Bruno Pereira Fornaro

8.5

FGV EMAp
ESCOLA DE MATEMÁTICA APLICADA

Projeto e Análise de Algoritmos

A1 | Prova

REV 9.0

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. É permitido fazer referências à algoritmos apresentados em aula para compor a solução, caso seja necessário.

Questão 1 (1 ponto)

Determine uma função $f(n)$ para o código a seguir tal que $T(n) = O(f(n))$. A função auxiliar $pow(b, e)$ calcula o valor de b elevado à e. A solução deverá explicar como $f(n)$ foi encontrada.

```
void f(int n) {
    int i = 0;    // O(1)
    while (pow(i, 2) <= n) { // O(1)·O(√n) , pois i ≤ √n
        i++;   // O(1)
        int x = 0;   // O(1)
        while (x < i) {  // O(√n) , pois x ≤ i ≤ √n
            ++x;  // O(1)
        }
    }
}
```

Questão 2 (1.5 pontos)

Dada uma sequência A com n números naturais distintos, projete um algoritmo capaz de verificar se existem três números distintos em A cuja soma seja igual a x. O algoritmo deverá apresentar uma complexidade $O(n^2 \log(n))$.

Questão 3 (3 pontos)

Uma sequência A de tamanho n contém inteiros positivos e negativos.

a) Projete um algoritmo $O(n^3)$ capaz de determinar os índices i e j, sendo $i < j$ tal que a soma de $A[i]+\ldots+A[j]$ é máxima.
b) Otimize a solução do item anterior produzindo um algoritmo $O(n^2)$.
c) Avalie se é possível produzir um algoritmo $O(n)$ para esse problema. Caso seja possível, apresente o algoritmo, caso contrário prove que é impossível.

Questão 4 (1 ponto)

Explique como funciona o algoritmo de remoção de um elemento em uma tabela hash, considerando resolução de colisão através de:

a) Lista encadeada, contendo em cada elemento o valor e um ponteiro para o próximo elemento.
b) Endereçamento aberto.

Questão 5 (2 pontos) m n

Dadas as sequências A e B contendo m e n inteiros positivos ($m \geq n$), respectivamente, projete um algoritmo capaz de gerar uma sequência C contendo os elementos de A reordenados segundo a ordem dos elementos de B. Os elementos de A que não estiverem mapeados na sequência B deverão ser inseridos ao final de C em ordem crescente. Exemplo:

Entrada:
$A = 5, 8, 9, 3, 5, 7, 1, 3, 4, 9, 3, 5, 1, 8, 4$
$B = 3,5,7,2$
Saída:
$C = 3, 3, 3, 5, 5, 5, 7, 1, 1, 4, 4, 8, 8, 9, 9$

O(mLog(m)) ordenar A
O(Log(m)) binary search

O algoritmo deverá apresentar uma complexidade $O(mlog(m))$.

Questão 6 (1.5 pontos)

Dada uma sequência A com n números reais distintos, projete um algoritmo capaz de encontrar os $\sqrt{n}$ menores números. O algoritmo deverá apresentar uma complexidade $O(n)$.

Questão 1

Considerando que pow(i, 2) é $O(1)$, pois pow(i, 2) = $i \cdot i$ (podemos ter uma condição na função pow() para simplicar dessa forma), podemos ver que temos diversas operações $O(1)$ e um caso onde fazemos $O(\sqrt{n})$ vezez uma operação $O(\sqrt{n})$ ✗ (nos Loops "while" encadeados, como podemos ver nos "comentários" feitos a cada linha de código no enunciado para auxiliar), dessa forma o custo computacional da função f é $O(n)$ e a função $f(n)$ tal que $T(n) = O(f(n))$ é $f(n) = n$.

0.5

Observação: As próximas questões têm descrições textuais dos algoritmos propostos, com pequenas partes entre as descrições na forma de pseudocódigo, para facilitar a compreensão.

Além disso, as respostas não foram ordenadas mas há títulos identificando as respectivas respostas de cada questão

Questão 2

Deveremos ter uma função que recebe como parâmetros A, n e x (como descritos no enunciado). Faremos:

- Começamos ordenando A com o heap sort (operação $O(n\,Log(n))$ mesmo no pior caso);
- Vamos iterar os elementos de A começando com uma variável de índice idx1 e outra idx2. Começamos com idx1=0 e idx2=1. Faremos (pseudocódigo):

```
De idx1=0, enquanto idx1 < n-3, idx1=idx1+1, Fazer:
    De idx2=1, enquanto idx2 < n-2, idx2=idx2+1, Fazer:
        int diff = x - A[idx1] - A[idx2];
        int result = binarySearch(A, idx2+1, n, diff);
        Se result != -1;
            retorna verdadeiro;
    Fim do Loop;
Fim do Loop;
retorna Falso;
```

1.5

- Vemos acima que as duas iterações (Loops "For" encadeados) têm tempo computacional no máximo $O(n^2)$.
- A função binarySearch() usada axiliarmente faz uma busca binária em A, dos índices idx2+1 até n, procurando o inteiro diff (retornando -1 caso não encontre). A operação é $O(Log(n))$
- Se é encontrado um natural result que somado aos números dos índices idx1 e idx2 resulte em X, retorna verdadeiro.
- Se após todas as iterações não encontra os 3 números que satisfaçam as condições desejadas, retorna Falso.

Vemos que fazemos no máximo $c_1 n^2$ vezes operações $O(Log(n))$, então o tempo computacional é $O(n^2 Log(n))$. (o restante das operações são $O(1)$).

Questão 3, item a

- Fazemos Loops "For" encadeados, externamente de $i=0$ até $n-2$, internamente de $j=i+1$ até $n-1$. A cada iteração dos respectivos Loops incrementamos os contadores i e j em 1.
- Dentro do Loop mais interno, somamos de $A[i]$ até $A[j]$ (com outro Loop "For", evidentemente). Após isso sempre armazenamos o valor da maior soma e os respectivos índices.
- Retornamos os índices da maior soma.

Temos um tempo no máximo $O(n^3)$ pois cada um dos 3 Loops encadeados tem tempo no máximo $O(n)$. Logo, o algoritmo é $O(n^3)$, já que retornar os índices é $O(1)$.

1.0

Questão 3, item c

É impossível, pois é sempre necessário verificar a soma de subconjuntos de tamanhos de 1 até n, com o tempo de soma sendo $O(m)$, com m o tamanho do subconjunto, e o devemos comparar $n-m$ subconjuntos de tamanho m, o que é pelo menos $O((n-m)m)$. Dessa forma, o algoritmo deve ser pelo menos $O(n^2)$.
(Tendo em vista que $m \in [0, n]$)

0

Questão 3, item b

Na segunda etapa, em vez de somar novamente os elementos de i até j todos de novo, podemos sempre ~~ter~~ salvar a soma do Loop anterior e quando o Loop interno faz $j+=1$ apenas somamos $A[j]$ na soma anterior e comparamos com a maior soma. É importante lembrar de zerar a variável da soma sempre que fazemos a iteração do Loop mais interno ($i+=1$) e começar o processo de salvar de novo.
("$j+=1$" e "$i+=1$" são os incrementos do Loop, a cada iteração).

1.5

Reduzimos um Loop interno $O(n)$ e podemos ver que com isso reduzimos o tempo do algoritmo para $O(n^2)$

Questão 4, item a.

- Primeiro, utilizamos a função hash da tabela para encontrar onde o elemento pode ter sido inserido.
- Após isso, percorremos a lista encadeada sempre armazenando o ponteiro do item anterior e do item atual, buscando o elemento que deve ser removido.
  - Se não encontramos o elemento, seguimos para o próximo nó
  - Se encontramos, removemos da lista mudando o ponteiro do nó anterior ao atual para o próximo ao atual (caso o anterior seja nulo, o nó é raiz e atualizamos a raiz para o próximo elemento ou ponteiro nulo). Devemos deletar o nó após isso, para liberar a memória. Retornamos após isso.
  - Se não encontramos o elemento após percorrer toda a lista, apenas retornamos.

Observação: Modificações podem ser feitas nos retornos, se desejado, mas não impacta na remoção (isso vale para a questão 4 item b também).

✓ 0.5

Questão 4 item b.

- Utiliza a função hash para encontrar a posição que o elemento deveria ter sido inserido (primeiro caso). Após isso, faz:
  - Se foi encontrado o elemento, marca como removido (flag necessária) e retorna.
  - Se na posição atual está marcado como elemento removido, utiliza a função de "remapeamento" para encontrar a próxima posição que o elemento pode ter sido inserido e volta para a etapa anterior
  - Se na posição atual não está marcado como elemento removido e o elemento não foi encontrado, apenas retorna.

✓ 0.5

(fim, o retorno já deve ter sido feito)

Questão 5.

- Ordenamos A com heap sort (sabemos que é O(mLog(m))
- Para cada elemento x em B, percorre A buscando x e quando encontra x adiciona no final de C (pode ser armazenado sempre o índice de onde deve ser inserido o próximo elemento em C, e C já deve ter sido criado). Além disso, ao encontrar x em A trocamos o respectivo valor em A para -1. Essa operação é O(nLog(m)) utilizando busca binária.
- Passamos por A inserindo em ordem em C todos os números diferentes de -1 (operação O(m)).

Logo, vemos que esse algoritmo terá custo computacional O(mLog(m)), pois é a "composição" ("soma") de etapas no máximo O(mLog(m)).

Observação: Existem duas etapas onde elementos podem ser inseridos em C, cada uma com custo menor ou igual O(mLog(m)), e isso também foi considerado implicitamente.

2.0

Questão 6

- Utilizaremos o algoritmo do quick selection com o algoritmo da mediana das medianas (que otimiza a escolha do pivô do quick sort), de forma "inplace" (ou seja, alterando a ordem dos elementos da sequência onde se busca), para encontrar o elemento do índice com valor $\sqrt{n}$ (ou, caso não seja inteiro, devemos truncar, ou seja, queremos o índice $\lfloor\sqrt{n}\rfloor$). Operação $O(n)$
- O cálculo de $\sqrt{n}$ pode ser feito em $O(1)$ por aproximação com a série de Taylor.
- Após encontrar o elemento desejado, os elementos do índice 0 até o índice $\lfloor\sqrt{n}\rfloor$ serão os $\sqrt{n}$ menores elementos e devemos retorná-los (operação $O(n)$).

Podemos que temos três etapas $O(n)+O(1)+O(n)$, logo o tempo computacional é $O(n)$

A SEQUÊNCIA NÃO ESTÁ NECESSARIAMENTE ORDENADA

✓ 1.0

REV: ✓ 1.5

ACEITA A EXPLICAÇÃO DOS ELEMENTOS ESTAREM NAS PRIMEIRAS $\sqrt{n}$ POSIÇÕES PELO QUICK SELECT TER SIDO FEITO INPLACE