FGV EMAp ESCOLA DE MATEMÁTICA APLICADA

Projeto e Análise de Algoritmos

2023.2 | A1 | Prova

9.8/

Nome: Otávio Augusto Matos Alves

Atenção: As questões que solicitarem o projeto de um algoritmo deverão conter em sua resposta (1) o pseudocódigo do algoritmo, ou (2) uma descrição textual sem ambiguidade com os passos necessários para implementá-lo.

Questão 1 (2 pontos)
Seja $T(n)$ a complexidade do pior caso do algoritmo executado por $func$, determine uma função $f(n)$ tal que $T(n) = \Theta(f(n))$. A solução deverá explicar como $f(n)$ foi encontrada.

```
void s(int v[], int i, int j) {
    int z = v[i];
    v[i] = v[j];
    v[j] = z;
}

int h(int v[], int a, int b) {
    int k = v[b];
    int f = a;
    for (int i=a; i < b; i++) {
        if (v[i] <= k) {
            s(v, i, f);
            f++;
        }
    }
    s(v, f, b);
    return f;
}

void g(int v[], int a, int b) {
    while (a < b) {
        int j = h(v, a, b);
        if (j - a < b - j) {
            g(v, a, j - 1);
            a = j + 1;
        } else {
            g(v, j + 1, b);
            b = j - 1;
        }
```

FGV EMAp ESCOLA DE MATEMÁTICA APLICADA

Projeto e Análise de Algoritmos

2023.2 | A1 | Prova

```
    }
}

bool func(int v[], int n, int k) {
    g(v, 0, n - 1);
    int i = 0;
    int j = n - 1;
    while (i < j) {
        int s = v[i] + v[j];
        if (s < k) {
            i++;
        } else if (s > k) {
            j--;
        } else {
            return true;
        }
    }
    return false;
}
```

Questão 2 (1 ponto)

Dada uma sequência A ordenada contendo n inteiros distintos, crie um algoritmo capaz de determinar se existe um índice i tal que $A[i] = i$. A complexidade do pior caso deverá ser $O(log(n))$. Justifique a sua resposta.

Nessa questão não é permitido apenas fazer referência a algoritmos apresentados em aula para compor a solução. Tais algoritmos podem ser utilizados, no entanto devem ser descritos explicitamente.

Questão 3 (3 pontos)

Um determinado sistema requer um módulo para gerenciar uma fila de tarefas. Cada elemento na fila possui uma referência para a estrutura de dados com as informações de execução da tarefa, e um número representando a sua prioridade (quanto menor o valor, maior a prioridade). O módulo deverá fornecer as seguintes operações:

- `void` **`add_task`**`(Task t, int p)`
  - Adiciona a tarefa t na fila com a prioridade p, em $O(log(n))$.
- `Task` **`next_task`**`()`
  - Retorna (sem remover) a tarefa t com maior prioridade, em $O(1)$.
- `void` **`remove_task`**`()`
  - Remove a tarefa t com maior prioridade, em $O(log(n))$.

a) Projete uma solução para construir esse módulo: descreva a ideia geral e a(s) estrutura(s) de dados utilizadas.
b) Crie um algoritmo para cada operação considerando a especificação acima, e analise sua complexidade.
c) Surgiu um novo requisito: alterar a prioridade de uma tarefa na fila. A operação deverá receber a referencia para a tarefa e a nova prioridade. Crie uma solução para atender essa nova operação e analise a sua complexidade. Quanto mais eficiente melhor.

Nessa questão não é permitido apenas fazer referência a algoritmos apresentados em aula para compor a solução. Tais algoritmos podem ser utilizados, no entanto devem ser descritos explicitamente.

Questão 4 (2 pontos)
Uma sequência A contém n inteiros positivos distintos. Crie um algoritmo que encontre os k números mais próximos da mediana de A, sendo $k \leq n$. A distância entre dois elementos a_i e a_j é definida por $|a_i - a_j|$. A complexidade do pior caso deverá ser $O(n)$.
Caso seja necessário, é permitido nessa questão fazer referência a algoritmos apresentados em aula para compor a solução.

Questão 5 (2 pontos)
Uma sequência A contém n inteiros positivos. Cada número em A pertence ao conjunto $\{n^2, n^2+1, n^2+2, \ldots, n^2+n\}$. Crie um algoritmo capaz de encontrar o número de A que mais se repete (caso exista mais de um pode retornar qualquer um dos números empatados). A complexidade do pior caso deverá ser $O(n)$.
Caso seja necessário, é permitido nessa questão fazer referência a algoritmos apresentados em aula para compor a solução.

1) O que a função funk faz é chamar g que é um algoritmo de ordenação (Quicksort) e depois a partir de um ponteiro no início e outro no fim pulam dois elementos toda vez que a soma seja k.

Essa última parte é O(n) já que percorremos com o ponteiro do início ou o do fim e é garantido de se encontrarem (no pior caso) não mais em n passos e cada operação dentro do loop é O(1).

A parte do Quicksort temos a função 1 = swap e h = partition e g é a parte lógica central do Quicksort. Vemos para cada partição é chamado a partition O(n) no tamanho da partição e a recursão é feita do lado esquerdo e direito do Pivô, no melhor caso temos:

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

Pelo teorema Mestre segundo nos: $T(n) = O(n \log n)$

Porém no pior caso temos que o pivô escolhido é o maior ou o menor elemento da partição então:

$$T(n) = T(n-1) + n$$

Por iteração:

$$T(n) = T(1) + n + n-1 + n-2 + \dots + 1$$

$$T(n) = O(n^2)$$

O algoritmo iteiro é então $O(n^2) + O(n) = O(n^2)$ no pior caso.

✓ 2.0

② A é ordenado e tem inteiros distintos

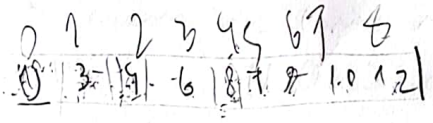
Verificar se existe $A[i] = i$ em $O(\log n)$

Suponha A ordenado em ordem crescente.

Vamos fazer uma busca binária no array

se $A[i] > i$ vamos fazer a busca novamente de 0 a $\frac{n}{2}$

Caso $A[i] = i$ retorna $A[i]$, caso contrários fazemos a busca de $\frac{n}{2}$ a n.




```
busca(A, inicio, fim):
    meio = (inicio + fim) // 2
    se inicio > fim:
        Retorna False
    se A[meio] == meio:
        Retorna A[meio]
    se A[meio] > meio:
        Retorna busca(A, inicio, meio - 1)
    se A[meio] < meio:
        Retorna busca(A, meio + 1, fim)
```

Como a busca sempre corta pela metade o nº de comparações e cada iteração são $O(1)$ temos:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

$$\Rightarrow T(n) = O(\log n).$$

✓ 1.0

(3) a) A ideia será usar a estrutura de Heap. que é uma árvore binária em que os pais são menores que os filhos (Heap Mínimo).
porém faremos em um Vetor tal que os filhos do nó i são 2i+1 e 2i+2.
Assim a ideia é primeiro criar o Heap Mínimo e sabendo dessa estrutura conseguiremos adicionar, ver e remover uma task no tempo desejado.
A criação do Heap é O(n).

✓ 0.5

b) Para o add-task vamos adicionar este elemento no vetor na última posição, verificamos se o novo elemento é menor que o pai COMO? fazemos um loop de trocar ~~este~~ deste elemento com seu pai até ele ser maior que o pai. Isso é O(1) + O(log n) pois inserir o elemento no vetor é constante e o Heap tem uma altura = log n o que garante que o loop tem tamanho máximo de log n.

Para o next-task basta verificar a primeira posição do vetor e retornar esta operação é O(1).

Para o remove-task, podemos fazer um swap o elemento da primeira posição com o do último e logo depois apagar este de nova prioridade. agora para arrumar o Heap, fazemos a comparação do primeiro com os dois filhos e faz swap com o menor filho, repete o processo até não ser maior que os filhos. ou não tem filho. Isso é feito em O(log n) já que o máximo de comparações ~~[illegible]~~ é função de log n pela altura. Isso também garante que não teremos elementos vazios no meio do vetor / árvore.

✓ 1.9

③ Depois de alterar a prioridade do elemento, verificamos se este elemento é maior que o filho, COMO ENCONTRAR? entramos num loop de trocas de elementos com seu filho até o elemento ser menor que o filho ou não ter filhos maior.

se o elemento é menor que o pai, entramos num loop de trocas com o pai deste elemento até o elemento ser maior que o pai, ou este ser o primeiro elemento da lista.

Isso é garantido de ajeitar o Heap já que sabemos que os filhos dos filhos são maiores que o nó pai.

Este algoritmo é $O(\log n)$ já que a manutenção da prioridade é constante e o máximo de trocas que podemos fazer é $\log(n)$. (altura do Heap).

✓ 0.4

(4) A ideia aqui é utilizar o quickselect com mediana of medianas (para otimizado e garantir que é $O(n)$) para achar a mediana,

Achada a mediana criamos um vetor auxiliar com todos os elementos de A tal que $B[i] = (|A[i] - \text{mediana}|, \text{bool})$ o bool é True se $A[i] > \text{mediana}$, False caso contrário.

Agora com o quickselect com mediana of medianas implace no auxiliar, achamos o K-ésimo elemento mais próximo de 0. (menor).

Para achar criamos uma lista e adicionamos os elementos do vetor auxiliar até K nessa lista se o bool for falso, adicionamos $-aux[i] + \text{mediana}$ se bool for True, adicionamos $+ aux[i] + \text{mediana}$, retorna essa lista.

Para achar a mediana o algoritmo tem complexidade $O(n)$.
Para criar o vetor auxiliar $O(n)$
para achar implace o k-ésimo menor no vetor auxiliar $O(n)$.
Para inserir os elementos na nova lista $O(K)$
Total $O(n) + O(n) + O(n) + O(K) = O(n)$

✓ 2.0

⑤ Sabemos que $A \in \{m^2, m^2+1, \ldots, m^2+m\}$

Criamos um array com $m+1$ posições, inicialmente zeradas.

Para cada elemento de A, $A[i]$, somamos 1 na posição $A[i] - m^2$ do array.

Depois percorremos o array guardando o maior número, e retornamos a posição com maior número no array $+ m^2$.

```
maior_frequente ( A , m)
    Vetor [m+1] = 0  em todos os m+1 posições
    Para cada i até m:
        Vetor [A[i] - m^2 ] += 1
    maior = (0,0)
    Para todos elementos no vetor:
        se elementos > maior [0]
            maior = (elementos, posição elementos).
    Retorna A[ maior [1] ] + m^2.
```

Vejo que a criação do array é $O(m)$ e passa pra saber o maior depois também é $O(m)$ concluímos que o algoritmo é $O(m)$.

✓ 2.0