Felipe Augusto Morais Silva

Counting Sort: **θ (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 θ (N). Counting sort beats the lower bound of Ω(N lg N) because it's not a comparison sort. No comparisons between input elements occur anywhere in the code. (Cormen intruduction 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 I =0 to k \n c[I] = 0 \n for j = 1 to A.length c[a[J]] = c[a[J]] + 1 \n for I = 1 to k \n c[i] += c[i – 1] \n for j = a.length down to 1 \n B[C[A[J]]] = A[J] \n C

Insertion Sort: θ (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: (NLogN) 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: (NLogN) best case. Build_Heap(first part, is O(N)) && Heapify(second part, is O(LogN) 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 fazen o(N^2) comparações
- Binary search é O(LogN)
- O quicksort, mergesort e heapsort fazem O(NLogN) 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-> qualquer pra cima, **θ->** exato, Ω-> qualquer para baixo.
- The Ω(NLogN) lower bound for comparisons does not apply when we depart from the comparision sort model (cormen)
- Só é **θ(n)** quando é O(N) e Ω(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 (NlogN)
- 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 S0, so trocar todos os n por 0 e ver se vai dar 0 no final.

Indução é isso aqui: Sn = Sn-1 + An -> esse an ai é a soma, então se for somatório de I^2 troca o An por n^2

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

Fibonacci with memoization

Caso repetir a questao 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 <= 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 mantem sua capacidade, o tamanho padrão de um objeto arraylist é zero. O arraylist pode ser manipulado para simular uma fila ou pilha.

HEAP

A posicao do pai no array vai ser a posicao do filho dividida pelo numero de nodes na fileira do pai. Entao no caso da arvore binaria: Array[1] vai ser sempre a raiz, nao pode comecar no zero pq se nao nao tem como checar se tem filho pq filho vai ser sempre array[I * 2]

Array[1] -> raiz

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

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