Riassunto

martedì 7 giugno 2022 20:37

Istruzione	Tempo
For i = 0 to n	M
For i = 0 to m For j = 0 to m	m^2
For i = 0 to m For j = 0 to i	$\frac{n(n-1)}{2}$
For i = 0 to m For j = i to m	$\frac{n(n+1)}{2}$
While i < m	$\log_k m$
While i < m While j < m	$\log_k^2 m$
While i < m For j to m	$n * \log_2 m$

Limiti asintotici

- O -> è la o grande, equivale al limite superiore estretto
- o -> è la o piccola, equivale al limite superiore largo (una O è anche una o, ma non il contrario)
- Ω-> è la omega grande, cioè limite inferiore stretto
- ω-> è la omega piccola, cioè limite inferiore largo
- $O = \Omega \rightarrow \theta$ -> theta

Ricerche

- Ricerca sequenziale
  - Scorre tutto un vettore alla ricerca di un valore K
  - $T_m(n) = 1 \quad T_p(n) = n$
- Ricerca dicotomica
  - Dato un array ordinato e un valore k che vogliamo cercare,
    Troviamo la metà del nostro array, la confrontiamo con k, se k è più piccolo
    Restringiamo il nostro array verso sinistra, senò verso destra
    E si continua così fino a che o abbiamo trovato il valore, oppure si inverte l'array
  - $T_m(n) = 1.\ T_P(n) = \log n$

Metodo dell'esperto

Dato un tempo di una funzione ricorsiva

$T(n) = aT\left(\frac{n}{b}\right) + D(n) + C(n)$

Con a quante volte la parte ricorsiva è ripetuta (ex. 2 se facciamo sinistra e destra)
Ed n quante volte dividiamo un array (ex. 2 se l'array è diviso in 2 parti)
Definendo questo:

- $r(n) = aT\left(\frac{n}{b}\right) = n^{\log_b a}$
- $f(n) = D(n) + C(n)$

Abbiamo 3 casi:

1) R(n)>F(n)
   -> $f(n) = O\left(n^{\log_b a - E}\right), E > 0$
   -> Tempo = $\theta\left(n^{\log_b a}\right)$
2) F(n)>R(n)
   -> $f(n) = \Omega\left(n^{\log_b a + E}\right), E > 0$
   -> $af\left(\frac{n}{b}\right) \leq kf(n)$
   -> Tempo = $\theta(f(n))$
3) R(n)=F(n)
   -> Tempo = $\theta\left(n^{\log_b a} * \log n\right)$

Ordinamenti

- Selection sort
  - Scorriamo l'array alla ricerca del minimo ogni volta, per poi metterlo all'inizio
  - $T_m(n) = n^2 \quad T_p(n) = n^2$

- Insertion sort
  - Confrontiamo il secondo numero ed il primo e, nel caso il secondo è minore del primo, scambiamo.
    Ora confrontiamo il secondo e il terzo e, mettendo caso il terzo sia maggiore del secondo,
    A. Scambiamo il 2 e il 3
    B. Controlliamo se il 2 è minore del primo. In caso affermativo, scambiamo
  - $T_m(n) = n \quad T_p(n) = n^2$

- Merge sort
  - E' un algoritmo divide et impera suddiviso in:
    - Divide: Dividiamo l'array in 2 parti
    - Impera: Ordiniamo la prima metà e poi la seconda
    - Combina: Fonde in maniera ordinata le parti divise
    - Caso base: Abbiamo 1 valore
  - $t(n) = \begin{Bmatrix} \theta(1) \to n = 1 \\ \text{n} * \log(n) \to \text{n} > 1 \end{Bmatrix} \to \theta(n)$

- Quick sort
  - E' un algoritmo divide et impera non stabile ed è suddiviso in:
    - Divide: Divide l'array in 2 parti; una con i numeri più grandi del pivot,
      l'altra con i numeri più piccoli del pivot, con il pivot fra i due array
    - Impera: Ordina la prima parte e poi la seconda richiamando divide
    - Combina: Inutile
  - $T_m = n \log n \quad T_p = n^2$

- Counting Sort
  - Algoritmo, non basato sui confronti, non in loco.
    Abbiamo 3 array: 1) Il nostri input 2) L'output 3) Contatore di valori
    Scorriamo il nostro array di input, contiamo i valori dentro il 3 array e poi mettiamo
    I valori del 3 array sul 2
  - $T(n) = n + k \to O(n + k)$

- Radix sort
  - Metodologia per potere ordinare per più campi:
    Dati 3 parametri in ordine per importanza con cui dover ordinare: X, Y, Z
    Ordiniamo prima per Z, poi Y e infine X
    Nota: L'ordinamento usato deve essere statbile

- Heap Sort
  - Ordinamento non stabile
    - Prima esegue la buildHeap
    - scambia radice con l'ultima foglia, decrementa la lunghezza
    - Chiama heapify, ritorna al punto 2 fino a che la heap non è vuota
  - $\theta(n) = n \log n$

Strutture dati

	Ins	Search	Del	Upd	Max	Min	Succ	Prec
Array	1 - N	N	N - 1	1	N	N	N	N
Array Ord	n	Log n	N	N	1	1	1	1
Liste	1	N	1	1	N	N	N	N
Liste ord	N	N	1	N	1	1	1	1
Pile	N	N	1	1	N	N	N	N
Code	1	N	1	1	N	N	N	N
ABR	log n	Log n	Log n	Log n	Log n	Log n	Log n	Log n
Max heap	log n	N	Log n	Log n	1	N	N	N
Min heap	log n	N	Log n	Log n	N	1	N	N

- Array:
  Elemento statico, ci permette di accedere rapidamente a tutti gli elementi con però
  Problemi di spazio

- Liste
  Elemento dinamico, permette una buonissima gestione dello spazio a discapito del tempo.
  Questo perché ogni elemento punta al proprio next (anche al prev nel caso di liste circolari)
  E quindi, per raggiungere l'ultima posizione dobbiamo continuare a fare next
  Nota: per raggiungere il primo valore dobbiamo fare head[L] con L la nostra variabile
  Operazioni:
  - Insert
  - Delete
  - listMin

- Stack
  Viene utilizzata una politica LIFO (last in first out)
  Operazioni:
  - Push
  - Pop
  - StackEmpty
  - Top

- Queue
  Viene utilizzata una politica Fifo (first in first out)
  Operazioni:
  - Enqueue
  - Dequeue
  - QueueEmpty
  - Top

- Albero
  E' un particolare grafo che non è orientato, è connesso ed aciclico.
  Quando un albero è binario, ogni nodo può avere alpiù 2 figli
  Termologie:
  - Parent di un nodo: E' il nodo che possiede in left/right il nodo in questione
  - Radice: L'unico nodo che non ha un parent
  - Figlio: Nodi in left/right di un determinato nodo
  - Grado: Quanti figli ha un determinato nodo
  - Profondità: Quanto lontano un nodo è dalla radice
  - Altezza albero: profondità massima di un albero
  - Albero completo: Ogni foglia ha la stessa profondità e tutti i nodi non foglia hanno 2 figli
    -> altezza = $\log_2 n$
  - Successore: il minimo numero più grande di un determinato nodo
  - Albero binario di ricerca: Ogni nodo è sempre più piccolo di ciò che è a destra
  - Tipologie di visite:
    - preOrdine: [radice][sinistra][destra] -> print-sinistra-destra
    - inOrder: [sinistra][radice][destra] -> sinistra-print-destra
    - postOrder: [sinistra][destra][radice] -> sinistra-destra-print

  Operazioni:
  - SBT_min: Andiamo a sinistra fino a che troviamo null
  - AVL: Indica quanto un albero è sbilanciato
  - SBT_Search
    Cerchiamo un determinato valore in un albero. Per farlo confrontiamo il nodo con k
    Se k è più grande allora andiamo a destra, senò a sinistra e continuiamo i confronti.
    Si continua così fino a che o si trova k, oppure raggiungiamo null
  - Insert: Facciamo SBT_Search senza = e continuiamo fino a che troviamo null; li metteremo il valore
  - Eliminazione:
    abbiamo 2 casi:
    - 0 figli, cancelliamo il riferimento ed è fatta
    - Cancellare un nodo con 1 figlio
      E' il caso più Semplice. Cancelliamo il nodo
      Ed uniamo il padre del nodo cancellato con
      L'unico figlio. L'albero rimane uguale
    - Cancellare un nodo con 2 figli
      In questo caso, bisogna mettere il successore di quel nodo

- Heap / Code con priorità
  E' un array visto come se fosse un albero binario quasi completo.
  Operazioni:
  - heapSize -> Quanto l'array è lungo
  - Max -> Prendi massimo senza toglierlo
  - extractMax -> Prendi massimo e toglilo
  - Insert
  - Left
  - Right
  - Heapify
    Dato un nodo, prendiamo left e right. Facciamo il massimo fra il nodo, left e right e,
    Se il massimo è il nodo stesso, non facciamo nulla. Nel caso fosse o left oppure right,
    Si scambia il nodo con left/right, e poi si fa heapify su left/right
  - Buildheap
    Sono tanti heapify che partono dalle foglie fino alla radice