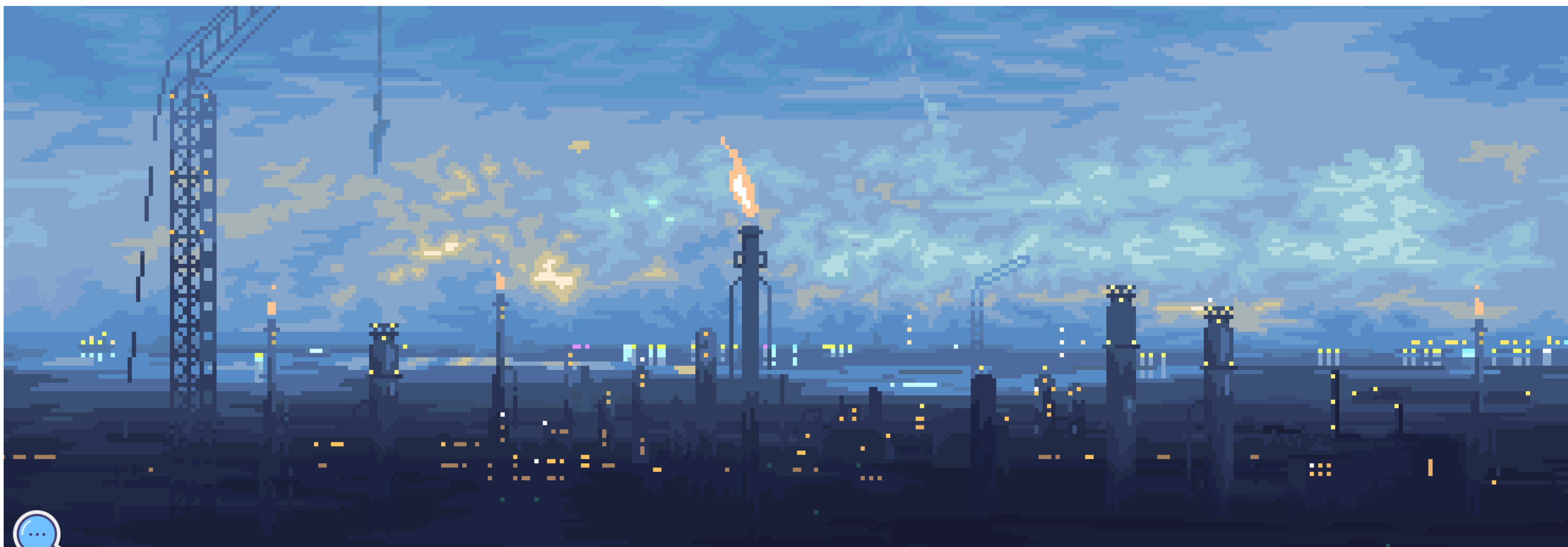

Il problema della ricerca

INDICE

4 Il problema della ricerca

4.1 Ricerca sequenziale

4.2 Stima del costo medio

4.2.1 Operatore *in del linguaggio Python*

4.3 Ricerca binaria

Esercizi

4 Il problema della ricerca

Nell'informatica troviamo problemi abbastanza ricorrenti, uno di questi è la ricerca di un elemento in un insieme di dati(ad es.numeri, cognomi, ecc…)

Questi problemi consistono in:

- Input: un array A ed un valore v da cercare al suo interno;
- Output: un indice i tale che $A[i] = v$, oppure $NULL(o-1)$ se il valore v non è presente nell'array.

4.1 Ricerca sequenziale

Un semplice algoritmo di ricerca è basato su:

- Ispezione: ossia controllare uno alla volta gli elementi dell'array;
- Confrontare: ossia confrontare ciascun elemento con v;
- Restituzione del risultato: interrompendosi appena(NON) trovato v.

Un esempio di ricerca sequenziale può essere questo indice:

```
def Ricerca_sequenziale(A,v):
  i = 0
  while((i<len(A)) and (A[i]!=v)):
    i+=1
  if i<len(A): return i
  else: return -1
```

Anche senza analizzare il codice possiamo vedere che il caso migliore è quando v si trova in $A[0]$ e quindi esce immediatamente dal ciclo, mentre il caso peggiore è il caso in cui v non si trova in A(poiché comunque dovrebbe venire analizzata l'intera lista per dire che l'elemento non c'è).

Dunque:

- Caso migliore: se $v = A[0]$, allora $\Theta(1)$;
- Caso peggiore: se $v \notin A$, allora $\Theta(n)$.

Visto che non abbiamo trovato una stima del costo che sia valida per tutti i casi, diremo che il costo computazionale dell'algoritmo è $\Theta(n)$.

4.2 Stima del costo medio

Quando il costo migliore e peggiore sono diversi, non è possibile determinare un valore stretto per il costo computazionale, possiamo però domandarci quale sia il costo computazionale dell'algoritmo nel caso medio.

Immaginiamo di avere un array A di n elementi al cui interno ogni posizione ha la stessa probabilità di contenere il valore v da cercare.

Possiamo dire che la probabilità che v sia in k-esima posizione è:

$$P = \frac{1}{n}$$

Applicando tale probabilità al numero totale di iterazioni, otteniamo:

$$P \cdot \sum_{k=0}^{n} k = \frac{1}{2} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Dunque possiamo dire che in media il ciclo viene eseguito $\frac{n+1}{2}$ volte, ossia $\Theta(n)$. Quindi il caso medio, si avvicina più al caso peggiore.

Il costo medio può essere trovato anche con il calcolo delle permutazioni.

Come ben sappiamo le permutazioni di una lista di n elementi corrisponde a $n!$, quindi possiamo dire che le permutazioni totali di A sono:

$$P_{tot} = n!$$

Dove al suo interno vi sono anche i seguenti sottoinsiemi:

- Permutazioni con v in prima posizione;
- Permutazioni con v in seconda posizione;
- Permutazioni con v in terza posizione;
- …

Tali sottoinsiemi, corrispondono ad una permutazione di $n-1$ elementi, ossia:

$$P_k = (n-1)!$$

quindi:

$$\sum_{k=0}^{n} k \cdot \frac{P_k}{P_{tot}} = \sum_{k=0}^{n} k \cdot \frac{(n-1)!}{n!} = \frac{1}{n} \sum_{k=0}^{n} k = \frac{n+1}{2}$$

Come possiamo vedere anche in questo verrà $\frac{n+1}{2}$ ossia $\Theta(n)$.

4.2.1 Operatore *in del linguaggio Python*

Prima di parlare dell'operatore *in* del linguaggio Python, bisogna ricordare la sua struttura:

$$< VALORE > IN < LISTA >$$

Ed un esempio del suo utilizzo può essere:

```
if v in A:
  print("Il valore v si trova in A")
else:
  print("Il valore v non si trova in A")
```

A prima vista essendo una condizione diremo che il costo è $\Theta(1)$, in realtà non è così, poiché esso corrisponde ad una ricerca sequenziale sappiamo benissimo che il costo sarà $\Theta(n)$.

4.3 Ricerca binaria

Quotidianamente non utilizziamo mai una ricerca di tipo sequenziale, ad esempio facciamo caso volessimo cercare una parola nel dizionario, ovviamente non leggeremmo mai l'intero dizionario fino a trovare quella parola.

Ciò che noi facciamo invece è aprire una pagina a caso e questo punto ci ritroviamo con 3 opzioni:

1. Troviamo immediatamente la parola che stiamo cercando nella pagina che abbiamo aperto;
2. La parola che cerchiamo si trova prima della pagina che abbiamo aperto dunque basterà solo scegliere un'altra pagina all'interno solo delle pagine precedenti;
3. La parola che cerchiamo si trova dopo la pagina che abbiamo aperto dunque basterà solo scegliere un'altra pagina all'interno solo delle pagine successive;

Quello che facciamo noi è un'altra tipologia di algoritmo di ricerca, chiamata Ricerca binaria, dove il procedimento sopra elencato viene ripetuto fino a quando non troviamo la parola che stiamo cercando.

Quindi nell'algoritmo di Ricerca binaria:

1. Viene ispezionato l'elemento centrale(che chiameremo m):
   a. Se corrisponde al valore v che stiamo cercando$(v = m)$, verrà restituito l'indice della posizione ritrovata;
   b. Se $v < m$, allora v si troverà nella metà inferiore della lista, quindi l'algoritmo verrà ripetuto solo su quella parte;
   c. Se $m < v$, allora v si troverà nella metà superiore della lista, quindi l'algoritmo verrà ripetuto solo su quella parte;
2. Ripetere il passaggio finché la lista non si sarà ridotta ad un solo elemento:
   a. Se l'unico elemento rimasto corrisponde a v, verrà restituito l'indice;
   b. Se non viene ritrovato verrà restituito -1, che va ad indicare per l'appunto che l'elemento non è nella lista;

📎 ***ESEMPIO RICERCA BINARIA:***

Ad esempio facciamo finta di star cercando 20 in questa lista:

2	5	8	12	17	20	40	51	63	91

Andiamo a prendere l'elemento centrale ossia 17, noteremo subito che $17 < 20$ quindi da 17 in giù possiamo eliminare la lista, e la lista diventerà:

20	40	51	63	91

Riprendiamo l'elemento centrale ossia 51 vedremo subito che $51 > 20$ quindi possiamo eliminare da 56 in poi:

20	40

Prendendo l'elemento centrale troveremo 20 e ritornerà il suo indice ossia 5.

📖 ***NOTA:***

Per applicare l'algoritmo di ricerca binaria l'array dev'essere ***obbligatoriamente ordinato in modo crescente***, altrimenti non può essere utilizzato.

Vediamo ora l'implementazione in codice di tale algoritmo:

```
def Ricerca_binaria(A,v):
 a=0 #Primo indice di A
 b=len(A)-1 #Ultimo indice di A
 m =(a+b)//2 #Indice a metà di A
 while (A[m]!=v):
  if (A[m] > v):
   b=m – 1 #Prendo la metà inferiore
  else:
   a=m + 1 #Prendo la metà superiore
  if a > b: #Se v non è in A
   return -1
  m=(a+b)//2 #Calcolo nuovamente il valore di m
return m
```

Andiamo ora a calcolare il costo di tale algoritmo, partendo dallo studio del ciclo while:

n.iterazione	1	2	3	...	k
Lunghezza A	$\frac{n}{2}$	$\frac{n}{2^2}$	$\frac{n}{2^3}$	...	$\frac{n}{2^k}$

La condizione necessaria per far terminare il ciclo è $\frac{n}{2^k} = 1$ che diventa:

$$\frac{n}{2^k} = 1 \rightarrow k = log_2(n)$$

Andando a calcolare il costo, ossia:

$$T(n) = \Theta(1) + log(n)(\Theta(1)) + \Theta(1) = 2\Theta(1) + \Theta(log(n)) = \Theta(log(n))$$

noteremo che sarà il caso peggiore, mentre il caso migliore è quando troviamo immediatamente v in A, in quel caso il costo sarà $\Theta(1)$.

Visto che abbiamo un caso peggiore e migliore discordanti andiamoci a calcolare il caso medio:

Analisi delle posizioni raggiungibili

1. Alla prima iterazione dell'algoritmo, le posizioni raggiungibili sono solo una, ossia quella centrale.
2. Alla seconda iterazione, le posizioni raggiungibili sono 2:
   - Quella al centro della metà inferiore
   - Quella al centro della metà superiore

3. Alla terza iterazione, le posizioni raggiungibili sono 4:

- Quella al centro della metà inferiore della prima metà inferiore
- Quella al centro della metà superiore della prima metà inferiore
- Quella al centro della metà inferiore della prima metà superiore
- Quella al centro della metà superiore della prima metà superiore

4. ecc...

Dunque, concludiamo che le posizioni raggiungibili da ogni k-esima iterazione sono:

$$n(k) = 2^{k-1}$$

Quindi P_k sarà uguale a 2^{k-1} mentre P_{tot} sarà uguale a n:

$$\sum_{k=0}^{log(n)} k \cdot \frac{2^{k-1}}{n} = \frac{1}{n} \cdot \sum_{k=0}^{log(n)} k \cdot 2^{k-1} = \frac{1}{n} \cdot \sum_{k=0}^{log(n)} log(n) \cdot 2^{log(n)-1} = \frac{log(n) \cdot 2^{log(n)-1}}{n} = log(n) - 1 + \frac{1}{n} = \Theta(log(n))$$

Esercizi

Sia dato un array A di interi e due valori a e b con $a \leq b$, il problema è quello di sapere quanti elementi di A sono compresi nell'intervallo chiuso $[a, b]$.

- Si progetti un algoritmo per risolvere tale problema su qualsiasi array A;
- Si progetti un algoritmo più efficiente del precedente assumendo che A sia già ordinato e che contenga solo valori distinti.

Per ciascun algoritmo si descriva a parole l'idea, si scriva lo pseudocodice e si analizzi il tempo di esecuzione asintotica.

🧮 ***Si progetti un algoritmo per risolvere tale problema su qualsiasi array A:***

```
def Conta_in_Intervallo(A, a, b):
 lenght, count = len(A), 0;         #Θ(1)
 for x in range(lenght):
  if a <= A[x] and A[x] <= b:        #Θ(1)
    count += 1;                       #Θ(1)
 return count                       #Θ(1)
```

Studiando il ciclo for vedremo che verrà ripetuto n volte, quindi il costo dell'algoritmo sarà:

$$T(n) = \Theta(1) + n(\Theta(1)) + \Theta(1) = 2\Theta(1) + \Theta(n) = \Theta(n)$$

🧮 ***Si progetti un algoritmo più efficiente del precedente assumendo che A sia già ordinato e che contenga solo valori distinti:***

```
def count_elements_in_range(A, a, b):
    left = 0
    right = len(A) - 1
    left_idx = binary_search_left(A, a,left,right)
    right_idx = binary_search_right(A, b,left,right)
    return right_idx - left_idx + 1

def binary_search_left(A, target,left,right):
    while left <= right:
        mid = (left + right) // 2
        if A[mid] < target: left = mid + 1
        else: right = mid - 1
    return left

def binary_search_right(A, target,left,right):
    while left <= right:
        mid = (left + right) // 2
        if A[mid] <= target: left = mid + 1
        else: right = mid - 1
    return right
```

In questo caso invece il ciclo si comporterà come l'esempio visto in precedenza e varrà $\Theta(log(n))$.