

Algoritmi e strutture dati

Matteo Mazzaretto

2024

Indice

1	Complessità problema	1
2	Cos'è un heap, complessità ricerca massimo	1
3	Metodo del limite e dimostrazione	1
4	Definizione albero binario, BST, complessità ricerca	1
5	Dimostrazione che il limite degli ordinamenti è $n \log n$	2
6	Complessità quicksort e spiegazione breve algoritmo, caso medio, peggiore e perché tante ripartizioni sono caso medio	2
7	Counting sort, perché non si può usare sempre per ottenere ordinamento lineare, condizione affinché $\Theta(n)$, Radix Sort	3
8	Tabella hash: complessità inserimento e rimozione, chaining e open addressing	4
9	Doppio hashing (come scegliere le funzioni)	4
10	Max-heap e heapsort	4
11	Funzionamento e complessità maxHeapify e buildMaxHeap, complessità inserimento	5
12	Cos'è heap, complessità insert in heap	6
13	Definizione di RB-Tree e costo inserimento	7
13.1	Definizione funzioni	8

14 Perché preferire BST a tabelle hash	11
15 Definizione di sottostringa e sottosequenza e numero di sottostringe e sottosequenze in una stringa	11
16 Esistenza eventuale altro ordine per calcolare elementi nella terza condizione della dinamica	13
17 Generico algoritmo top down memoizzato	13
18 Codici di Huffman	13
19 Quale tra gli esercizi di programmazione dinamica svolti a lezione non si risolveva solo risolvendo sottoproblemi	13
20 Definizione e vantaggi memoizzazione, complessità e funzionamento dell'inserimento in un min heap	13
21 Codice prefisso	13
22 Confronto tra varie funzioni, algoritmo di dinamica del compito memoizzato, vantaggio algoritmi memoizzati rispetto agli iterativi	13
23 Scansione riempimento della tabella di LCS	13
24 Differenza tra approccio top-down e bottom-up per programmazione dinamica. Vantaggi e svantaggi di entrambi	13
25 Complessità sottostringa, complessità sottosequenza, complessità LCS	13

1 Complessità problema

Dato un problema P, la complessità di P è la complessità del più efficiente algoritmo che lo risolve.

Per complessità di algoritmo si intende il suo limite stretto, ovvero quando limite inferiore (Ω) e il suo limite superiore (O) coincidono

2 Cos'è un heap, complessità ricerca massimo

L'heap è un albero ordinato binario quasi completo implementato come array, in cui $\forall i$ il suo figlio sinistro ha indice $2*i$ e il suo figlio destro $2*i+1$. La radice dell'heap è l'elemento con indice 1 nell'array e il genitore di ogni nodo si trova troncando il risultato di $i/2$.

$\forall i \geq n/2$ (dove n è la dimensione dell'array) è una foglia, ovvero non ha nè figli sinistri nè figli destri.

Se si parla di max-heap, la complessità di ricerca del massimo è $O(1)$ poiché il massimo è sempre il primo elemento dell'array. Il max-heap ha proprietà che ogni nodo è \geq discendenti e \leq antenati

3 Metodo del limite e dimostrazione

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = k > 0 \rightarrow f(n) = \Theta(g(n))$$

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0 \rightarrow f(n) = O(g(n))$$

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = \infty \rightarrow f(n) = \Omega(g(n))$$

Foto dimostrazione

4 Definizione albero binario, BST, complessità ricerca

Per albero binario si intende una struttura dati in cui ogni nodo non foglia ha uno o due figli, e una foglia non ha nodi figli.

Per albero binario di ricerca si intende un albero binario ordinato tale \forall nodo x

- 1) \forall nodo y sottoalbero sx $y.key \leq x.key$
- 2) \forall nodo y sottoalbero dx $y.key \geq x.key$

Questa è una proprietà globale dell'albero, se fosse solo locale (come negli

heap) l'albero non sarebbe più di ricerca

La complessità della ricerca è $O(h)$, dove $h=\log n$ se albero bilanciato

5 Dimostrazione che il limite degli ordinamenti è $n \log n$

Il limite degli ordinamenti sfrutta l'albero di decisione, il quale è un albero che descrive in maniera astratta l'esecuzione di algoritmi su input di dimensione n in cui ogni foglia rappresenta una e una sola permutazione dell'input, e ogni input è rappresentato da almeno una foglia. Se ciò non avviene vuol dire che l'algoritmo non è corretto.

Bisogna comunque dire che $\Omega(n \log n)$ si applica solo agli algoritmi che usano confronto tra elementi

Foto dimostrazione.

6 Complessità quicksort e spiegazione breve algoritmo, caso medio, peggiore e perché tante ripartizioni sono caso medio

Il quicksort è un algoritmo del tipo divide et impera, il quale, prendendo un pivot, porta alla sinistra dell'array gli elementi \leq e alla destra gli elementi \geq tramite la funzione Partition (divide), e successivamente esegue ricorsivamente il quicksort sulla parte sinistra e parte destra della partizione (impera)

```
1 QuickSort(A,p,r)
2 if p<r
3     q=Partition(A,p,r)
4     QuickSort(A,p,q-1)
5     QuickSort(A,p,q+1)
```

La sua complessità nel caso peggiore è del tipo $O(n^2)$ perché il caso peggiore avviene quando l'array è già ordinato e si avrebbe un tempo di esecuzione $T(n)=T(n-1)+\Omega(n)+T(0)$ poiché $T(n)=\sum_{i=1}^n(a(n-i)+b)$

La sua complessità nel caso medio è del tipo $O(n \log n)$

Il partizionamento proporzionale o sbilanciato, studiandolo tramite gli alberi di ricorsione, porta a dire che l'altezza dell'albero è di tipo logaritmica con una forma del tipo $T(n)=\Omega(n)+T(k)+T(n-k-1)$ che porta ad affermare che $T(n) \leq cn \log n$ (base del partizionamento) $\rightarrow T(n)=O(n \log n)$

7 Counting sort, perché non si può usare sempre per ottenere ordinamento lineare, condizione affinché $\Theta(n)$, Radix Sort

Il counting sort è un ottimo algoritmo di ordinamento in tempo lineare ma richiede delle ipotesi che restringano l'input, ovvero che sia un array di interi compresi fra 0 e $k > 0$

$$\begin{cases} \text{Input } A[1\dots n] \text{ con } A[j] \in [0,1,\dots,k] \\ \text{Output } B[1..n] \text{ copia di } A \text{ ordinata} \end{cases}$$

```
1 CountingSort(A,B,k)
2
3 C[0...k] riempito di tutti 0
4 for j=1 to A.length
5     C[A[j]]++
6 for i=1 to k
7     C[i]=C[i]+C[i-1]
8 for j=A.length down to 1
9     B[C[A[j]]]=A[j]
10    C[A[j]]--
```

Inoltre, se $k=O(n) \rightarrow$ costo $O(n)$ in quanto il costo totale è $\Theta(n+k)$ Oltretutto ha il contro di essere un algoritmo instabile, ovvero che, se sono presente delle ripetizioni, non è detto che esse mantengano l'ordine iniziale nell'array ordinato

RadixSort è un tipo di algoritmo di ordinamento lineare che ordina, dato d =numero cifre significative, i numeri presenti nell'array per cifra significativa con un algoritmo stabile. Nella modalità MDM ordina dalla cifra più significativa, nella modalità LSD ordina dalla cifra meno significativa

Per il RadixSort si usa il Counting Sort, quindi ogni iterazione ha $\Theta(n+b)$ e con d =numero iterazioni si ha $\Theta((n+b)d)$

$b=O(1)/O(n) \rightarrow \Theta(nd)$

se $d=1 \rightarrow \Theta(n)$

Spazio: $\Theta(n+b)$

```
1 RadixSort(A,d)
2 for j=1 to d
3     ordina A rispetto alla cifra j-esima
4     con algoritmo stabile
```

8 Tabella hash: complessità inserimento e rimozione, chaining e open addressing

Le tabelle hash sono molto efficienti, in quanto hanno un costo medio di $\Theta(1)$ e un costo peggiore di $\Theta(n)$

L'idea delle tabelle hash è usare uno spazio proporzionale al numero di elementi nella struttura

Con le funzioni di hash ci possono essere più risultati uguali poiché gli hashing non sono iniettive

L'inserimento e la rimozione hanno la grandissima qualità di essere entrambe $O(1)$

Esistono due modi di costruire una tabella hash: chaining e open addressing

Con il chaining se $h(x1.key)=h(x2.key)$ allora la cella in cui entrambi finiranno viene implementata come una lista e viene considerato un fattore di carico $\alpha=\frac{n}{m}$ con n =celle tabella e m =elementi memorizzati. Bisogna però mettere in evidenza la distribuzione degli input e la qualità della funzione hash, la quale idealmente dovrebbe avere probabilità di assegnare ogni elemento in input con probabilità $1/m$

Nell'open addressing tutti gli elementi dell'insieme dinamico vengono memorizzati nella tabella

9 Doppio hashing (come scegliere le funzioni)

Date $h1(k)$ e $h2(k)$ hash unarie $h(k,i)=(h1(k)+i(h2(k)))$ la condizione è che $h2(k)$ e m (celle tabella) siano coprimi

Ma come fare in modo che $h(k,0)...h(k,m-1)$ è permutazione?

$$(h1(k)+ih2(k) \bmod m) = (h1(k) + i'h2(k) \bmod m)$$

$$(h1(k)+ih2(k) \bmod m) - (h1(k) + i'h2(k) \bmod m) = 0$$

$$(h1(k)+ih2(k)-h1(k)-i'h2(k)) \bmod m = 0 \rightarrow (ih2(k)-i'h2(k)) \bmod m = 0$$

$$m \text{ divide } (i-i') \text{ e } 0 \leq i-i' \leq (m-1) \rightarrow i-i' = 0 \rightarrow i=i'$$

$$m \text{ primo } h2(k) < m \rightarrow h2(k)=1+k \bmod m', m' < m$$

10 Max-heap e heapsort

Il max-heap è un albero binario ordinato quasi completo costruito come array, il quale ha le proprietà degli heap (figlio destro, sinistro e parent) in cui l'elemento massimo è il primo elemento dell'array, ogni nodo è \leq antenati e \geq discendenti

L'heap sort è un algoritmo di ordinamento che sfrutta l'array costruito come

heap per ordinarlo e ha complessità $O(n \log n)$

```
1 HeapSort(A)
2
3 BuildMaxHeap(A)
4 for i=A.length down to 2
5     A[1] $\text{iff}$ A[i]
6     A.size=A.size-1
7     MaxHeapify(A,1)
```

Invariante è che $A[1...i]$ sia MaxHeap e che $A[i+1...n]$ sia ordinato

Inizio: $A[1...n]$ è maxHeap

Iterazione: $A[1...i]$ è MaxHeap perché MaxHeapify su 1 funziona, il più grande va in fondo quindi $A[i+1...n]$ ordinato

Fine: $A[1]$ MaxHeap, $A[2..n]$ ordinato

11 Funzionamento e complessità maxHeapify e buildMaxHeap, complessità inserimento

MaxHeapify(A,i) assume che i due sottoalberi di $A[i]$ siano MaxHeap e costruisce un MaxHeap in $A[i...n]$ col seguente algoritmo:

```
1 MaxHeapify(A,i)
2
3 l=left(i) r=right(i)
4 if (l<=A.size) and (A[l]>A[i])
5     max=l
6 else
7     max=i
8 if (r<=A.size) and (A[r]>A[max])
9     max=r
10 if max!=i
11     swap(A,i,max)
12     MaxHeapify(A,max)
```

Esso ha una complessità di $O(\log n)$ Invece BuildMaxHeap è una funzione per costruire un MaxHeap partendo dall'array A, e sapendo che ogni foglia è $\geq (n/2)$ si costruisce il seguente algoritmo:

```

1 BuildMaxHeap(A)
2
3 for i=(A.length)/2 down to 1
4     MaxHeapify(A,i)

```

il quale costruisce continuamente MaxHeap partendo dal presupposto che una foglia è già MaxHeap

Ha una complessità di $O(n \log n)$ perché esegue $n/2$ volte un algoritmo di complessità $O(\log n)$

Ma, volendo essere più precisi, si arriva ad una complessità di $O(n)$.

Foto dimostrazione.

12 Cos'è heap, complessità insert in heap

Spiegazione heap fatta sopra.

L'insert ha complessità di tipo $O(\log n)$ e l'algoritmo è costruito così:

```

1 Insert(A,k)
2
3 A.size=A.size+1
4 A[A.size]=k
5 MaxHeapifyUp(A,A.size)

```

Questo algoritmo sfrutta MaxHeapifyUp che ha la proprietà di costruire un MaxHeap di $A[i..n]$ partendo da un nodo "basso", forse maggiore di altri nodi (quindi \geq antenati) che rovina le proprietà di MaxHeap ripristinandole con questo algoritmo:

```

1 MaxHeapifyUp(A,i)
2
3 if (i>1) and (A[i]>A[parent(i)])
4     swap(A,i,parent(i))
5     MaxHeapifyUp(A,parent(i))

```


13 Definizione di RB-Tree e costo inserimento

I Red-Black trees sono BST in cui i nodi hanno i soliti attributi assieme all'attributo color, il quale può essere Red/Black

T.nil è un nodo con tutti gli attributi il cui colore è Black e si considera come una foglia (escluso per il T.nil sopra radice)

Queste sono le sue proprietà:

- 1) ogni nodo ha uno e un solo colore
- 2) la radice è Black
- 3) le foglie (T.nil) sono Black
- 4) i figli di un nodo Red sono Black
- 5) \forall nodo x, \forall cammino x \rightarrow foglia ha lo stesso numero di nodi Black. Vale per tutti i nodi \iff vale per la radice

Inoltre:

- 1) se elimino i nodi Red ogni cammino radice \rightarrow foglia avrà la stessa lunghezza come un albero completo
- 2) in ogni cammino i nodi rossi sono al più la metà

$$h \leq 2 \log_2(n+1)$$

$n(x)$ = numero nodi interni in Tx, $bh(x)$ = numero nodi Black incontrati

Inoltre $n(x) \geq 2^{bh(x)} - 1$ Il costo dell'inserimento è incredibilmente $O(h)$, dove $h = \log n$, ma è fattibile \iff il nodo da inserire ha colore RED

13.1 Definizione funzioni

La funzione Insert è definita così:

```
1 RB-Insert(T,z)
2
3   Insert(T,z)
4   z.color=RED
5   RB-InsertFixUp(T,z)
```

Con RB-InsertFixUp definita così:

```
1 RB-InsertFixUp(T,z)
2
3 while(z.p.color==RED)
4   if(z.p==z.p.p.left)
5     y=z.p.right
6     if(y.color==RED)           //CASO 1
7       y.color=BLACK
8       z.p.color=BLACK
9       z.p.p.color=RED
10      z=z.p.p
11   else                         //CASO 2
12     if(z==z.p.right)          //CASO 2.1
13       Left(T,z.p)
14       z=z.left
15       z.p.color=BLACK
16       z.p.p.color=RED
17       Right(T,z.p.p)
18   else                         //CASO 2.2
19     Right(T,z.p)
20     z=z.right
21     z.p.color=BLACK
22     z.p.p.color=RED
23     Left(T,z.p.p)
24   endif
25 endif
26 endif
27 endwhile
28 T.root.color=BLACK
```

La funzione RB-Delete è definita così

```
1 RB-Delete(T, z)
2
3 y = z
4 originalcolor = y.color
5 if (z.left == T.nil)
6     x = z.right
7     Transplant(T, z, z.right)
8 elif (z.right == T.nil)
9     x = z.left
10    Transplant(T, z, z.left)
11 else
12     y = Min(z.right)
13     originalcolor = y.color
14     x = y.right
15     if(y!=z.right)
16         TransPlant(T,y,y.right)
17         y.right=z.right
18         y.right.p=y
19     else
20         x.p=y
21         Transplant(T,z,y)
22         y.left=z.left
23         y.left.p=y
24         y.color=z.color
25     if(originalcolor==BLACK)
26         RB-DeleteFixUp(T,x)
```

RB-DeleteFixUp definita così:

```

1 RB-DeleteFixUp(T,x)
2 while (x != T.root and x.color == BLACK)
3     if (x == x.p.left)
4         w = x.p.right
5         if(w.color == RED)
6             w.color = BLACK
7             x.p.color = RED
8             Left(T, x.p)
9             w = x.p.right
10        if(w.left.color == BLACK and w.right.color == BLACK)
11            w.color = RED
12            x = x.p
13        elif (w.right.color == BLACK)
14            w.left.color = BLACK
15            w.color = RED
16            Right(T, w)
17            w = x.p.right
18        w.color = x.p.color
19        x.p.color = BLACK
20        w.right.color = BLACK
21        Left(T, x.p)
22        x=T.root
23    else
24        w = x.p.left
25        if(w.color == RED)
26            w.color = BLACK
27            x.p.color = RED
28            Right(T, x.p)
29            w = x.p.left
30        if(w.right.color == BLACK and w.left.color == BLACK)
31            w.color = RED
32            x = x.p
33        elif (w.left.color == BLACK)
34            w.right.color = BLACK
35            w.color = RED
36            Left(T, w)
37            w = x.p.left
38        w.color = x.p.color
39        x.p.color = BLACK
40        w.left.color = BLACK
41        Right(T, x.p)
42        x=T.root
43    endif
44 x.color=BLACK

```

14 Perché preferire BST a tabelle hash

In realtà non c'è esattamente una struttura preferibile in quanto tutte hanno sia dei pregi che dei difetti

Gli BST sono molto utili se è necessario un ordinamento all'interno della struttura, e oltretutto ha la proprietà di eseguire la maggior parte delle operazioni (se bilanciato) in tempo $O(\log n)$

Invece le tabelle hash sono utili per mettere in corrispondenza una data chiave con un dato valore.

Inoltre in una tabella di hashing ben dimensionata il costo medio di ricerca di ogni elemento è indipendente dal numero di elementi

Il loro problema è che nei casi peggiori hanno un caso spaziale di $O(n)$ ma come costo medio $O(1)$

15 Definizione di sottostringa e sottosequenza e numero di sottostringhe e sottosequenze in una stringa

Dato un alfabeto finito Σ , una stringa $X = \langle x_1, \dots, x_m \rangle$ $x_i \in \Sigma$ per $1 \leq i \leq m$ è una concatenazione finita di simboli in Σ

Lunghezza di X : $|X| = m$

Σ^* è l'insieme di tutte le stringhe di lunghezza finita costruibile su Σ

Stringa vuota: $\epsilon \in \Sigma^*$ per convenzione

Una sottostringa di X :

$X_{i \dots j} = \langle x_i, \dots, x_j \rangle$

$1 \leq i \leq j \leq m$

Quante sono le sottostringhe di una stringa?

$1 + m + \binom{m}{2}$ (scelta di 2 indici estremi) $= 1 + m + \frac{m(m-1)}{2} = \Theta(m^2)$

$= 1 + \sum_{i=1}^m \sum_{j=i}^m 1 = 1 + \sum_{i=1}^m (m - i + 1) = 1 + \frac{m(m+1)}{2} = \Theta(m^2)$ Una sottosequenza di X :

Z è sottosequenza di X se esiste una successione crescente di indici $1 \leq i_1 < i_2 < \dots < i_k \leq m$ tale che $z_j = x_{i_j}$, $1 \leq j \leq k$

Quante sono le sottosequenze di una stringa?

$\sum_{k=0}^m \binom{m}{k} = 2^m$

- 16 Esistenza eventuale altro ordine per calcolare elementi nella terza condizione della dinamica
- 17 Generico algoritmo top down memoizzato
- 18 Codici di Huffman
- 19 Quale tra gli esercizi di programmazione dinamica svolti a lezione non si risolveva solo risolvendo sottoproblemi
- 20 Definizione e vantaggi memoizzazione, complessità e funzionamento dell'ineserimento in un min heap
- 21 Codice prefisso
- 22 Confronto tra varie funzioni, algoritmo di dinamica del compito memoizzato, vantaggio algoritmi memoizzati rispetto agli iterativi
- 23 Scansione riempimento della tabella di LCS
- 24 Differenza tra approccio top-down e bottom-up per programmazione dinamica. Vantaggi e svantaggi di entrambi
- 25 Complessità sottostringa, complessità sottosequenza, complessità LCS