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 $t_i f$ 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]).
$t_i f$ 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)

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

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

$$tw_1 = n,\ tw_2 = n\ \dots tw_n = n$$

$$\begin{aligned} T_p(n) &= 2c + 3c * n + c * t_i f + 2c \sum_{i=1}^{n} n \\ &= 2c + 3c * n + 0 + 2cn^2 \end{aligned}$$

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

$$tw_i = 0 \quad \forall i \rightarrow t_i f = n$$

$$\begin{aligned} t_m(n) &= 2c + 3cn + cn + 2c \sum_{i=1}^{n} 0 \\ &= \Omega(n) \end{aligned}$$

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 . . . n]
B[1 . . . n]
C[1 . . . 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 $t_i f$ e indica quando l'if è vero e fif è $f_i f$ e indica quando l'if è falso.

$$t(n) = 2c * 1 + 3c * n + 1c * t_i f + 2c * f_i f$$

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:

A[n]=B[n]=1
A[i] $\neq$ 0 e B[i] $\neq$ 0 per lo stesso i
$\forall 1 \leq i \leq n - 1$

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

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 $t_i f = 0$, 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)
r=0
for i=1 to n-1
    for j=
```