Lista de Exercícios 1 - Algoritmos I

Rita Rezende Borges de Lima
Matrícula: 2020065317

March 8, 2021

1. Uma maneira simples de encontrar o "pico" de um vetor, v, é aplicar uma busca binária neste. Considere n o tamanho do vetor dado, iremos sempre analisar o valor central da parte analisada do vetor, no início este número m será $\lceil \frac{n}{2} \rceil$, este será comparado com os elementos em seu lado esquerdo e direito. Teremos três casos:

   - `v[m-1]` > `v[m]` > `v[m+1]` como o vetor está em declínio o pico necessariamente estará a esquerda do ponto m analisado, logo continuaremos procurando apenas na parte a esquerda de m.
   - `v[m-1]` < `v[m]` < `v[m+1]` como o vetor está em crescendo o pico necessariamente estará a direita do ponto m analisado, logo continuaremos procurando apenas na parte a direita de m.
   - `v[m-1]` < `v[m]` > `v[m+1]` m será o pico, logo a execução pode ser encerrada.

Algorithm 1: Encontra o "pico" de um vetor

```
input  : O vetor V de tamanho n
output: O índice p do pico

l ← 0, r ← size
while l < r do
    m ← (l+r)/2
    if m for igual a 0 ou size then  termina loop

    if V[m-1] > V[m] > V[m-1] then
        p está depois de m
        l ← m
    else if V[m-1] < V[m] < V[m-1] then
        p está antes de m
        r ← m + 1
    else
        termina loop
p ← m
return p
```

A complexidade do algoritmo proposto é a complexidade da busca binária, $O(log(n))$. Como sempre estamos dividindo nosso vetor e fazendo comparações é simples perceber que o algoritmo segue a seguinte relação de recorrência:

$$T(n) = T(\lceil n/2 \rceil) + 1 \tag{1}$$

Lemma 0.1. *Se T(n) satisfaz a seguinte equação de recorrência, T(n) = $log_2(n)$.*

$$f(x) = \begin{cases} 0, & se\ n = 1. \\ T(\lceil n/2 \rceil) + 1, & se\ n > 1. \end{cases} \tag{2}$$

Proof.

$$T(n) = T(\lceil n/2 \rceil) + O(1) \tag{3}$$

Expandindo a equação 1:

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + 1 \\ T(\lceil n/2 \rceil) &= T(\lceil n/4 \rceil) + 1 \\ T(\lceil n/4 \rceil) &= T(\lceil n/8 \rceil) + 1 \\ T(\lceil n/8 \rceil) &= T(\lceil n/16 \rceil) + 1 \end{aligned}$$

Substituindo Uma equação na outra obtemos a seguinte relação:

$$\begin{gathered} T(n) = T(\lceil n/8 \rceil) + 1 + 1 + 1 \\ T(n) = T(\lceil n/16 \rceil) + 1 + 1 + 1 + 1 \\ \vdots \\ T(n) = T(1) + k \end{gathered}$$

É possível perceber o seguinte padrão:

$$T(n) = T(n/2^k) + k \tag{4}$$

Como $2^k = n$, $k = log_2(n)$. Usando T(n) = T(1) + k, é simples perceber que a complexidade é da forma $O(log_2(n))$.

$\square$

2. Uma mandeira de modelar o problema é considerar cada dia um elemento de um vetor com n dias onde a posição 0 é o primeiro dia e a posição n-1 é o último. Usaremos a estratégia dividir para conquistar. Inicialmente iremos dividir esse array em dois, a resposta ótima pode ocorrer de três maneiras:

   - O par solução está completamente na primeira metade
   - O par solução está completamente na segunda metade
   - O par solução está em ambas as partes, compramos na primeira e vendemos na segunda.

   Para acharmos a solução do primeiro e segundo item podemos chamar recursivamente a função com limites para qual parte do vetor analisar. Para encontrarmos a solução do terceiro item basta encontrar o menor elemento da parte da esquerda do vetor e o maior item da parte direita por uma busca linear simples.

Algorithm 2: Encontrar maior diferença entre dois elementos de um vetor quando o primeiro elemento é menor que o segundo

```
Function divideAndConquer(l,r, V):
    if l ≥ r then return par(l, r)

    m ← (l+r)/2
    parEsq ← divideAndConquer(l, m, v)
    parDir ← divideAndConquer(m + 1, r, v)
    esq ← índice do menor elemento da parte esquerda
    dir ← índice do maior elemento da parte direita
    if v[parEsq.i] - v[parEsq.j] > v[parDir.i] - v[parDir.j] then
        if v[parEsq.i] - v[parEsq.j] > v[esq] - v[dir] then
            return parEsq
        else
            return par(esq, dir)
    else
        if v[parDir.i] - v[parDir.j] > v[esq] - v[dir] then
            return parDir
        else
            return par(esq, dir)
```

A função chama a si mesma duas vezes com metade do tamanho do vetor. Como cada iteração tem tempo de execução linear (as buscar por maior e menor valor no vetor), a função pode ser descrita pela seguinte relação de recorrência:

$$T(n) = 2T(\lceil n/2 \rceil) + cn \tag{5}$$

Lemma 0.2. *Se T(n) satisfaz a seguinte equação de recorrência, T(n) = n * log_2(n).*

$$T(n) = \begin{cases} 0, & se\ n = 1. \\ 2T(\lceil n/2 \rceil) + cn, & se\ n > 1. \end{cases} \tag{6}$$

Proof. Por indução em `n`.

- **Caso Base:** n = 1, T(1) = 0 que por sua vez é igual n*$\log_2$(n)
- **Hipótese Indutiva:** T(n) = n*$\log_2$(n)

T(2n) = 2T($\lceil n \rceil$)+$2n$	Recorrência para 2n
$T(2n) = 2n * log_2(n) + 2n$	Substituindo a Hipótese Indutiva
$T(2n) = 2n*(log_2(2n)-1)+2n$	Pela propriedade de Log, log(a) - log(b) = log(a/b)
$T(2n) = 2n * log_2(2n)$	

□

É interessante notar que existe uma solução que não utiliza da estratégia de dividir para conquistar com complexidade linear. Basta percorrer o vetor da última posição até a primeira e manter três variáveis auxiliares para guardar o máximo utilizado, o máximo global e o minimo utilizado. Caso a posição atual analisada seja maior que o máximo global o máximo global vira esta posição, caso a posição analisada seja tal que o máximo global menos esta seja maior que o máximo utilizado menos o minimo utilizado, o máximo global vira o máximo utilizado e esta posição vira o mínimo utilizado. No final da execução os dias serão os indíces do vetor marcados por mínimo utilizado e máximo utilizado.

Algorithm 3: Encontra melhores dias de compra e venda

```
input  : O vetor V com dias e seu tamanho sz
output: Os melhores dias para compra e venda

cmax ← sz - 1, global max ← size - 1, min ← size - 1
for i = sz − 2; i >= 0; i − − do
    if v[i] > v[global max] then
        global max ← i
    else if v[global max] - v[i] > v[cmax] - v[min] then
        min ← i, cmax ← global max
return min, cmax
```

3.

4.

Lemma 0.3. *O algoritmo dado na questão 4 minimiza o número de paradas*

Proof. Por indução em p_i.

* **Caso Base:** Considere o caso de base p_0, como ainda estamos no início do trajeto e nenhuma decisão foi feita, uma solução ótima ainda pode ser encontrada.
* **Hipótese Indutiva:** Até o ponto de pausa p_k as escolhas até ali permitiram que o trajeto até aquele momento tenha o mínimo de paradas necessárias, ou seja, o trajeto de p_0 até p_k é ótimo.

* De acordo com o enunciado os viajantes sempre seguem até o próximo ponto de descanso se possível, logo caso ao chegar em p_k seja aferido que é possível ir até p_{k+1} antes do anoitecer não haverá uma parada em p_k, de forma que o trajeto p_k até p_{k+1} é ótimo. Devido a nossa hipótese indutiva p_0 até p_{k+1} é ótimo.

□

5. A maneira ótima de ordenar as licenças na ordem que devem ser adquiridas é por tamanho da taxa de forma decrescente. Ou seja a solução sempre será da forma $L = \{l_1, \cdots, l_n\}$ onde seja r_i a taxa de crescimento de l_i, $r_1 > r_2 > \cdots > r_n$.

Lemma 0.4. *A ordenação L que segue preço da taxa de forma decrescente sempre tera o menor custo total.*

Proof. Para provarmos que a ordenação descrita produz o menor valor basta provar que toda soma de inversão gera um valor maior do que a soma de um par da ordenação, ou seja: $100 * r_{k+i}^{k} + 100 * r_{k}^{k+i} > 100 * r_{k}^{k} + 100 * r_{k+i}^{k+i}$.

Considere a ordenação descrita onde $r_k > r_{k+i}$. Manipulando a inequação:

r$_k^k > r_{k+i}^k$ válido pois k é um inteiro maior que 1

$r_k^i > r_{k+i}^i$ válido pois i é um inteiro maior que 1

$r_k^i + 1 > r_{k+i}^i + 1$

$1 - r_{k+i}^i > 1 - r_k^i$

$-(1 - r_{k+i}^i) < -(1 - r_k^i)$

$-r_{k+i}^k(1 - r_{k+i}^i) < -r_k^k(1 - r_k^i)$ válido pois $r_k^k \geq r_{k+i}^k > 0$

$r_{k+i}^k(1 - r_{k+i}^i) > r_k^k(1 - r_k^i)$

$r_{k+i}^k - r_{k+i}^{k+i} > r_k^k - r_k^{k+i}$

$r^k_{k+i} + r^{k+i}_k > r^k_k + r^{k+i}_{k+i}$

$100 * r^k_{k+i} + 100 * r^{k+i}_k > 100 * r^k_k + 100 * r^{k+i}_{k+i}$

Como k e i eram número arbitrários provamos que para qualquer par dos $\binom{n}{2}$ pares possíveis obtemos um valor menor com as taxas de crescimento maiores antes. $\square$

6. Para encontrarmos o número da sequência S = $\{x_0, \cdots, x_{n-1}\}$ que está faltando é necessário inicialmente descobrir de quanto em quanto cresce a sequência, isso pode ser feito da seguinte maneira:

$$dif = min(x_1 - x_0, x_{n-1} - x_{n-2}) \tag{7}$$

Com esse valor é simples calcular quanto cada número da sequência seria caso não faltasse um número:

$$x'_i = dif * i + x_0 \tag{8}$$

Desta forma temos dois casos:

- $x'_i = x_i$: O número que falta na sequência está a direita de i.
- $x'_i > x_i$: O número que falta na sequência está a esquerda de i.

Logo, com uma simples busca binária conseguimos encontrar o número adjacente ao que está faltando, olhamos os vizinhos deste e verificamos onde o número está faltando.

Algorithm 4: Encontra o número que falta na sequência

```
input  : Sequência S
output: Número que falta

dif ← constante de crescimento
l ← 0, r ← size
while l < r do
    m ← (l+r)/2
    if S_m > dif * m + S_0 then
        r ← m
    else
        l ← m + 1
return S_m − S_{m−1} == dif ? S_m + dif : S_m − dif
```

A complexidade do algoritmo será a complexidade da busca binária que como demonstrado na questão 1 é $O(log(n))$.

7. Uma maneira de encontrar o maior valor que pode ser cobrado é utilizar um vetor para guardar o maior valor possível de ser adquirido em pedágio para cada ponto intermediário. O próximo valor será calculado em função do valor anterior, assim utilizando programação dinâmica. O valor de V[i] será calculado procurando o maior valor intermediário entre todos os pontos a mais de 10km deste e a menos de 20km.

Algorithm 5: Encontra o maior valor a ser cobrado de pedágios em uma rodovia

input : O vetor V com as distâncias e consequentemente preços, dos pedágios
output: Maior custo

```
memo ← vetor de tamanho |V| inicializado com 0s
memo[0] ← V[0]
for i = 1; i < |V|; i++ do
    j ← 1, max ← 0
    while i − j ≥ 0 and V[i] − V[i − j] < 20 do
        if V[i] − V[i − j] ≥ 10 and memo[max] < memo[i − j] then
            max ← i - j
        j ← j + 1
    memo[i] ← V[i] + memo[max]
return memo[|V|]
```

O algoritmo implementa um laço que vai de 1 até $|V|$, dentro deste analisamos os indices do vetor com distância de no máximo 20, como esse valores são inteiros as operações dentro do loop tem como limite superior 20. Logo a complexidade do algoritmo é simplesmente $O(|V|)$, que é polinomial.