <= dx
        return(m);
    else
        return(-1);
```

Ricerca e analisi caso migliore Il caso migliore è quando il numero da cercare k si trova esattamente al centro dell'array, quindi in posizione $\frac{n}{2}$, dato che non entra mai nel ciclo.
Verifichiamo quante volte vengono eseguite le istruzioni:

```
int Ricerca_Binaria(int k, int v[])
1    sx = 1;
1    dx = length(v);
1    m = (sx+dx) div 2; //divisione intera
1    while (v[m] != k AND sx <= dx)
         if v[m] > k
             dx = m-1;
         else
             sx = m+1;
         m = (sx+dx) div 2;
1    if sx <= dx
1        return(m);
     else
         return(-1);
```

In totale abbiamo $t(n) = 6$ istruzioni eseguite. Anche in questo caso il valore non dipende da n perchè non dipende dalla dimensione del vettore, basta che l'elemento si trovi al centro dell'array.

Ricerca e analisi caso peggiore Qui il caso peggiore è quando k non è presente in v.
Verifichiamo quante volte vengono eseguite le istruzioni:

```
int Ricerca_Binaria(int k, int v[])
1     sx = 1;
1     dx = length(v);
1     m = (sx+dx) div 2; //divisione intera
tw+1  while (v[m] != k AND sx <= dx)
tw        if v[m] > k
ft            dx = m-1;
          else
ff            sx = m+1;
tw        m = (sx+dx) div 2;
1     if sx <= dx
          return(m);
      else
1         return(-1);
```

Quante volte cicla il while? Dipende dall'input!
Indichiamo questo valore con t_w
Quante volte testo l'if nel while? Sempre t_w volte.
Quante volte l'if risulta vero e quindi eseguo il codice nell'if? Anche qua

dipende, quindi indico questo valore con t_{if}, per l'else assegno f_{if} (flase if). Assegno t_w anche per l'assegnazione di m.

Semplificazioni per il calcolo del tempo Posso considerare il numero di esecuzioni dell'if e dell'else come una sola variabile, quindi sempre t_w, in totale ho quindi $2t_w$ esecuzioni, all'iterno del while, mentre i controlli del while sono $t_w + 1$ (il +1 è per il controllo del while finale).
Ottengo quindi:

```
int Ricerca_Binaria(int k, int v[])
1     sx = 1;
1     dx = length(v);
1     m = (sx+dx) div 2; //divisione intera
tw+1  while (v[m] != k AND sx <= dx)
2(tw)    if v[m] > k
               dx = m-1;
          else
               sx = m+1;
tw         m = (sx+dx) div 2;
1     if sx <= dx
          return(m);
      else
1         return(-1);
```

In totale avrò che $T(n) = 6 + 4t_w$
Ma t_w quante volte viene eseguito? Dato che ogni volta l'array viene diviso in 2 parti ho la seguente progressione:

- Passo 0 n
- Passo 1 $\frac{n}{2}$
- Passo 2 $\frac{n}{2^2}$
- Passo 3 $\frac{n}{2^3}$
- ...

Questa è una tipica progressione logaritmica infatti avrò che $n = 2^r$ e quindi $r = log_2(n)$. Quindi il tempo peggiore sarà $T(n) = 6 + 4log_2(n)$.

Caso medio In questo caso mi aspetto che mediamente dovrà dividere metà delle volte l'array rispetto al caso peggiore, quindi avrà $t_m(n) = 6 + 2log_2(n)$.

Capitolo 2

Calcolo tempo di esecuzione algoritmi

In questo capitolo vedremo come calcolare il tempo di esecuzione di algoritmi, partiremo ad analizzare algoritmi di ordinamento, più che per la loro funzione primaria, per studiare come è possibile affrontare un problema con diverse tecniche e come l'utilizzo di diverse tecniche influenzi anche notevolmente l'efficienza.

2.1 Selection Sort

Ordinamento per selezione, dove per selezione intendo che ad ogni passo seleziono il valore minimo presente nell'array, scambio l'elemento più piccolo con l'elemento in prima posizione, mi sposto sul secondo valore e cerco il più piccolo, andrà a sostituirlo nella seconda posizione, e così via fino a quando non ho un solo valore da ordinare.
Qui di seguito il codice:

```
void SelSort(int A[])
  (n-1)+1  For i = 1 to length(A) - 1
  (n-1)        Pmin = i
  sum(n-i)          For j = i + 1 to length(A)
  sum(n-i)               if A[j] < A[Pmin]
  t-if                       Pmin = j
  //Variabile appoggio per lo scambio
  (n-1)             app = A[i]
  (n-1)             A[i] = A[Pmin]
  (n-1)             A[Pmin] = app
```

Length(A) - 1, perchè l'ultimo valore sono sicuro che sarà il più grande di tutti dato che per gli tutti gli altri elementi ho cercato il minimo.

Tempo esecuzione Il For viene eseguito (n-1)+1 volte perchè devo considerare anche il controllo finale che viene effettuato.
Il secondo For invece è più complesso da gestire perchè dipende anche da i che è esterna al ciclo stesso. Ogni volta che eseguo il secondo For l'array si restringe di 1 perchè ogni volta ordino un valore (trovando il minimo) quindi avrò una progressione del tipo $(n + (n-1) + (n-2) + (n-3) + \cdots + (1))$, quindi $(n-i)+1$, devo considerare perchè che verrà eseguito ogni volta che il primo For viene eseguito quindi $\sum_{i=1}^{n-1}(n-i)$.
Il tempo di esecuzione sarà quindi:

$$5(n-1) + 2\sum i = 1^{n-1}(n-1) + t_{if}$$

Dove 5(n-1) e la sommatoria non dipendono dall'input, mentre t_if sì.

Ricerca e Analisi caso peggiore Il caso peggiore è A ordinato al contrario, dato che in questo caso l'if viene eseguito ogni volta (dato che A[j] sarà sempre minore di A[Pmin]).
t_if avrà il seguente tempo di esecuzione $(n-1)+(n-2)+(n-3)+\cdots+1 = \sum_{i=1}^{n-1} = \sum_{i=1}^{n} i$
Quindi:

$$T_p(n) = 5(n-1) + 3\sum_{i=1}^{n-1}(n-1) = 5(n-1) + 3\sum_{i=1}^{n} i$$

Ricordiamo questa equivalenza:

$$\sum_{i=1}^{f} i = \frac{f(f+1)}{2}$$

Quindi sostituendo otteniamo:

$$5(n-1) + 3\frac{(n-1)n}{2}$$

Dato che a noi interessa l'ordine di grandezza e non ci interessano i dettagli possiamo approssimare questo risultato come:

$$\approx 5n + n^2 \approx n^2$$

Nota In realtà se è decrescente quando scambio il minimo in fondo all'array con il primo valore, che sarà il maggiore, sta ordinando entrambi gli elementi, però per semplificazione consideriamo che l'if viene eseguito ogni volta.

Ricerca e Analisi caso migliore Il caso migliore è quando l'array è ordinato dato che non eseguo mai l'if, quindi dato che $t_{if} = 0$ ottengo:

$$t_m(n) = 5(n-1) + 2\sum_{i=1}^{n-1}(n-1) + 0 = 5(n-1) + \frac{2}{2}(n-1)n = 5n + n^2 \approx n^2$$

Notiamo quindi che il caso migliore e peggiore non sono molto diversi, anzi hanno lo stesso ordine di grandezza.

Insertion Sort

In questo caso ordino partendo dal primo numero e controllando il secondo, verifico se il primo è maggiore del secondo e nel caso li scambio, poi controllo il terzo numero e procedo a ordinarlo insieme a primi due e così via.

Esempio É come ordinare un mazzo di carte pescandole mano, quindi inizialmente ho due carte, le ordino confrontandole ed eventualmente scambiandole, poi pesco la terza e la ordino con le altre due e così via, fino a quando non ho pescato tutte le carte.

2.1.1 Implementazione

```
    void InsSort(A[])
(n-1)c  For i = 2 to length(A)
(n-1)c      App = A[i]
(n-1)c      j = j - 1
sum(tw)     while App < A[j] AND j > 0
sum(tw)         A[j+1] = A[j]
sum(tw)         j--
(n-1)c      A[j+1] = App
```

Uso il For perchè in ogni caso devo ordinare tutti i numeri dell'array, parto da 2 perchè un numero da solo è ordinato, quindi è inutile analizzarlo, inoltre j sarebbe uguale a 0, e noi partiamo a contare da 1 gli indici dell'array.

2.1.2 Tempi d'esecuzione, caso migliore e peggiore

Tempo di esecuzione Faremo alcune approssimazioni dato che cerchiamo l'ordine di grandezza dei tempi di esecuzione:

- Non consideriamo l'ultima istruzione di controllo del For (quella che ci fa uscire dal ciclo)
- Non considero neanche l'ultima istruzione di controllo del While

Il For quindi richiederà n-1 istruzioni, dove **c** è un generico tempo di esecuzione per ogni istruzione, dato che parte da 2 e non da 1.
Il while non ha un numero di volte fisso per cui è vero, dipende dall'input! Ci sarà un caso peggiore e uno migliore, scrivo quindi $\sum_{i=2}^{n} t_w$ per indicare il numero di volte in cui il While è vero, la sommatoria è inserita perchè devo sommare tutte le esecuzioni While per ogni i-esima esecuzione del ciclo For. Qui di seguito la spiegazione della sommatoria

$$tw_{i=2} + tw_{i=3} + tw_{i=4} + \cdots + tw_{i=n}$$

$$\sum_{i=2}^{n} tw_i$$

Il tempo di esecuzione è quindi

$$t_{is}(n) = 4c(n-1) + 3c\sum_{i=2}^{n} tw_i$$

Dato che non è possibile definire in maniera univoca il numero di volte in cui il while verrà eseguito andiamo a determinare il caso migliore e quello peggiore.

Caso migliore Il vettore è già ordinato. Analizzando il codice dobbiamo verificare cosa può cambiare, in questo caso il While, quindi qual è il caso dove il while viene eseguito meno volte? Quando App $<$ A[j] sempre, quindi $tw_i = 0$ per ogni i e questo accade quando A[] è già ordinato.

$$t_m(n) = 4c(n-1) + 3c * 0 = 4c(n-1)$$

Caso peggiore Il while viene eseguito il maggior numero di volte per ogni i, cioè $tw_i = j = i - 1$. Questo si verifica quando A è ordinato al contrario.

$$T_p(n) = 4c(n-1) + 3c\sum_{i=2}^{n}(i-1)$$

Converto la sommatoria

$$\sum_{i=2}^{n} i - 1 = 1 + 2 + 3 + \cdots + (n-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

Inserisco nella formula

$$4c(n-1) + 3c\frac{(n-1)n}{2} \approx 4cn + 3c\frac{n^2}{2} \approx n^2$$

Approssimazione caso migliore Il caso migliore lo posso approssimare come n.

2.1.3 Confronto tra Selection e Insertion sort

Grazie alle approssimazioni possiamo confrontare i due algoritmi e notiamo subito che il caso migliore di Insertion è migliore di Selection Sort dato che è nel primo caso è n, mentre nel secondo è n^2.
Insertion Sort è quindi migliore di Selection Sort.

2.2 Nozioni per rappresentare i tempi di esecuzione

$$T(n) = O(n^2)$$

O (o grande) indica il limite asintotico superiore del tempo di esecuzione (il caso peggiore), in altre parole indica il tempo massimo che posso aspettare per ricevere il risultato.
Mentre:

$$t(n) = \Omega(n)$$

Indica il limite inferiore del tempo di esecuzione (il caso migliore), quindi il tempo minimo che devo aspettare.
Nei casi dove questi due casi corrispondono si indicano con:

$$\Theta(n^2)$$

Θ indica che il tempo è uguale (approssimativamente) per tutti gli input.

2.2.1 Limiti asintotici

I limiti asintotici sono delle funzioni per cui sono sicuro che la funzione presa in considerazione $f(n)$ è sempre minore del limiti asintotico. In questo caso dico che la funzione è asintoticamente limitata $f(n) = O(n^2)$. Le lettere O, Ω, Θ, indicano una costante moltiplicativa per cui posso moltiplicare la funzione e trova una funziona per la quale la mia funzione n non sarà mai maggiore (nel caso di O), o minore (nel caso di Ω) della funzione che la limita asintoticamente. Questa costante indica tutto ciò che ho trascurato nel calcolo dei tempi, posso sceglierla a piacere per verificare che la funzione non violerà mia i limiti asintotici. (FORMALIZZARE QUESTA PARTE AGGIUNGENDO FORMULE - GUARDA APPUNTI IPAD)

$$O(g(n)) = \{f(n) : \exists k > 0, n_0 \geq 0 \text{ t.c} \\ 0 \leq f(n) \leq k * g(n), \forall n \geq n_0\}$$

Questo ci porterà a fare delle approssimazioni come per esempio non considerare l'ultimo controllo del for o del while, questo produrrà delle situazioni che possono sembrare non sensate, per esempio potrà risultare che la condizione del while non viene mai eseguita, in realtà non è vero, perchè almeno 1 volta verrà eseguito, però per le approssimazioni che abbiamo scelto di fare non lo consideriamo.

Attenzione

1. Devo scegliere la funzione che solo per alcune costanti sia più grande/-piccola di quella confrontata ($\exists k$ $(t.c)$), altrimenti sto scegliendo un o (o-piccolo) dove la funzione è grande/piccola per ogni costante scelta ($\forall$).
2. Non mi interessa se il il termine di ordine maggiore sia seguito da addizioni o sottrazioni
3. Di fronte a un tempo costante come Ω indichiamo 1

2.2.2 Proprietà

O, Ω, *Theta* hanno le seguenti proprietà:

- Sono Transitivi
- Simmetria vale solo per Θ
- Simmetria trasposta per O e Ω - Es. $2n = O(n^2)$ $n^2 = \Omega(2n)$

2.3 Esercizio ricerca elementi V2 in V1

Richiesta Dati 2 vettori V1 e V2, con n valori interni, quanti elementi di V2 compaiono in V1? Valutare quindi i tempi di esecuzione dell'algoritmo.

Osservazioni Dati i seguenti due vettori

$$\text{V1} = (7, 4, 4, 4, 12)$$
$$\text{V2} = (3, 7, 4, 15, 20)$$

Gli elementi di V2 che compaiono in V1 sono 2, il 4 non devo contarlo più volte!
Viceversa, se ci fosse stato il 4 più volte in V2 e anche 1 sola volta in V1, andava contato più volte.

2.3.1 Implementazione

```
    int confronta(V1[], V2[])
c*1         cont = 0
c*n         for i=0 to length(V2)
c*n             j=1
sum(c*twi)      while (V2[i] != V1[j]) AND j <= length(V1)
sum(c*twi)          j++;
c*n             if j <= length(V1)
c*tif               cont++
c*1         Return(cont)
```

Dove sum(c*twi) sarebbe $\sum_{i=1}^{n} c * tw_i$

Tempo algoritmo

$$2c * 1 + 3c * n + c * t_i f + 2c \sum_{i=1}^{n} tw_i$$

Posso portare fuori la c e il 2 dalla sommatoria dato che sono 2 costanti.

Ricerca caso peggiore Analizzo le due istruzioni variabili, cioè

- if
- while

Quale pesa di più? Il controllo dell'if viene eseguito ogni volta, l'incremento del conteggio non sempre, però conta comunque 1 istruzione, e comunque l'if viene sempre eseguito, mentre il while può essere rieseguito diverse volte, questo mi suggerisce che devo cercare un caso dove il while viene eseguito molte volte, dato che è potenzialmente molto più pesante dell'if.
In generale ci capiteranno sempre 2 quantità in contrasto, come in questo caso abbiamo l'if e il while, dovremo determinare la quantità più pesante.
Il caso peggiore è quindi quello in cui il while cicla il maggior numero di volte possibile, quando tw_i è max per ogni i, quindi quando nessun elemento di V2 è presente in V1.

Tempo

$$tw_1 = n,\ tw_2 = n\ \ldots tw_n = n$$
$$T_p(n) = 2c + 3c * n + c * t_i f + 2c \sum_{i=1}^{n} n$$
$$= 2c + 3c * n + 0 + 2cn^2$$

Perchè $\sum_{i=1}^{n} n = n + n + n + n + n...$ per n volte $= n^2$
Da questo risultato mi rendo conto che l'ordine di grandezza è n^2, per esprimerlo in maniera formale scrivo:

$$= O(n^2)$$

Ricerca caso migliore Il caso migliore è quello dove il while non viene mai eseguito (0 esecuzioni del while ad ogni iterazione del for).
Il caso migliore NON è i 2 array sono uguali, perchè ci saranno delle iterazioni in cui il while verrà eseguito, il caso migliore è quando V2 contiene un solo elemento ripetuto n volte e lo stesso elemento è presente in V1[1].

V1 1,2,3,4
V2 1,1,1,1

Tempo

$$tw_i = 0 \quad \forall i \rightarrow t_i f = n$$
$$t_m(n) = 2c + 3cn + cn + 2c \sum_{i=1}^{n} 0$$
$$= \Omega(n)$$

In questo caso (opposto al precedente) while non viene mai eseguito e if n volte.

Verifica caso medio É ragionevole pensare che su molte esecuzioni il primo valore di V2 in V1 si trovi mediamente al centro.
T medio(n) $\rightarrow V_2[j]$ viene trovato sempre in $V_1(\frac{n}{2}) \rightarrow tw_i = \frac{n}{2} \forall i$

$$= 2c + 3cn + cn + 2c \sum_{i=1}^{n} fracn2 =$$
$$= 2c + 4cn + cn^2 = \Theta(n^2)$$

Riassumendo

- Tempo peggiore - $O(n^2)$
- Tempo migliore - $\Omega(n)$
- Tempo medio - $\Theta(n^2)$

2.4 Riassunto ordini di grandezza

Qui di seguito sono riportati gli ordini di grandezza in ordine crescente

- 1 (intesa come costante)
- $\log n$
- n
- $n \log n$
- n^2
- n^3
- 2^n

2.5 Somma di due valori costituiti da bit

Testo Sommare due valori binari contenuti in due array e salvare il risultato in un terzo array.

Struttura

$$A[1 \dots n]$$
$$B[1 \dots n]$$
$$C[1 \dots n+1]$$

Dove i bit più significativi sono quelli a sinistra (quelli iniziali) e quelli meno significativi a destra.

Considerazioni Devo effettuare la somma bit a bit, quindi si configurano 3 casi

- Sommo 0 con 0 e ottengo 0
- Sommo 1 con 0 (o 0 con 1) e ottengo 1
- Sommo 1 con 1 e ottengo 1, ma devo ricordarmi del riporto!

Conviene quindi gestire ogni somma come una somma a 3 bit (bit A, bit B, bit di riporto).

2.5.1 Implementazione

```
void Somma(A[],B[])
c*1 riporto = 0;
    //parola chiave per indicare i-- nel for e' Downto
c*n for i = length(A) Downto 1
c*n     c[i+1] = A[i]+B[i]+riporto;
c*n     if c[i+1]<=1
c*tif       riporto = 0;
c*fif   else
c*fif       riporto = 1;
c*fif       c[i+1]=c[i+1]-2;
c*1 C[1]=riporto;
```

Alla fine riduco in due casi dove se la somma dei due bit mi restituisce 1 o 0 allora il riporto = 0, mentre se è ¿ 1 setto il riporto a 1 e tolgo 2 dalla somma, perchè:

- Se ho sommato 1 e 1 con riporto 0, ottengo 2 e ho riporto 1, togliendo 2 avrò 0 e in binario $1 + 1 + 0 = 10$
- Se ho sommato 1 e 1 con riporto 1, ottengo 3 e ho riporto 1, togliendo 2 avrò 1 e in binario $1 + 1 + 1 = 11$

2.5.2 Calcolo tempo

Come al solito tif è t_if e indica quando l'if è vero e fif è f_if e indica quando l'if è falso.

$$t(n) = 2c * 1 + 3c * n + 1c * t_if + 2c * f_if$$

Caso migliore Quando ho due vettori che sommati non danno mai riporto, quindi non esistono due bit a 1 nella stessa posizione i.

$$\begin{aligned} t_m(n) = 2c + 3cn + c * n + 2c * 0 = \\ = 4cn + 2c = \Omega(n) \end{aligned}$$

Caso peggiore Quando c'è sempre riporto, quindi quando:

$$\begin{gathered} \text{A[n]=B[n]=1} \\ \text{A[i]} \neq 0 \text{ e B[i]} \neq 0 \text{ per lo stesso i} \\ \forall 1 \leq i \leq n-1 \end{gathered}$$

$$T_p(n) = 2c + 3cn + c * 0 + 2c * n = 5cn + 2c = O(n)$$

Confronto due casi Il caso migliore è effettivamente più veloce del peggiore dato che il migliore ha 4cn operazioni, mentre il peggiore ha 5cn, ma hanno lo stesso ordine di grandezza, quindi per valori grandi non ci sarà molta differenza.
In questo caso dato che il caso migliore e caso peggiore hanno lo stesso limite asintotico posso usare Θ e dire che il tempo dell'algoritmo sarà:

$$t_{somma}(n) = \Theta(n)$$

Tempo medio Quando il limite asintotico del caso peggiore e caso migliore coincide, sicuramente il tempo medio avrà lo stesso limite asintotico, quindi non avrà molto senso calcolarlo, in questo caso per esempio ci aspettiamo che in metà dei bit avrò riporto e in metà dei bit no, posso scrivere la formula specifica, ma comunque otterrò sempre lo stesso limite asintotico, dato che se il limite superiore e inferiore coincidono per ordine di grandezza, se cercao qualcosa in mezzo avrà necessariamente lo stesso ordine di grandezza.

2.6 Esercizio - Calcolo tempo codice già scritto

```
int f-y(n)
c    z=n;
c    t=0;
c log n while z>0
c log n      x= z MOD 2
c log n      z = z DIV 2
c log n      if x == 0
c tif*n          for i=1 to n
c tif*n              t=t+1
c    return(t)
```

2.6.1 Calcolo tempo istruzioni

Capiamo che il while viene eseguito $\log n$ volte perchè dipende da $z > 0$ e z viene diviso ogni volta per 2 (divisione intera) e questa è la tipica progressione logaritmica. Infatti questo non è un vero e proprio while, dato che il numero di istruzioni eseguite è determinato già all'inizio, potrebbe essere sostituito con un for.

Tempo di esecuzione generico

$$t(n) = 3c + 4c \log n + 2ct_{if}n$$

Caso migliore $t_if = 0$, cioè quando non entro mai nell'if e quindi quando ho sempre resto nella divisione intera di z per 2, ma questo quando si verifica? Si verifica quando dividendo per due un numero dispari, ottengo sempre un numero dispari. La divisione intera per due tenendo da parte il resto non è altro che la procedura per convertire un numero decimale in un numero binario, quindi ottengo sempre 1 quando sto convertendo un numero decimale che in binario è composto da solo 1 e questo si verifica quando ho un numero del tipo $2^k - 1$. Esempi validi sono 63, 127, 1023, ecc.

$$t_m(n) = 3c + 4c \log n + 0c = \Omega(\log n)$$

Caso peggiore If sempre vero e quindi $n = 2^k$

$$t_{if} = \log n$$
$$T_p = 3c + 4c \log n + 2cn * \log n = O(n \log n)$$

In questo caso non posso usare Θ perchè i limiti asintotici del caso migliore e caso peggiore sono diversi.

Caso medio Il caso medio è quello dove per metà dei bit ho riporto e per metà no, quindi è dove eseguo il for metà volta, quindi sarà $\frac{n \log n}{2}$.

2.6.2 Miglioramento Algoritmo

C'è un modo per migliorare questo algoritmo? Sì, il for è sostituibile con t=t+n.
Così facendo il caso peggiore (e quello medio) diventa:

$$O(n) = logn$$

2.7 Esercizio for innestati

```
    f_x(n)
c       r=0
c(n-1)  for i=1 to n-1
sum(n-i)    for j=i+1 to n
sum(sum(j))     for k=1 to j
                    r=r+1
        return(r)
```

2.7.1 Calcolo tempo

In questo caso abbiamo 3 for annidati, con indici legati uno con l'altro.
In questi casi è necessario partire dal for più interno, il numero di volte per cui sarà eseguito sarà la sommatoria del for sopra stante, se ha più di 1 for, ci sarà una sommatoria per ogni for, per intenderci, il for più interno nel codice sopra riportato avrà la seguente sommatoria:

$$\sum_{i=1}^{n-1}(\sum_{j=i+1}^{n} j)$$

Analizziamo questa formula:

- Il j più interno è il tempo di esecuzione del for più interno (for k=1 to j), dato che è un semplice incremento di variabile, quindi internamente va inserito il tempo di esecuzione del for più interno
- La sommatoria centrale riporta invece gli indici del for centrale, infatti parte da j = i+1 fino a n (come scritto nel for)
- Infine la sommatoria più esterna prende gli indici del primo for cioè parte da n fino ad arrivare a n-1

Questa formula rappresenta il numero di esecuzioni del for più interno.

For centrale In questo caso avrò una sola sommatoria $\sum_1^{n-1} n-1$

For esterno Come al solito in questo caso conto semplicemente quante volte viene eseguito guardando fino a dove arrivare il contatore, in questo caso n-1.

Risoluzione formula

$$t(n) = c + c(n-1) + c\sum_{i=1}^{n-1}(n-i) + 2c\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} j + c$$

$$\approx 2c + c * n + c\sum_{i=1}^{n-1} i + 2c \cdots =$$

$$= 2c + cn + c\frac{(n-1)(n)}{2} + 2c\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} j + c$$

- Con l'approssimazione tolgo i -1 per esempio in $c(n-1)$, dato che non è rilevante.

- $\sum_{i=1}^{n-1}(n-i)$ se la leggo al contrario è come se stessi facendo $n-1, n-2, n-3, \ldots, n-n$ quindi la posso scrivere come $\sum_{i=1}^{n-1} i$ e quindi convertirla con la solita formula della sommatoria
- Per la doppia sommatoria devo cercare di convertire prima di tutto quella più interna

La doppia sommatoria la gestisco nella seguente maniera:

- Devo convertire quella interna per scriverla in una forma che posso poi convertire in qualcosa contenente

- dato che ho $j = i+1$ posso pensare di riscriverla come $\sum_{j=1}^{n} j - (\sum_{j=1}^{i} j)$ e poi togliere tutte le i che sto trascurando (la sommatoria dopo il meno serve proprio a bilanciare la prima)

- Adesso posso convertire le due sommatorie come $\frac{n(n+1)}{2} - \frac{i(i+1)}{2}$, dato che nel secondo caso come indice in alto alla sommatoria ho i

- Quindi ora ho convertito la prima sommatoria e tornando alla formula iniziale ho $\sum_{i=1}^{n-1}(\frac{n(n+1)}{2} - \frac{i(i+1)}{2})$

- Ora posso fare delle approssimazioni, perchè tanto arriverò a scrivere un limite asintotico

- La prima parte è fissa dato che la sommatoria indica l'incremento di i, ma nel primo termine ho solo n, quindi avrò $\approx \sum_{i=1}^{n-1} \frac{n^2}{2} - \sum_{i=1}^{n-1} \frac{i(i+1)}{2}$

- La prima sommatoria la approssimo da i=1 a n (anzichè n-1), tanto il -1 non influenzerà l'ordine di grandezza, quindi avrò $\approx n * \frac{n^2}{2}$ mentre la seconda porto fuori il termine costante $(\frac{1}{2})$, $-\frac{1}{2}\sum_{i=1}^{n-1}(i^2 + i)$

- Ora divido come prima i due termini in 2 sommatorie distinte $\approx \frac{n^3}{2} - \frac{1}{2}\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i$ in entrambe le sommatorie il limite superiore $n-1$ è stato approssimato a n per il solito discorso che -1 non influenza l'ordine di grandezza

- La seconda sommatoria so come convertirla, mentre per la prima introduciamo una nuova formula (si trova in fondo a questo elenco), avrò quindi $\approx \frac{n^3}{2} - \frac{1}{2}(\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2})$

- Così facendo ho risolto entrambe le sommatorie e ho solo n nella formula!

Conversione importante da ricordare

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Conclusione risoluzione formula Alla fine quindi avrò

$$\approx \frac{n^3}{2} - \frac{1}{2}(\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2})$$

Ora che ho solo n posso fare ulteriori approssimazioni per poter cercare un limite asintotico

$$\approx \frac{n^3}{2} - \frac{1}{2}\frac{2n^3}{6} - \frac{1}{4}n^2$$

Ho considerato solo gli ordini di grandezza maggiori, in questo modo posso dedurre che

$$\begin{aligned} t(n) &\simeq 2c + cn + \frac{cn^2}{2} + 2c(\frac{n^3}{2} - \frac{n^3}{6} - \frac{n^2}{4}) \\ &\simeq 2c + cn + cn^2 + 2cn^3 = \Theta(n^3) \end{aligned}$$

Ho tolto sia le frazioni che i termini che vengono sottratti perchè comunque sono più piccoli dato che il confronto è tra $\frac{n^3}{2}$ e $\frac{n^3}{6}$, dato che viene diviso per 6 è più piccolo del primo, n^2 è proprio di un ordine inferiore.

Non c'è caso migliore o peggiore Questo perchè l'algoritmo ha un numero fisso di istruzioni eseguite dato che ci sono solo for, NON ci sono condizioni variabili come if o while, infatti scrivo Θ.

2.7.2 Quando usare le sommatorie

Le sommatorie annidate una dentro l'altra sono da usare solo quando i for sono collegati da indici in comune (in questo caso k,j,i erano collegati), se ho due for che non sono collegati uso semplicemente n per indicare il numero di volte che vengono eseguiti, non userò la sommatoria.

2.7.3 Note per l'esame per le approssimazioni

Se sono palesi le approssimazioni possiamo anche non esplicitare cosa abbiamo fatto, ma se saltiamo molti passaggi o facciamo approssimazioni non palesi dobbiamo esplicitare cosa abbiamo fatto scrivendo due righe.

2.8 Esercizio - Verifica se la matrice data è simmmetrica

```
boolean Sim(M[][])
    i=2, simmetrica=TRUE;
    while i <= length(M) AND simmetrica==TRUE
        j=1
        while j<1 AND simmetrica
            if M[i,j] != M[j,i]
                simmetrica = FALSE;
            j++;
        i++;
    return(simmetrica);
```

Tempo esecuzione

$$t(n) = 3c + 3ctw_1 + 3ctw_2 + ct_{if} \tag{2.1}$$

Sicuramente sarà il ciclo più interno a pesare di più, dato che

Caso peggiore Devo verificare qual è il ciclo peggiore, sicuramente è quello interno, dato che se viene eseguito tante volte verrà rieseguito n volte in base a quante volte eseguirò il primo while, il caso peggiore è quando la matrice è simmetrica.

$$t_if = 0 \qquad M[i,j] = M[j,i] \quad \forall_{i,j} \ldots \tag{2.2}$$

M è simmetrica.

N esecuzioni While

$$tw_1 = n - 1 \tag{2.3}$$

$$tw_{2,i} = \sum_{j=1}^{n-1} i = \frac{(n-1)n}{2} \tag{2.4}$$

Ricordiamoci che il caso peggiore dipende sempre dal numero di istruzioni eseguite, è slegato da cosa deve fare l'algoritmo.

$$T_p(n) = 3c + 3c(n-1) + 3c\frac{(n-1)n}{2} + 0 = \tag{2.5}$$

$$\approx 3c + 3cn + 3cn^2 + 0 = O(n^2) \tag{2.6}$$

Caso migliore Quando entro 1 volta sono nei while e quando l'if è subito vero e setta simmetrica=false, il caso migliore quindi non è semplicemente quando la matrice non è simmetrica, ma è quando subito i primi due valori sono diversi, così mi accorgo subito che la matrice non è simmetrica.

$$t_m(n) = 3c + 3c + 3c + c = 10c = \Omega(1)$$

Notiamo che essendo Omega una costante, il caso migliore non dipende dalla dimensione dell'input, dato che è sempre il primo elemento che mi fa capire che la matrice non è simmetrica.

2.8.1 Come interrompo un ciclo?

Per interrompere un ciclo NON utilizzare break o return, ma utilizzare una variabile booleana come in questo caso.

Caso medio Mediamente mi accoregerò a metà matrice che non è simmetrica, per questo il caso medio è

$$t_m \approx \frac{n^2}{4} = \Theta(n^2)$$

Dato che ho diviso per due il caso peggiore

Capitolo 3

Ricorsione

La programmazione ricorsiva è strettamente legata all'induzione matematica, si basa sul fatto che per risolvere un problema mi riconduco a un problema non ancora risolto, ma più semplice, fino ad arrivare a un caso base già risolto. I vantaggi della ricorsione sono due:

- Più semplice rispetto agli algoritmi iterativi (solitamente)
- La logica ricorsiva è più efficiente rispetto a quella iterativa

3.1 Fattoriale

3.1.1 Iterativo

```
int Fatt(n)
    Ris=1
    For i=n downto 1
        Ris=Ris*i;
    return Ris;
```

3.1.2 Ricorsivo

```
int Fatt(int n)
    if n==0
        return(1);
    else
         Ris=(Fatt(n-1));
         Tot = n*Ris;
         return(Tot);
```

3.2 Potenza ricorsiva

```
int Potenza (int A, int n)
c    if (n == 0)
c tif    return(1)
     else
NO c fif   Ris=A*Potenza(A, n-1)
NO c fif      return(Ris)
```

3.2.1 Calcolo tempo di esecuzione

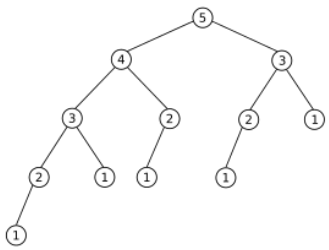
In questo non avendo cicli dobbiamo controllare le funzioni ricorsive, è sbagliato dire che la chiamata ricorsiva impiega $c * f_{if}$, perchè dipende da n non è un tempo di esecuzione costante! Vedremo più avanti come calcolare il tempo di esecuzione.

3.3 La ricorsione è sempre più efficiente rispetto all'iterazione?

Prendiamo l'esempio della sequenza di Fibonacci eseguita ricorsivamente.

```
int fibonacci(int n)
    if (n <= 1)
        return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
```

Se scomponiamo l'esecuzione di questa funzione noteremo che vengono rieseguiti più volte i calcoli per gli stessi numeri, dato che mi ritroverò più volte gli stessi numeri.



Notiamo infatti dall'albero che vengono calcolati più volte gli stessi numeri e questa è una grande inefficienza.
In questi casi quindi l'iterazione risulta migliore rispetto alla ricorsione.

3.4 Ricerca carattere in Array

Dato un Array trova tutte le occorrenze di una lettera data in input (in questo caso z).
Devo sempre ricondurmi all'affermazione ricorsiva, quindi partire da un caso, ricondurmi a uno più piccolo che però non mi da ancora il risultato, ma che mi avvicina sempre di più al caso base.
In questo caso la penso nel seguente modo: Guarda A[n] se contiene z allora 1+tutte le z in A[n-1] altrimenti 0 + tutte le z in A[n-1].
Fino a ricondurmi al primo elemento dell'array, il caso base sarà proprio quello! Controllare quando ho un solo elemento se è z o no, se è z ritorno 1 se no ritorno 0.

3.4.1 Implementazione

```
int trova(char car, char A[], int pos)
    if pos == 1
        if A[1] == 'z'
            return(1)
        else
            return(0)
    else
        Ris=trova(char car, char A[], pos -1)
        if A[pos] == 'z'
            Ris++
        return(Ris)
```

Solitamente nelle funzioni ricorsive per scorrere un array abbiamo bisogno di due indici, per indicare la porzione di array che stiamo analizzando (es int h, int k), in questo caso il primo indice è fisso dato che devo scorrere dall'ultimo valore a scendere fino al primo, quindi sarebbe come avere h=1 fisso e k=k-1, quindi h non lo considero.
Le varie chiamate ricorsive mi portano a scorrere tutti i valori e quando mi riconduco al caso base, la risoluzione di tutte le chiamate aperte mi porta a controllare mano a mano tutti gli indici dell'array restituendo mano a mano il contatore incrementato o meno a in base al fatto che ci sia z o meno.

3.5 Divide et Impera

Si tratta di un approccio di tipo ricorsivo che consiste in 3 fasi:

- Problema P è diviso in due o più sottoproblemi - DIVIDE
- Ogni sottoproblema è risolto ricorsivamente - IMPERA
- Le soluzioni ai sottoproblemi sono riunite a formare la soluzione completa - COMBINA

3.5.1 Ordinamento tramite Divide et Impera - Merge Sort

Problema P, ordina un vettore A di n elementi (usando Divide et Impera):

- Divide - Divido in 2 parti l'array da ordinare, ogni parto con $\frac{n}{2}$ elementi, approssimando verso il basso nel caso il valore non fosse intero (n dispari)
- Impera - Ordina la prima metà, poi ordina la seconda metà
- Combina - Fonde (MERGE) in modo ordinato le due metà ordinate

Merge sort è un algoritmo di ordinamento STABILE.

Definizione di algoritmo di ordinamento STABILE Un algoritmo di ordinamento si dice stabile se elementi di uguale valore al termine dell'ordinamento mantengono tra di loro l'ordine che avevano inizialmente.
Quindi se ho per esempio [1, 5, 3, 2a, 7, 0, 2b], una volta ordinato avrò [0, 1, 2a, 2b, 3, 5, 7], i 2 anche se identici hanno preservato il loro ordina iniziale, quindi 2a si trova ancora prima di 2b.
Non è necessario che un algoritmo di ordinamento sia stabile per poter funzionare, ma questa caratteristica può essere utile per determinati utilizzi (strutture dati).

3.5.2 Esecuzione ordinamento

Dato che è necessario disegnare e usare diversi colori la spiegazione viene riportata scritta a mano, qui di seguito il link per la visualizzazione.

Note importanti Nel libro sembra che l'esecuzione dei sottopassi sia effettata in maniera parallela, NON è così, viene ordinata prima la prima metà, poi le sotto metà della prima, fino ad arrivare ai casi base, poi si torna indietro a ritroso, non si ordinano parallelamente tutti i sotto array.
Nella spiegazione infatti gli ordinamenti sono stati messi a livelli di altezza diversi per sottolineare questo aspetto.
Ecco il PDF Link al PDF.

3.5.3 Implementazione codice

```
void MergeSort (A[], int pin, int pfin)
    if pin < pfin
        m = (pin + pfin) DIV 2 //DIVIDE
        //divisione intera approssima verso il basso
        MergeSort(A[], pin, m) //IMPERA
        MergeSort(A[], n+1, pfin) //IMPERA
        Merge(A[], pin, m, pfin) //COMBINA

B[] //Array di appoggio per Merge

void Merge(A[],In, meta, fin2)
    I1=In
    I2=meta+1
    Ib=In
    while I1 <= meta AND I2 <= fine
        if A[I1]<=A[I2] //Maggiore uguale rende Merge stabile
            B[Ib] = A[I1]
            IB++
            I1++
        else
            B[Ib] = A[I2]
            IB++
            I2++
    while I1 <= meta
        B[Ib] = A[I1]
        Ib++
        I1++
    while I2 <= fine
        B[Ib] = A[I2]
        Ib++
        I2++
    Ib=In
    while Ib <= fine //ricopio l'array ordinato nell'array inizi
        A[Ib]=B[Ib]
        Ib++
```

- pin e pfin sono rispettivamente l'indice iniziale e finale dell'array che sto analizzano (serve per dividere l'array per la divide)

- Li sommo e divido per 2 (divisione intera) per dividere l'Array e poi do in input la pin e la metà a una chiamata ricorsiva di MergeSort - DIVIDE

- Chiamo una funzione Merge (diversa da MergeSort) che ordina i sottoarray

Merge

- Qua è come inserire indice inizio1, fine1, inizio2, fine2, ma dato che fine1 è la metà dell'array, e inizio2 e m+1, passo solo meta come parametro, ma il ragionamento in realtà è quello di passare gli indici di inizio e fine di 2 array

- if A[I1] $\leq$ A[I2] rende Merge stabile perchè se sono uguali scelgo di ordinare prima A[I1] quindi l'elemento di sinistra, questo mantiene i due elementi com'erano ordinati inizialmente

- Dopo il primo while aggiungo 2 while perchè potrebbe essere che la prima parte è ordinata e ho finito gli elementi, ma nella seconda ci siano ancora elementi, servono quindi a continuare a copiare gli elementi rimasti in una delle due parti quando l'altra parte è già stata ordinata. Chiaramente verrà eseguito uno dei due while, mai tutti e due.

- Ultimo while serve a ricopiare tutto nell'array iniziale A (copia B in A)

- Quest'ultimo ciclo while in realtà è un for perchè la dimensione dell'array da copiare è fissa, so già quante istruzioni devo eseguire

Osservazione, perchè uso B e non lavoro direttamente su A? Si può fare un MergeSort che non usa array di appoggio, ma è molto più complicato da implementare perchè durante l'ordinamento del Merge mi cambierebbe l'ordine degli elementi nell'array dato che li sto ordinando, e questa complicazione è difficile da gestire.

3.5.4 Calcolo tempo Merge Sort

In $\longleftrightarrow$ Fin = n
cioè la quantità di elementi compresi tra In e Fin nell'array

Funzione Merge In questa funzione ho 4 while

- W1 Confronta i 2 array
- W2 Ricopia i restanti della prima parte
- W3 Ricopia i restanti della seconda parte
- W4 - Ultimo ciclo while è in realtà un for che viene eseguito n volte

W1 confronta i 2 elementi, W2 e W3 non vengono mai eseguiti insieme, o eseguo l'uno o l'altro.
Se conto quante volte itero il primo e quante il secondo e terzo ottengo n perchè se per esempio ho 100 elementi e nel primo while ricopio 80 elementi in B dai confronti e me ne restano 20 nella seconda parte, i restanti 20 elementi saranno copiati nel terzo while. In pratica W1+W2+W3 è uguale a n iterazioni.
Il tempo della MERGE ci verrà quindi $\Theta(n)$, che è stato possibile calcolare senza troppi sforzi perchè è comunque una funzione iterativa, sapevamo già come fare.

Tempo MergeSort Questa è una parte ricorsiva.
Per prima cosa applichiamo un ragionamento che vale per tutti gli algoritmi divide et impera:

Il tempo di esecuzione di una Divide et Impera è dato da:

$$D(n) + I(n) + C(n)$$

Quindi la somma delle 3 fasi (Divide Impera Combina)

Consideriamo anche che **le parti Divide e Combina sono iterative, mentre Impera è ricorsiva**.
Nella merge sort:

- Divide è calcola M (metà) $\Theta(1) = c$
- Combina è MERGE $\Theta(n)$
- Divide + Combina è asintoticamente $\Theta(n)$
- Impera le chiamate ricorsive sono 2 e gli vengono dati $\frac{n}{2}$ elementi

Equazione di ricorrenza della Merge Sort

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & n > 1 \end{cases}$$

Si chiama equazione di ricorrenza perchè T(n) compare anche a destra dell'equazione, in questo caso in $2T(\frac{n}{2})$. Per calcolare i tempi di esecuzione di un algoritmo ricorsivo generico abbiamo bisogno di risolvere queste equazioni di ricorrenza dove la nostra incognita (nel nostro caso i tempi di calcolo) si trova sia a destra e che sinistra dell'uguale.

Come risolvere un'equazione di ricorrenza Ci sono principalmente tre metodi:

- Metodo iterativo - Itero l'applicazione della ricorrenza volta per volta l'incognita fino a quando arrivo a un caso base
- Metodo per sostituzione - Ipotizzo quale possa essere la soluzione dell'equazione e poi verificarla con una dimostrazione
- Metodo principale o metodo dell'esperto - Funziona SOLO per i Divide et Impera, controlla se si verificano determinate condizioni (3) e in base alle condizioni ci restituisce i tempi di esecuzione

Equazione generica per tutti gli algoritmi Divide et Impera

$$T(n) = \begin{cases} \Theta(1) & \text{caso base} \\ aT(\frac{n}{b}) + D(n) + C(n) \end{cases}$$

In generale negli algoritmi Divide et Impera $a \geq 1$ e $b > 1$, questa differenza è molto importante, perchè **un Divide et Impera lo riconosco dal fatto che l'input è diviso in una frazione con un deminatore strettamente maggiore di 1!**, mentre A può anche essere 1, ci sono dei casi limite dove il sottoproblema è 1 solo (nel merge sono 2), l'importante per essere divide et impera è che la parte che consideriamo nella chiamata ricorsiva non è n - qualcosa, ma è $\frac{n}{\text{qualcosa}}$, qualcosa maggiore di 1.
La parte $\frac{n}{b}$ può essere solitamente approssimata, per esempio nella Merge Sort se è pari ho la metà esatta, se è dispari invece approssimo dato che la divisione qua mi sembra intera, ma in alcuni casi (non comuni), non è così, devo verificare se il cambio del comportamento nelle due casistiche (per esempio pari e dispari) non mi cambi radicalmente il problema.

3.6 Metodi per calcolo tempo in algoritmi ricorsivi

Vedremo i 3 sopra citati

3.6.1 Metodo iterativo o dell'albero di ricorsione

Prende in input un'equazione di ricorrenza e iterativamente cerca di ricondursi al caso base.

Esempio

$$T(n) = \begin{cases} 6 & n = 1 \\ 8 + T(n-1) & n > 1 \end{cases}$$

$$T(n) = 8 + T(n-1) = 8 + [8 + T(n-2) = 2 * 8 + T(n-2)] = 2 * 8 + [8 + T(n-3)] = 3 * 8 + T(n-3) = 3 * 8 + [8 + T(n-4)] = 4 * 8 + T(n-4)$$

Genericamente a un passo k avrò:

$$k_n = k * 8 + T(n-k)$$

Se scelgo $k = n - 1$ ottengo:

$$(n-1)8 + T(n + (n-1)) = (n-1) * 8 + T(1) = (n-1) * 8 + 6 = \Theta(n)$$

Genericamente il tempo di esecuzione sarà questo (tempo medio), potrebbe essere che il caso peggiore o migliore cambiano l'equazioni di ricorrenza.

Altro esempio

Anche in questo caso link per la spiegazione scritta a mano: Ecco il PDF Link al PDF.

3.6.2 Metodo di sostituzione

Non molto utilizzato, consiste nell'ipotizzare un tempo di esecuzione e poi dimostrare per induzione che è quello.
Non ci sarà all'esame, serve solo sapere che esiste.

3.6.3 Teorema dell'esperto o Metodo principale

Questo è molto utile, dato che è abbastanza semplice da usare, ma si può applicare solo a Divide et Impera, funziona nella maggior parte dei casi Divide et Impera, in alcuni però non funziona, dato che per applicarlo devo sapere qualche regola, che in alcuni casi non viene rispettata.

Equazione di ricorrenza Divide et Impera

- Parte iterativa - $D(n) + C(n) = F(n)$ Divide, Combina
- Parte ricorsiva - $aT(\frac{n}{b})$ Impera

La parte più problematica è la parte ricorsiva, ma se noi scriviamo $n^{\log_b a}$ non abbiamo il tempo di esecuzione della parte ricorsiva, ma abbiamo un'idea approssimata. Dobbiamo confrontare la parte iterativa e ricorsiva e capire quale pesa di più (o se asintoticamente hanno lo stesso peso).

Caso 1 - Parte ricorsiva pesa di più Se

$$F(n) = O(n^{(\log_b a)-\epsilon}), \quad \exists \epsilon > 0 \rightarrow T(n) = \Theta(n^{\log_b a})$$

Questo significa che se F(n) è limitato superiormente dal tempo di esecuzione della parte ricorsiva - ϵ allora il tempo di esecuzione dell'algoritmo è quello della parte ricorsiva, devo trovare quindi almeno un ϵ per cui si verifichi questa condizione.

Caso 2 - Parte ricorsiva e iterativa hanno lo stesso tempo

$$F(n) = \Theta(n^{\log_b a}) \rightarrow T(n) = \Theta(n^{\log_b a} \log n)$$

Questo caso è quello in cui i tempi sono gli stessi, infatti nell'equazioni si equivalgono e non è presente nessuna ϵ, in questo caso basterà semplicemente scrivere il tempo esecuzione è il tempo della parte ricorsiva moltiplicata per un fattore $\log n$, quindi il tempo è $n^{\log_b a} * \log n$.

Caso 3 - Parte iterativa pesa di più

$$F(n) = \Omega(n^{(\log_b a)+\epsilon}) \quad \epsilon > 0$$

In questo caso non basta che si verifichi questa condizione, ma devo anche verificare che:

$$\text{se} \quad aF(\frac{n}{b}) \leq k * F(n) \qquad k < 1$$

Se è verificata anche questa condizione allora posso dire che:

$$\rightarrow T(n) = \Theta(f(n))$$

Osservazione Dato che il teorema dell'esperto funziona solo con Divide et Impera, e questa tipologia di algoritmi come caso base hanno $\Theta(1)$, posso anche sottointendere il caso base e scrivere solo il caso generale.

3.6.4 Esempio applicazione teorema dell'esperto su MergeSort

Ricordiamo che l'equazione di ricorrenza di MergeSort è la seguente:

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & n > 1 \end{cases}$$

Quindi abbiamo che la per la parte ricorsiva $n^{\log_b a}$

- a = 2 cioè il numero di sottoproblemi in cui viene diviso il problema
- b = 2 cioè per quanto viene diviso l'input ($\frac{n}{2}$)

Quindi sostituendo abbiamo che:

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

E abbiamo che la parte iterativa è $F(n) = \Theta(n)$, quindi la parte ricorsiva e iterativa hanno lo stesso tempo di esecuzione, ci troviamo nel Caso 2, quindi abbiamo che:

$$T(n) = \Theta(n^{\log_b a} * n) = \Theta(n * \log n)$$

3.6.5 Altro esempio

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

Come detto prima sott'intendo il caso base $\Theta(1)$
Qua abbiamo che $a = 9$ e $b = 3$ e $F(n) = n$. La parte ricorsiva sarà:

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

Abbiamo quindi che $F(n) = O(n^2)$ dato che la parte ricorsiva ha un tempo di esecuzione più grande quindi siamo nel caso 1, ma non basta dire questo, dobbiamo anche trovare un ϵ tale per cui:

$$F(n) = O(n^{2-\epsilon}) \qquad \epsilon > 0 \qquad (es) \quad \epsilon = \frac{1}{2}$$

Notiamo che per quel valore di ϵ il tempo resta asintoticamente identico, per questo possiamo dire che il tempo di esecuzione è:

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$$

Cioè il tempo di esecuzione è determinato dalla parte ricorsiva.

3.6.6 Esercizio teorema dell'esperto

Anche in questo caso il link per il PDF scritto a mano, Link

3.6.7 Esercizi Divide et Impera

Link al PDF → PDF.
Esercizi relativi a:

- Scrittura algoritmi in Divide et Impera
- Utilizzo del teorema dell'esperto per calcolare i tempi di esecuzione
- Simulazione esecuzione algoritmo di ordinamento (in questo caso MergeSort)

3.7 Esercizi esame (primo parziale o prima parte completo)

Tipologie di esercizi presenti all'esame:

- Simulare l'esecuzione di un algoritmo di ordinamento (uno fra Selection Sort, Insertion Sort, Merge Sort), dove si spiegano brevemente i passaggi più importanti, vanno indicati nello schema tutti i passaggi. NB NON mettere tutto in parallelo, mettere su livelli diversi in base al momento in cui vengono eseguite le istruzioni.
- Scrivere un algoritmo utilizzando il metodo Divide et Impera
- Calcolare il tempo di esecuzione di un algoritmo (iterativo o ricorsivo)
- Domanda di teoria

3.8 Quick Sort

Algoritmo di ordinamento Divide et Impera mediamente più rapido di MergeSort, dato che i tempi sono i seguenti:

$$\begin{array}{cc} \Theta(n \log n) & \text{Caso medio} \\ \emptyset(n^2) & \text{Caso peggiore} \end{array}$$

Quick sort a differenza di merge sort, ha un caso peggiore, ed è un caso abbastanza lento dato che è dell'ordine n^2. I tempi di esecuzione asintotici

però nascondo delle costanti e le costanti di Quick sort sono più piccole di Merge sort, per questo è mediamente più veloce, oltretutto il caso peggiore è molto raro.
Tendenzialmente il caso peggiore può essere un problema quando l'algoritmo ha come caso di peggiore un caso che si presenta spesso.
Quick Sort si comporta molto male quando l'array è già ordinato o semi ordinato, però vedremo che c'è una tecnica per far sì che non ci sia un caso che lo metta in crisi, creando una versione randomizzata.

3.8.1 Differenze tra Merge Sort e Quick Sort

Note positive di Quick Sort

- Mergesort è un algoritmo di ordinamento non in loco dato che sfrutta delle variabili di appoggio che dipendono dalle dimensioni dell'input (quindi si occupa più spazio con input più grandi)
- Quick sort è un algoritmo di ordinamento in loco dato che ha delle variabili di appoggio, ma sono costanti, non dipendono dall'input (lo spazio occupato dalle variabili è sempre lo stesso), questo significa che l'algoritmo occupa poca memoria

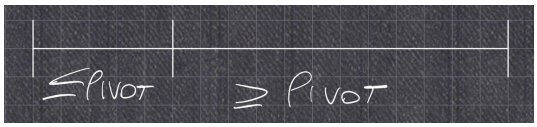
Note negative di Quick Sort

- Il caso peggiore di Quick Sort è n^2, quindi peggiore di Merge, anche se vedremo che il caso peggiore è raro, la presenza di esso mi permette di garantire un tempo di esecuzione maggiore, rispetto a Merge
- Quick Sort non è stabile, quindi elementi con lo stesso valore possono avere un ordine di verso rispetto a quello iniziale alla fine dell'esecuzione

3.8.2 Logica Divide et Impera

Nel Quick sort c'è un ribaltamento della complessità delle parti dato che la divide è più complessa della combina.

Divide L'array è diviso in 2 parti (non necessariemente uguali, possono essere due parti con una quantità di valori molto diversa fra loro).
Per decidere come dividere l'array l'algoritmo prende un elemento come riferimento denominato PIVOT e considera i numeri a sinistra "Piccoli" e quelli a destra "Grandi".



Come Pivot scelgo il più semplice, che è il primo elemento, potrei anche scegliere un elemento non dell'array, ma conviene prendere un numero dell'array perchè se prendo un Pivot troppo piccolo o grande rischio di ricadere nel caso peggiore. Se scelgo il primo elemento dell'array sono sicuro che quello almeno è un valore dell'array e se sono fortunato ho una buona idea dei valori che ci sono nell'array.
Ordino a sinistra gli elementi uguali o più piccoli e a destra quelli uguali o più piccoli. NON viene specificato che quelli che erano a destra stanno a destra e viceversa, per questo motivo Quicksort non è stabile.
Questa fase di Divide viene chiamata **PARTITION**.

Impera Ricorsivamente ordina la prima parte e la seconda parte.

Combina Non fa niente, perchè dovrebbe affianca la prima parte con la seconda, ma l'algoritmo è in loco, quindi è già tutto ordinato e nell'array.

Nota sulle edizioni del libro di testo

- La prima e seconda edizione del libro riportano Quicksort con il partizionamento di Hoare (che è quello che faremo noi)
- Dalla terza in poi viene riportata quella di Lomuto (non si sa perchè)

3.8.3 Simulazione esecuzione

Anche in questo caso appunti a mano, Quicksort.

3.8.4 Implementazione

```
QuickSort(A[], Inizio, Fine)
    if Inizio < Fine
        Q = Partition(A[],Inizio,Fine)
        QuickSort(A[],Inizio,Q)
        QuickSort(A[],Q+1,Fine)
```

La combina non c'è perchè a fine Impera mi risulta tutto ordinato, else non c'è perchè se Inizio non è più ¡ Fine significa che non devo fare più nulla.

```
Partition(A[], I, F)
    Pivot=A[I]
    sx=I-1
    dx=F+1
    while sx < dx
        do
            sx++
        while A[sx] < Pivot
        do
            dx--
        while A[dx] > Pivot
        if sx < dx // controllo per scambio valori
            app = A[sx]
            A[sx] = A[dx]
            A[dx] = app
    return(dx)
```

Osservazioni

- sx e dx partono fuori perchè così ad ogni iterazione io li faccio spostare, così facendo devo farli partire fuori perchè all'inizio dell'esecuzione così analizzeranno i primi valori, posso anche far partire dal primo e ultimo valore, ma la gestione degli spostamenti in questo caso risulterebbe più complessa

- Fare fermare gli indici degli array quando si accavallano o trovano il Pivot previene che che essi sforino i limiti dell'array

3.8.5 Scelta del Pivot

Come Pivot posso fare qualsiasi scelta, tranne nel caso in cui scelgo l'ultimo valore e quello è il maggiore di tutti perchè in quel caso il taglio del vettore mi produrrebbe 2 parti costituire da, 1 parte tutto ciò che sta a sinistra del Pivot compreso il Pivot (in questo caso tutto il vettore), dall'altra parte un vettore vuoto, e questo causa un loop delle chiamate ricorsive perchè in realtà non stanno dividendo il vettore, continua a passare lo stesso vettore.

Soluzione Se scelgo l'ultimo valore lo scambio con qualsiasi altro valore e lo tengo come Pivot, così non c'è possibilità che generi un loop.

3.8.6 Tempi di esecuzione Partition

La partition fa scorrere gli indici fino a quando non si accavallano, se per esempio scorrono tutti senza mai bloccarsi fino a quado si incontrano a metà hanno fatto n iterazioni, se per esempio scambiano ogni volta fanno $\frac{n}{2}+\frac{n}{2}+\frac{n}{2}$ tutti e 3 i while, quindi alla fine fanno circa n.
Possiamo dire quindi che

$$T_{\text{Part}}(n) = \Theta(n)$$

3.8.7 Tempo di esecuzione Quicksort

$$T_{QS}(n) \begin{cases} \Theta(1) \\ n + 2T(Q) + T(n-Q) \end{cases}$$

L'array non viene diviso in 2 metà uguali, possiamo affermare comunque che è b + 2 e quindi ho $T(\frac{n}{2})$? Dobbiamo verificarlo.
Iniziamo dicendo che il tempo dipende dalla velocità di Partition n + la velocità di risoluzione di Quicksort in un array con Q elementi (prima parte) + la seconda parte che è $T(n-Q)$, solo che Q dipende dall'input.
Prendiamo due casi estremi per controllare il tempo di esecuzione.

1 - Ipotiziamo che l'array viene sempre diviso bene a metà Ipotesi poco realistica, ma che utilizziamo per verificare i tempi di esecuzione.
Qui i tempi sono:

$$T_{QS} = 2T(\frac{n}{2}) + n = \Theta(n \log n)$$

Dato che è esattamente il tempo di MergeSort.

1 - Ipotizziamo che 1 elemento è da una parte e n-1 dall'altra Cioè la divisione più sbilanciata.
In questo caso il tempo è:

$$T_{QS} = T(1) + T(n-1) + n = 1 + T(n-1) + n = T(n-1) + n$$

Risolvo T(n-1) scrivendo il suo albero.
Avrò la seguente equazione:

$$n+n+(n-1)+(n-2)+(n-3)+\cdots+(2) = n+(\sum_{i=1}^{n} i)-1 = n-1+\frac{n(n-1)}{2}$$

In questo caso quindi è circa n^2, per determinare se è il caso peggiore dovrei fare uno studio di funzione sull'equazione di ricorrenza, però intuitivamente, quando viene effettuata la divisione sbilanciata? Quando l'array è ordinato! Perchè il primo valore, che sceglierò come Pivot, sarà il più piccolo di tutti e quindi avrò le 2 parti sbilanciate dato che un array avrà 1 valore solo, il Pivot che è il più piccolo, e l'altro array contenente n-1 valori.
Se A è ordinato o ordinato al contrario sono nel caso in cui il taglio è sempre sbilanciato e quindi il caso peggiore, quindi avrò un tempo tendente a n^2.
Possiamo quindi dire che il tempo di esecuzione di Quick Sort è:

$$TQ_s(n) \leq CQ^2$$

Minore o uguale perchè nel caso peggiore è proprio n^2.

3.8.8 Come ridurre di molto la probabilità che si scelga un Pivot pessimo

Si randomizza la scelta del Pivot, così anche se viene dato un array ordinato o un array con tutti i valori più piccoli nella prima parte ad ogni esecuzione di Partizion (quindi ad ogni chiamata ricorsiva) il Pivot viene scelto il maniera casuale:

```
Partition
    Pos=Random(I,F)
    Scambio(A[I], A{Pos})
```

Aggiungendo questa parte a Partition non esisterà più un input specifico che mi mette in crisi l'esecuzione dell'algoritmo, dato che ad ogni passo viene scelto casualmente il Pivot, lo scambio avviene perchè se per caso Random seleziona l'ultima posizione e l'ultimo valore è il più grande di tutti si genera

un loop, così facendo scambio il valore scelto con il primo ed elimino la possibilità che l'algoritmo possa andare in loop.
Così facendo ho ridotto di molto la probabilità che si presenti il caso peggiore.
Con varie dimostrazioni comunque notiamo che non basta che il taglio sia sbilanciato in alcuni tagli per aumentare così tanto il tempo, servirebbe che in ogni chiamata ricorsiva faccia male i tagli, ma l'algoritmo è randomizzato, ciò rende estremamente improbabile che questo accada. Mentre la versione non randomizzata risulta più sensibile agli input.

3.9 Esiste un algoritmo di ordinamento con tempo Theta n?

Gli algoritmi che possiamo fare sono infiniti, dobbiamo affermare in maniera molto pesante che un algoritmo non sia possibile da scrivere.
Per quanto riguarda i confronti non è possibile scrivere un algoritmo di ordinamento con un tempo inferiore a $n \log n$. Gli algoritmi sui confronti si basano sul farsi una serie di domande (confronti) per capire qual è l'ordine, quante domande devo fare per capire l'ordine? Se ho n numeri ho $n!$ possibili ordinamenti, facendo un alberto per verificare il numero di domande che mi faccio quando confronto i numeri ottengo che devo farmi almeno 2^k domande per poter ordinare il numero e $2^k > n!$, quindi k deve essere $\geq \log n!$.
Ma questo logaritmo è grande o piccolo? Attraverso l'approssimazione di Stirling approssimiamo il fattoriale e troviamo che questo numero è $\Theta(n \log nn)$, ma quindi $k > \log n! > \Omega(n \log n)$. Questo dimostra che non è possibile far meno di $n \log n$ domande utilizzando i confronti, quindi non è possibile fare meglio di Mergesort e Quicksort, posso ridurre le costanti nascoste, ma asintoticamente non posso fare meglio di così.

Importanza di questa dimostrazione Questa dimostrazione è importante perchè evidenzia la complessità di dimostrare che un algoritmo non si possa fare, devo trovare una caratteristica generale che vale per tutte le tipologie di algoritmi prese in considerazione, in questo caso considero il confronto.

Counting sort

Counting sort è un algoritmo di ordinamento stabile che non si basa sui confronti, infatti riesce ad avere un tempo di esecuzione $O(n)$, ma funziona se il range di valori da ordinare sia 1...k con $k = O(n)$. Counting sort ha un tempo di esecuzione lineare solo se i numeri hanno delle specifiche caratteristiche, in caso contrario performa peggio di Quicksort e Mergesort.
Per essere precisi ha un tempo di esecuzione $O(n + k)$, infatti se ho 1000 numeri da ordinare, con un range di valori compreso da 1 a 1000 avrò un tempo di $O(1000 + 1000)$ essendo $n = 1000$, avrò $O(2n)$ che è ancora un tempo lineare, nel caso invece avessi 1000 valori, ma con un range molto più ampio, per esempio compreso da $1 \ldots 1.000.000$ mi ritrovo ad avere un tempo di esecuzione $O(n + 1.000.000)$, ed essendo n=1000, il tempo di esecuzione non sarà più lineare, in questo caso sarà meglio Quicksort o Mergesort.

3.9.1 Funzionamento

La posizione del numero dipende dal suo valore nel senso che se ho 100 valori e analizzo il 5 so che dovrò circa metterlo all'inizio.
Non si basa sui confronti, ma distribuisce i numeri all'interno dell'array.

Implementazione Counting sort utilizza 3 array d'appoggio:

- $A[1 \ldots n]$
- $B[1 \ldots n]$
- $C[1 \ldots k]$

ed è composto da 3 fasi:

1. Azzera C[] - $\Theta(k)$
2. Conto quante volte compare ogni elementi di A, memorizzandole in C - $\Theta(n)$
3. Somma in C gli elementi da sinistra verso destra da C[2] a C[n] - $\Theta(k)$
4. Scorri A dalla fine verso l'inizio e piazza in B i numeri sulla base dei valori in C - $\Theta(n)$

In pratica l'algoritmo conta tutte le occorrenza dei numeri contenuti in A e salva il conteggio in C, poi somma da sinistra verso destra i vari indici, così facendo otterrà la posizione iniziale di ogni numero ordinato! Nella quarta fase inizierà a copiare i numeri da A nella posizione giusta in B (dalla fine verso l'inizio) e ogni volta che troverà un numero decrementerà il conteggio in C, così facendo poi quando troverà un'altra volta quel numero lo metterà a sinistra del numero precedentemente copiato, così facendo non ordino solo l'array, ma lo ordini stabilmente!

Tempo di esecuzione Sommando le varie fasi ottengo che:

$$T_{CS}(n,k) = \Theta(n) + \Theta(k) + \Theta(n) + \Theta(k) = \Theta(n) + \Theta(k) = \Theta(n+k)$$

3.9.2 Implementazione codice

```
CountingSort(A[], B[], C[], k)
    for i=1 to k
2k      C[i]=0
    for i=1 to length(A)
3n      Pos=A[i]
        C[Pos]++
    for i=2 to k
2k      C[i]=C[i]+C[i-1]
    for i= length(A) downto 1
5n      p1=A[i]
        p2=C[p1]
        B[p2]=A[i]
        C[p1]--
```

Notiamo nuovamente che il tempo è:

$$T_{CS}(n,k) = 4k + 8n = \Theta(n+k)$$

Riassumendo Countingsort

- Risulta più veloce di Quicksort solo per specifici input
- É stabile
- Riordinamento non in loco
- Dà quindi meno garanzie rispetto a Merge o Quick se non sappiamo che tipo di input stiamo trattando

3.10 Radix sort

Radix ci serve per capire perchè ci servono gli algoritmi stabili.
Se devo ordinare per più campi come posso fare?

Esempio

- Ordina per Cognome
- Ordina poi i gruppi con lo stesso Cognome per nome
- Ordina per età

Dovrei lanciare un algoritmo per ordinare tutto per cognome, poi per ogni sottogruppo dovrei lanciare ogni volta un algoritmo di ordinamento per ogni gruppo e sottogruppo, non c'è un modo più efficiente?
Se partissi a ordinare dal gruppo meno importante (nell'esempio età) e poi procedo a ritroso? Funziona! Mi trovo tutto ordinato facendo molti meno passaggi, funziona però se mantengono ad ogni passaggio l'ordinamento effettuato nel passo precedente e questa è una caratteristica degli algoritmi **stabili**, quindi questa tecnica funziona solo se utilizzo un algoritmo stabile.

3.10.1 Implementazione

```
Radix sort(AC[])
    For r = Meno signficativo to significativo
        sort_stabile(A[],R)
```

Si tratta più di una pseudo implementazione per capirci, inoltre la significatività può essere data in input o scelta secondo diversi criteri. Evidenziamo l'utilità degli algoritmi stabili.

Tempo esecuzione Il tempo dipende principalmente dal tempo di esecuzione dell'algoritmo di ordinamento, se per esempio posso usare Counting allora il tempo sarà lineare, se invece non posso usarlo sceglierò Merge e quindi il tempo sarà $n \log n$. Posso scegliere qualsiasi algoritmo di ordinamento, basta che sia stabile.