Algoritmi e strutture dati

Matteo Mazzaretto

2024

Indice

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 >= 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 >= discendenti e <= antenati

3 Metodo del limite e dimostrazione

$\lim_{n\to+\infty}\frac{f(n)}{g(n)}$=k>0 $\rightarrow$ f(n)=Θ(g(n))
$\lim_{n\to+\infty}\frac{f(n)}{g(n)}$=0 $\rightarrow$ f(n)=O((g(n))
$\lim_{n\to+\infty}\frac{f(n)}{g(n)}$=$\infty$ $\rightarrow$ f(n)=Ω(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 tc $\forall$ nodo x
1) $\forall$ nodo y sottoalbero sx y.key<=x.key
2) $\forall$ nodo y sottoalbero sx y.key>=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=logn se albero bilanciato

5 Dimostrazione che il limite degli ordinamenti è nlogn

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'agoritmo non è corretto.
Bisogna comunque dire che Ω(nlogn) 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 <= e alla destra gli elementi >= 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)+Ω(n)+T(0) poiché T(n)=$\sum_{i=1}^{n}(a(n-i)+b)$
La sua complessità nel caso medio è del tipo O(nlogn)
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)=Ω(n)+T(k)+T(n-k-1) che porta ad affermare che T(n)<=cnlogn (base del partizionamento) $\rightarrow$ T(n)=O(nlogn)

7 Counting sort, perché non si può usare sempre per ottenere ordinamento lineare, condizione affinché Θ(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

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

```
CountingSort(A,B,k)

C[0...k] riempito di tutti 0
for j=1 to A.length
   C[A[j]]++
for i=1 to k
   C[i]=C[i]+C[i-1]
for j=A.length down to 1
   B[C[A[j]]]=A[j]
   C[A[j]]--
```

Inoltre, se k=O(n)→costo O(n) in quanto il costo totale è Θ(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 Θ(n+b) e con d=numero iterazioni si ha Θ((n+b)d)
b=$O(1)/O$(n)→Θ(nd)
se d=1 → Θ(n)
Spazio: Θ(n+b)

```
RadixSort(A,d)
for j=1 to d
   ordina A rispetto alla cifra j-esima
   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) mod m) = (h1(k) + i'h2(k) mod m)
(h1(k)+ih2(k) mod m) - (h1(k) + i'h2(k) mod m) = 0
(h1(k)+ih2(k)-h1(k)-i'h2(k)) mod m = 0 $\rightarrow$ (ih2(k)-i'h2(k)) mod m = 0
m divide (i-i') e 0<=i-i'<= (m-1) $\rightarrow$ i-i' = 0 $\rightarrow$ i=i'
m primo h2(k) < m $\rightarrow$ h2(k)=1+k mod 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 <= antenati e >= discendenti
L'heap sort è un algoritmo di ordinamento che sfrutta l'array costruito come

heap per ordinarlo e ha complessità O(nlogn)

```
HeapSort(A)

BuildMaxHeap(A)
for i=A.length down to 2
   A[1]$\iff$A[i]
   A.size=A.size-1
   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 complessita 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:

```
MaxHeapify(A,i)

l=left(i) r=right(i)
if(l<=A.size) and (A[l]>A[i])
   max=l
else
   max=i
if(r<=A.size) and (A[r]>A[max])
   max=r
if max!=i
   swap(A,i,max)
   MaxHeapify(A,max)
```

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

```
BuildMaxHeap(A)

for i=(A.length)/2 down to 1
   MaxHeapify(A,i)
```

il quale costruisce continuamente MaxHeap partendo dal presupposto che una foglia è già MaxHeap
Ha una complessità di O(nlogn) perché esegue n/2 volte un algoritmo di complessità O(logn)
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(logn) e l'algoritmo è costruito così:

```
Insert(A,k)

A.size=A.size+1
A[A.size]=k
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 >= antenati) che rovina le proprietà di MaxHeap ripristinandole con questo algoritmo:

```
MaxHeapifyUp(A,i)

if (i>1) and (A[i]>A[parent(i)]
   swap(A,i,parent(i))
   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/Blach
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<=$2\log_2(n+1)$
n(x)=numero nodi interni in Tx, bh(x)=numero nodi Black incontrati
Inoltre n(x)>=$2^{bh(x)}$-1 Il costo dell'inserimento è incredibilmente O(h), dove h=logn, ma è fattibile $\iff$ il nodo da inserire ha colore RED

13.1 Definizione funzioni

La funzione Insert è definita così:

```
RB-Insert(T,z)

   Insert(T,z)
   z.color=RED
   RB-InsertFixUp(T,z)
```

Con RB-InsertFixUp definita così:

```
RB-InsertFixUp(T,z)

while(z.p.color==RED)
   if(z.p==z.p.p.left)
      y=z.p.right
      if(y.color==RED)          //CASO 1
         y.color=BLACK
         z.p.color=BLACK
         z.p.p.color=RED
         z=z.p.p
      else                      //CASO 2
         if(z==z.p.right)       //CASO 2.1
            Left(T,z.p)
            z=z.left
            z.p.color=BLACK
            z.p.p.color=RED
            Right(T,z.p.p)
         else                   //CASO 2.2
            Right(T,z.p)
            z=z.right
            z.p.color=BLACK
            z.p.p.color=RED
            Left(T,z.p.p)
         endif
      endif
   endif
endwhile
T.root.color=BLACK
```

La funzione RB-Delete è definita così

```
RB-Delete(T, z)

y = z
originalcolor = y.color
if (z.left == T.nil)
   x = z.right
   Transplant(T, z, z.right)
elif (z.right == T.nil)
   x = z.left
   Transplant(T, z, z.left)
else:
   y = Min(z.right)
   originalcolor = y.color
   x = y.right
   if (y.p != z)
      Transplant(T, y, y.right)
      y.right = z.right
      y.right.parent = y
   endif
   Transplant(T, z, y)
   y.left = z.left
   y.left.parent = y
   y.color = z.color
if (originalcolor == "BLACK")
   RB-DeleteFixUp(T, x)
```

RB-DeleteFixUP definita così:

```
RB-DeleteFixUp(T,x)
while (x != T.root and x.color == "DOUBLEBLACK")
   if (x == x.p.left:)
      w = x.p.right
      if(w.color == "RED")
         w.color = "BLACK"
         x.p.color = "RED"
         Left(T, x.p)
         w = x.p.right
      if(w.left.color == "BLACK" and w.right.color == "BLACK"
         )
         w.color = "RED"
         x = x.p
      else:
         if w.right.color == "BLACK":
         w.left.color = "BLACK"
         w.color = "RED"
         Right(T, w)
         w = x.p.right
      w.color = x.p.color
      x.p.color = "BLACK"
      w.right.color = "BLACK"
      Left(T, x.p)
      x = T.root
   else:
      w = x.p.left
      if w.color == "RED":
         w.color = "BLACK"
         x.p.color = "RED"
         Right(T, x.p)
      w = x.p.left
      if (w.right.color == "BLACK" and w.left.color == "BLACK
         ")
         w.color = "RED"
         x = x.p
      else:
         if w.left.color == "BLACK":
            w.right.color = "BLACK"
            w.color = "RED"
            Left(T, w)
            w = x.p.left
         w.color = x.p.color
         x.p.color = "BLACK"
         w.left.color = "BLACK"
         Right(T, x.p)
         x = T.root
endwhile
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 **Esistenza eventuale altro ordine per calcolare elementi nella terza condizione della dinamica**

16 **Definizione di sottostringa e sottosequenza e numero di sottostringe e sottosequenze in una stringa**

17 **Generico algoritmo top down memorizzato**

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**