Algoritmi e Strutture Dati 2023-24
(M. Benerecetti)

Contents

1 Lezione 02 - 15/09/2023

1.1 Algoritmo di Conteggio

Descrivere un algoritmo che accetta come input un intero $N \geq 1$ e produce in output il numero di coppie ordinate $i, j \in \mathbb{N} \quad (i,j) : 1 \leq i \leq j \leq \mathbb{N}$
Esempio:

- Input:N=4
- Output:10 {(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4),(4,4)}

```
1 Conta(N):
2   ris = 0; //Assegnamento costante (1 operazione elementare)
3   for i=1 to N do //Assegnamento/Incremento + confronto (2 op. elementari)
4     for j=1 to N do //idem for di sopra
5       if i<=j then //2 letture + confronto (3 op. elementari)
6         ris = ris+1 //lettura+scrittura+assegnamento (3 op. elementari)
7   return ris //1 op. elementare
```

Ogni riga ha un costo che corrisponde alle operazioni elementari effettuate, mediamente ogni op. elementari ha un costo di 1 unità di tempo, prendiamo come esempio il for al primo giro: Assegnamento + Confronto (2 op. elementari), invece i successivi giri: Incremento+confronto (2 op. elementari).

Ognuna di queste operazioni (righe) vengono eseguite più di una volta, quindi il costo sarà maggiore, andiamo ad esprimerlo:

- 2) Costo = 1 (fuori dal ciclo)
- 3) La testa viene eseguita $n+1$ volte poiché abbiamo anche l'ultima operazione per uscire dal ciclo, quindi Costo $= 2 * (n+1) = 2 * \sum_{i=1}^{N+1} 1$
- 4) Questo for verrà ripetuto N volte poiché il corpo del for viene eseguito N volte, quindi il suo costo sarà:

$$\underbrace{2}_{\text{costo dell'operazione}} * \underbrace{\sum_{i=1}^{N}}_{\text{for esterno}} \underbrace{\sum_{j=1}^{N+1} 1}_{\text{for interno}}$$

- 5) L'if stando in entrambi i for avrà un costo di: $3 * \sum_{i=1}^{N} \sum_{j=1}^{N+1} 1$
- 6) Questa operazione non ha un numero fisso di volte di esecuzione. Pertanto e necessario stabilirne un algoritmo per decretarne il numero. Pensandoci il numero di volte che questa operazione esegue dipende da N e dall' i fissate in precedenza. Calcolando, anche banalmente a mano, quante operazioni vengono eseguite ci troveremo con:

$$N - i + 1 \text{volte che l'operazione viene eseguita.}$$

- 7) Costo = 1 (fuori dal ciclo)

Dopo che viene effettuata l'analisi, possiamo andare a sommare tutti i risultati che abbiamo ottenuto in termini di unita (correggere accento) di tempo.
La funzione $T(n)$ e(correggere) la funzione che ci tiene traccia della complessita dell'algoritmo.

Andiamo semplicemente a sommare i nostri risultati di ogni riga:

$$T(n) = 1 + 2 * (N) + 2 * (N^2 + N) + 3 * \frac{N(N+1)}{2}$$

Questo risultato e ottenuto semplificando le nostre sommatorie:

- 3) $2 * \sum_{i=1}^{N+1} 1 = 2 * (N+1)$
- 4) $2 * \sum_{i=1}^{N} \sum_{i=1}^{N+1} 1 = 2 * \sum_{i=1}^{N} N + 1 = 2 * (N^2 + N)$
- 5) $3 * \sum_{i=1}^{N} \sum_{i=1}^{N} 1 = 3 * \sum_{i=1}^{N} N = 3N^2$
- 6) $\sum_{i=1}^{N}(N-i+1) = N-(k-1)$ cioe ad ogni ciclo il numero delle volte che viene eseguita questa operazione diminuisce costantemente di 1 (cioe dipendente dal salire di i).

$$T(n) = \frac{13}{2}N^2 = \frac{9}{2}N + 4$$

Come vediamo questa funzione e quadratica, quindi cresce esponenzialmente nel tempo, molto pesante e lenta come funzione.

1.1.1 Algoritmo di conteggio v2

Dopo aver ottenuto i risultati dell'analisi sopra, possiamo dire che e sicuramente possibile semplificare il nostro codice in modo tale da far eseguire meno operazioni al nostro processore e quindi utilizzare meno tempo.

```
1 Conta(N):
2   ris = 0;
3   for i=1 to N do
4     ris = ris + (N-i+1)
5   return ris
```

Cosi facendo abbiamo semplicemente detto al nostro codice che deve sommare soltanto gli elementi che nel momento in cui i e fissato, sono $\leq$ di se stesso.
Cosi facendo si dovrebbero eliminare molte operazion inutili, analizziamo.

- 2)sempre 1 operazione
- 3) $2 * \sum_{i=1}^{N+1} 1$
- 4) $\sum_{i=1}^{N} 7 = 7 \cdot N$

Come possiamo osservare abbiamo eliminato il secondo for, dunque abbiamo eliminato la quadraticita, ora l'operazionea riga 4 viene eseguita solamente N volte, e il numero di operazioni semplici che esegue e fissato.

$$T(n) = 1 + 2(N+1) + 7N + 1 = 9N + 4$$

1.1.2 Algoritmo di conteggio v3

Da come possiamo notare e possibile di nuovo semplificare l'ultima sommatoria della riga 4 dello scorso algoritmo.

Da

$$\sum_{i=1}^{N} N - i + 1 \rightarrow \sum_{i=1}^{N} i$$

Il risultato della sommatoria e lo stesso, se andiamo a semplificarlo.

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

```
1 Conta(N):
2   ris = 0
3   ris = ris+(N-i+1)
4   return ris
```

In questo modo abbiamo eliminato qualsiasi ciclo e quindi il risultato sara un numero fisso di operazioni.

- 1) 1 operazione
- 2) 5 operazioni elementari

$$T(n) = 5 + 1 = 6$$

In questo caso la funzione tempo per eseguire queste operazioni e costante, non dipendente da nessun N, dunque e la migliore soluzione possibile per questo algoritmo.

1.2 Algoritmo della massima sottosequenza contigua

Preso un array di n elementi, vorremmo provare a trovare la sottosequenza la cui somma di tutti i valori e massima.

Come fare? La soluzione piu naive possibile e effettuare 3 cicli for innestati:

```
1 int Max_seq_sum_1(int N, array a[])
2   maxsum = 0
3     for i=1 to N
4       for j=i to N
5         sum = 0
6         for k=i to j
```

```
        sum = sum + a[k]
      maxsum = max(maxsum,sum)
  return maxsum
```

Come possiamo notare l'avere 3 for innestati ci fa avere una complessita computazionale di N^3, comunemente rappresentato dalla formula della notazione asintotica $O(N^3)$.

1.2.1 Algoritmo v2

Come e facile notare, nel terzo for innestato c'e una ripetizione abbastanza inutile di operazioni che effettuiamo per andarci a sommare i valori delle sottosequenze. In particolare andiamo a ripetere scorrere piu volte lo stesso numero di celle, solamente per trovare la sottosequenza poco piu grande (a volte anche di una cella).

Gli stessi valori delle sottosequenze possono essere semplicemente trovati scorrendo avanti l'indice e sommando alla sequenza precedente il valore successivo. Analiticamente ci troveremmo che:

$$\sum_{k=i}^{j+1} a_k = a_{j+1} + \sum_{k=i}^{j} a_k$$

Il valore sum rimarra inalterato, ma verra solamente aggiornato del valore successivo.
Il codice si presentera in questo modo:

```
int Max_seq_sum_1(int N, array a[])
  maxsum = 0
    for i=1 to N
      sum = 0
      for j=i to N
        sum = sum + a[j]
        maxsum = max(maxsum,sum)
  return maxsum
```

2 Lezione 03 - 21-09-2023

2.1 Algoritmo v3

L'algoritmo può essere anche migliorato, riusciendo ad arrivare ad una complessità **lineare**, nel seguente modo:

```
int Max_seq_sum_3(int N, array a[])
  maxsum = 0
  sum = 0
  for j=1 to N
    if (sum + a[j] > 0) then
      sum = sum + a[j]
    else
      sum = 0
    maxsum = max(maxsum,sum)
  return maxsum
```

Il ragionamento è il seguente: Se prendiamo un insieme di numeri da sommare, (da i ad a), possiamo controllare se esso è positivo o negativo. Nel caso in cui $\sum_{e=i}^{a} A[e]$ risultasse positiva, andiamo a espandere il nostro range fintantochè il risultato della sommatoria riamanga positivo. Nel caso in cui invece il risultato fosse negativo, non ci conviene tenere traccia dei numeri più piccoli di quel range, dato che se quella sommatoria è minore del numero successivo alla sommatoria, non ha senso tenerne conto. E quindi invece ha senso tenere traccia del numero successivo. Da quel numero poi sommare i numeri successivi continuando il processo sopracitato.

2.2 Strutture Dati - Insieme Dinamico

Vediamo come rappresentare un insieme di dati dinamico S (con insieme dinamico si intende una collezione di elementi variabile nel tempo, quindi è possibile aggiungere o rimuovere elementi);

$$S = \{a_1, a_2, ..., a_n\} \quad n \geq 0$$

Andiamo a definire alcune operazioni:

- Insert$(S, a) \rightarrow S'$ $(S' = S \cup \{x\})$
- Deletes$(S, a) \rightarrow S'$ $(S' = S \backslash \{x\})$
- Search$(S, a) \rightarrow \{True, False\}$
- Massimo$(S) \rightarrow a$
- Minimo$(S) \rightarrow a$
- Successore$(S, a) \rightarrow a'$
- Predecessore$(S, a) \rightarrow a'$

3 Lezione 04 - 23/09/2023

3.1 Array non ordinato

Abbiamo un insieme S che vogliamo rappresentare in A.

$$|S| = |A|$$

La funzione $A.free$ ci restituirà la posizione della prima cella libera.

3.1.1 Ricerca

Dato un array e un elemento da cercare, restituisce la posizione in cui è presente il valore

```
Search(A,e) //A: array in input, e: elemento da cercare
  pos = A.free-1
  while A[pos] != e and pos >= 0 do
    pos = pos-1
  return pos
```

Ci posizionamo all'ultima cella piena e mano a mano tornando all'indietro andiamo a cercare il valore che abbiamo in input, se non è presente nell'array andiamo a restistuire il valore -1. La complessità computazione sarà **lineare** nello specifico:

$$T_s(n) = c \cdot n$$

Dove c è il costo fisso delle operazione e n è quante volte si ripete il ciclo.

3.1.2 Inserimento

Dato un array e un elemento da inserire, andiamo a verificare tramite la funzione **ricerca** se l'elemento non sia già presente poiché non vogliamo elementi duplicati.

```
Inserimento(A,e) //A: array in input, e: elemento da cercare
  pos = search(A,e) //-1: non trovato, >=0: indice del valore
  if pos = -1 then
    if(A.free < length(A)) then
      A=resize(A)
    A[A.free]=e
    A.free=A.free+1
```

La complessità computazione sarà **lineare** nello specifico:

$$T_i(n) = 2c \cdot n + c'$$

3.1.3 Cancellazione

Dato un array e un elemento da cancellare, andiamo a verificare tramite la funzione **ricerca** se l'elemento sia presente in modo da portelo eliminare.

```
Delete(A,e) //A: array in input, e: elemento da cercare
  pos = search(A,e) //-1: non trovato, >=0: indice del valore
  if pos >= 0 then //Elemento trovato
    A[pos]=A[A.free-1]
    A.free=A.free-1
```

3.2 Array Ordinato

Uno dei grosso problemi dell'array visto in precendenza era la ricerca, poiché al più avevo costo **lineare** cioè la lunghezza di tutto l'array, con l'array ordinario andiamo a sopperire a questo problema ma ovviamente aggiungendone degli altri.

3.2.1 Ricerca Binaria

La ricerca binaria è un algoritmo applicabile solo ad array ordinato che permette una ricerca molto rapida.
L'algoritmo è simile al metodo usato per poter trovare una parola sul dizionario: sapendo che il vocabolario è ordinato alfabeticamente, l'idea è quella di iniziare la ricerca non dal primo elemento, ma da quello centrale, cioè a metà del dizionario. Si confronta questo elemento con quello cercato:

- se corrisponde, la ricerca termina indicando che l'elemento è stato trovato;
- se è superiore, la ricerca viene ripetuta sugli elementi precedenti (ovvero sulla prima metà del dizionario), scartando quelli successivi;
- se invece è inferiore, la ricerca viene ripetuta sugli elementi successivi (ovvero sulla seconda metà del dizionario), scartando quelli precedenti.

Andiamo a definirlo per ricorsione:

```
BinSearch(A, e, i, j) //i: punto inizio, j: punto fine
  if i <= j then
    q=(i+j)/2 //punto centrale
    if A[q] > e then
      i=BinSearch(A, e, i, q-1) //ricerca a "destra"
    else if A[q] < e then
      i=BinSearch(A, e, q+1, j) //ricerca a "sinistra"
    else return q //trovato
  else return -1 //non trovato
```

Per studiare la complessità di questo algoritmo (ricorsivo), andiamo a rappresentare le varie chiamate tramite un "albero degenere" in cui ogni nodo ha un solo figlio e così via, il costo di ogni nodo sarà c, ad ogni chiamatà andiamo a dimezzare la lunghezza

dell'array $n...n/2...n/4....$ il massimo delle chiamate possono essere $h+1$ dove h è l'altezza dell'albero andiamo a generallizare con:

$$\frac{n}{2^h} \qquad h : \text{è il numero dei richiami}$$

Dobbiamo trovare il valore di h per il quale l'intervallo si riduce a 1 elemento, in quanto l'elemento desiderato sarà stato trovato. Quindi, dobbiamo risolvere l'equazione:

$$\frac{n}{2^h} = 1$$

Per risolvere questa equazione per h, possiamo moltiplicare entrambi i lati per 2^h:

$$n = 2^h$$

Ora, per isolare h, possiamo applicare il logaritmo in base 2 ad entrambi i lati:

$$h = log_2(n)$$

Quindi, il numero di chiamate ricorsive necessarie per trovare l'elemento desiderato è logaritmico rispetto alla dimensione dell'array n. Pertanto, la complessità computazionale dell'algoritmo di ricerca binaria è $O(logn)$ nel caso peggiore.
N.B Una complessita computazionale logaritimica è più efficente di una lineare.

3.2.2 Inserimento/Cancellazione nella ricerca binaria

Abbiamo visto come in un array ordinato la ricerca (binario) è molto efficemente ma abbiamo come contro che l'inserimento/cancellazione hanno costo peggiore poiché dobbiamo mantenere l'ordinamento, quindi se vogliamo inserire un elemento dobbiamo andare a spostare i valpri per creare un posto disponibile.