

Felipe Augusto Morais Silva

Counting Sort: $\theta(N+K)$ both time and space, stable; counting sort assumes that each of the n input elements is an integer in the range 0 to k , for some integer k . When $k = O(n)$, the sort runs in $\theta(N)$. Counting sort beats the lower bound of $\Omega(N \lg N)$ because it's not a comparison sort. No comparisons between input elements occur anywhere in the code. (Cormen introduction to algorithms ch8). **THE FORS ARE NOT NESTED** Code-> `func CountingSort(A,B,K){ \n Let c[0...k] be a new array \n for l=0 to k \n c[l] = 0 \n for j = 1 to A.length c[a[j]] = c[a[j]] + 1 \n for l = 1 to k \n c[l] += c[l - 1] \n for j = a.length down to 1 \n B[C[A[j]]] = A[j] \n C`

Insertion Sort: $\theta(N)$ time best case. Presta quando o array ta ordenado ou quase ordenado, caso contrario vira $O(N^2)$. Vai passando da esquerda pela direita, se o da esquerda for menor que o da direita, pega o da direita e bota no lugar certo dele (for i = 1; i < length-1; i++) `\n j = i; \n while j > 0 && a[j-1] > a[j] \n swap(a[j], a[j-1]) \n j = j - 1;`

Selection Sort: $O(N^2)$ in all cases. in each iteration we select the smallest item that hasn't been sorted yet, from left to right. For(j = 0; j < n-1; j++) `\n int iMin = j; \n for(i = j + 1; i < n; i++) \n if(a[i] < a[iMin]) \n iMin = i; \n if(iMin != j) \n swap(a[j], a[iMin]);`

QuickSort: ($N \log N$) best case. PIVOT. Pega o pivot (geralmente o do meio) e divide baseado no pivot, joga todos os menores pra esquerda e depois os maiores pra direita, quando cruzar o ponteiro da direita com o ponteiro da esquerda, quebra o array em 2, e repete recursivo pras partes menores.

HeapSort: ($N \log N$) best case. Build_Heap(first part, is $O(N)$) && Heapify(second part, is $O(\log N)$ called $n-1$ times). We gotta look at the array as a tree. Heap invertido. Maior na raiz (root)

ShellSort: n tem analise de complexidade. É basicamente um insertion modificado, seleciona elementos a partir do vão entre eles

OBSERVAÇÕES DO MAX QUE PODEM VIR A SER ÚTEIS

- Insertion e select ambos fazem $O(N^2)$ comparações
- Binary search é $O(\log N)$
- O quicksort, mergesort e heapsort fazem $O(N \log N)$ comparações
- O melhor caso do algo de inserção ocorre quando ordenamos um array já ordenado ou quase ordenado. Bala de prata.
- O counting sort realiza $O(N)$ comparações, contudo só funciona em casos específicos e triplica a memória gasta
- Shellsort não tem análise de complexidade estabelecida
- Se um algo for $O \rightarrow$ qualquer pra cima, $\theta \rightarrow$ exato, $\Omega \rightarrow$ qualquer para baixo.
- The $\Omega(N \log N)$ lower bound for comparisons does not apply when we depart from the comparison sort model (cormen)
- Só é $\theta(n)$ quando é $O(n)$ e $\Omega(n)$ ao mesmo tempo
- No insertion sort o i indica o elem. A ser inserido no conj ord. O j começa em $i - 1$ e $j -$ - until we find the insertion pos.
- O laço interno do insertion procura a posição de inserção e movimenta os elementos maiores que o valor inserido
- Algo. In-place = memória adicional é requerida independente do tamanho do array
- A posição do pai no arrya vai ser a posição do filho dividida pelo numero de nodes da fileira do pai
- O max falou que a segunda etapa do heapsort é ($N \log N$)
- O selection sort é o melhor alg em termos de movimentação de elementos do array, ele faz $O(N)$ movimentações.

NAME	BEST	AVERAGE	WORST	MEMORY	STABLE
Quicksort	$N \log n$	$N \log n$	N^2	$\log n$	NO
Merge Sort	$N \log n$	$N \log n$	$N \log n$	N	YES
Heapsort	$N \log n$	$N \log n$	$N \log n$	1	NO
Insertion sort	N	N^2	N^2	1	YES
Selection sort	N^2	N^2	N^2	1	NO

Summation:

Somatório de constante = $n + 1$ quando começa do 0, esse 1 aí é só tirar caso comece do 1 no lugar do zero

Gauss -> é quando vai de 0 a N com i sendo somado, aí fica $(n*(n+1))/2$

Passo base é S_0 , so trocar todos os n por 0 e ver se vai dar 0 no final.

Indução é isso aqui: $S_n = S_{n-1} + A_n \rightarrow$ esse a_n aí é a soma, então se for somatório de i^2 troca o A_n por n^2

Felipe Augusto Morais Silva

Quando der potência tem que fazer o produto notável antes!!!

Fibonacci with memoization

Caso repetir a questão da prova do ano passado, so fazer fib com cache: Array global de long pra salvar o cache -> instanciar o array global na main -> fazer o fibonacci: long fib n(int n) se $n \leq 1$ retorna n; \n checar se o cache na pos n != 0, se sim retorna o cache[n], long ans = fib(n-2) + fib(n-1); \n cache[n] = ans; \n return ans;

Stack, Queue and list

A implementação das 3 usa um array[]. O max implementou o stack usando queue.

STACK -> Tem 2 possibilidades de implementação: IF e RF (A correta, que presta) / II e RI (inserção e remoção não eficientes, pq vai ter que movimentar todos os outros elementos da estrutura pra fazer isso).

QUEUE -> 2 possible solutions: IF e RI (aí fica com a remoção não eficiente) / II e RF (fica com a inserção não eficiente)

ARRAYLIST

A inserção em um arraylist cheio duplica dinamicamente sua capacidade, o tamanho de um objeto arraylist pode ser definido em usa declaração e modificado dinamicamente, a remoção de um elemento do arraylist mantém sua capacidade, o tamanho padrão de um objeto arraylist é zero. O arraylist pode ser manipulado para simular uma fila ou pilha.

HEAP

A posição do pai no array vai ser a posição do filho dividida pelo número de nodes na fileira do pai. Então no caso da árvore binária: Array[1] vai ser sempre a raiz, não pode começar no zero pq se não não tem como checar se tem filho pq filho vai ser sempre array[l * 2]

Array[1] -> raiz

Array[l/2] -> é o pai do array[l], para $l > 1$

Se Array[2 x l] e array [2 x l + 1] existem, então eles são filhos de array[l].

Felipe Augusto Morais Silva