Algoritmos e Estrutura de Dados III

Documentação do Trabalho Prático 3

Ramon Gonçalves Gonze

04 de julho de 2017

1. INTRODUÇÃO

A Rua Pai no Bar é uma rua que possui vários bares e moradores com costumes festivos, principalmente de festas juninas. Todos os donos dos bares moram também nessa rua. Cada bar fica no lado oposto da rua em relação à casa do dono deste bar. Os números dos bares e casas estão ordenados de forma crescente na rua, em ambos os lados. Um lado da rua possui somente números pares e o outro, ímpares.

Os moradores planejam enfeitar a rua com bandeirolas, e para isto, pegarão emprestado as bandeirolas com os donos dos bares. Cada bar irá contribuir com uma linha de bandeirola, e será amarrada uma ponta dela no bar e a outra ponta na casa do dono deste bar. Porém, há uma restrição: as linhas de bandeirolas não podem se cruzar. Cada bar não necessariamente fica em frente a casa do seu dono, um bar pode estar no começo da rua e a casa do dono no final da rua. A regra é que a casa do dono do bar pode estar em qualquer lugar, desde que seja do lado oposto da rua de onde está o bar.

O objetivo deste trabalho prático é encontrar o maior número possível de linhas de bandeirolas que se podem colocar na rua, dado um número específico de bares e casas, cada um com um número. Lembrando que os números dos estabelecimentos estão ordenados de forma crescente em ambos os lados da rua. Foram criados três algoritmos para encontrar a solução do problema, cada qual com uma abordagem diferente: um algoritmo que busca a solução através da **força bruta**, um algoritmo **guloso** e, por fim, um algoritmo que utiliza **programação dinâmica**.

2. SOLUÇÃO DO PROBLEMA

Seguem as três soluções implementadas para o problema:

- **Algoritmo de Força Bruta**

O algoritmo de força bruta consiste em testar todas as opções possíveis, para então selecionar a melhor. Para este problema, as opções foram selecionadas do seguinte modo: Fora construído dois vetores, um de números pares representando um lado da rua - *evenArray* - e outro de números ímpares - *oddArray* - representando o outro lado da rua. Sendo *options* o conjunto de bares e casas (as relações entre bares e seus donos) e *n* a quantidade de bares e casas, o Algoritmo 1 abaixo descreve o funcionamento da força bruta.

Algoritmo 1

```
function bruteForce(options, n)
1       max_flags ← 0
2       odd_index ← 0
3       for i = 1 to 2^n
4               possibilities ← i
5               number_of_flags ← 0
6               while possibilities[j] == 0 do
7                       j ← j + 1
8               while j < n do
9                       even_index ← evenArray[j]
10                      aux ← findCorrespondent(even_index)
11                      if aux > odd_index then
12                              odd_index ← aux
13                              number_of_flags ← number_of_flags + 1
14                              j ← j + 1
15                      else break // Stop the while from the 8th line
16              if number_of_flags > max_flags then
17                      max_flags ← number_of_flags
18      return max_flags
```

O algoritmo acima consiste em criar um vetor auxiliar binário, representado na linha 4 do Algoritmo 1, que indicará se aquela casa ou bar do vetor *evenArray* está naquela solução ou não (1 está, 0 não está). Para cada estabelecimento escolhido no vetor *evenArray*, verificamos o seu correspondente no vetor *oddArray*. Caso ele seja menor que o correspondente do estabelecimento anterior (que está salvo em *odd_index*), significa que uma linha de bandeirola

foi cruzada, logo não é necessário continuar analisando essa opção. Caso ele não seja, atualizamos *odd_index* para esse correspondente, e incrementamos em 1 *number_of_flags*, ou seja, adicionamos essa linha de bandeirola no conjunto solução. Ao final do loop principal, a variável *max_flags* terá armazenado o maior número possível de linhas de bandeirolas.

- **Algoritmo guloso**

O algoritmo guloso consiste em encontrar um conjunto solução $S \subseteq C$, sendo C o conjunto inicial que contém todos os bares/ casas. A construção da solução S é feita adicionando candidatos um a um que tenham a maior probabilidade de pertencerem ao conjunto solução. A estratégia utilizada pelo Algoritmo 2, é de que a escolha seja feita com bares/ casas que estão, desde a menor distância possível, até a maior distância possível do seu complementar (o complementar de um bar é a casa do dono do bar, e o complementar da casa do dono, é o bar).

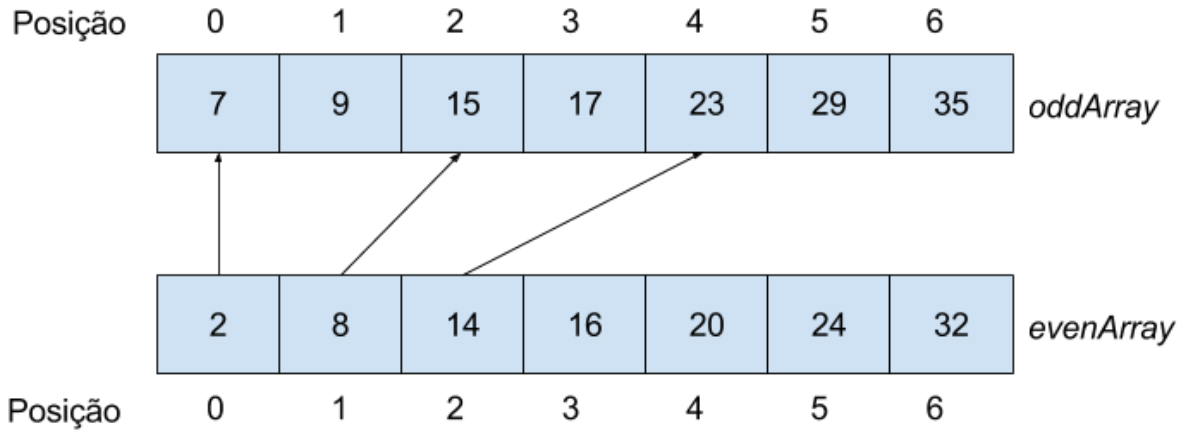

Figura 1: Exemplo de rua que possui um total de 14 bares/ casas (7 em cada lado).

Na Figura 1, observe o par (2, 7). Ambos os números estão nas posições **0** dos vetores, logo a distância de 2 até o seu complementar é **0**. Já os número que compõe o par (8, 15) estão nas posições 1 e 2 respectivamente de seus vetores, sendo então a distância de 8 até o seu complementar é **1**. Aplicando também ao par (14, 23), a distância entre eles é **2**.

Casas (ou bares) que possuem distâncias menores entre ele e seu complementar, intuitivamente, tendem a possuir uma menor quantidade de conflitos, isto é, linhas que se cruzam. Apesar do tempo de execução do algoritmo guloso ser muito menor em comparação

com o algoritmo de força bruta, a solução encontrada pelo algoritmo guloso não é ótima. Há casos em que o número máximo de linhas de bandeirolas encontradas pelo algoritmo é inferior ao verdadeiro máximo de linhas. Este caso geralmente acontece quando a solução para o problema é composto de muitas casas e bares que possuem distâncias grandes entre si. Sendo *options* o conjunto de bares e casas (as relações entre bares e seus donos) e *n* a quantidade de bares e casas, o Algoritmo 2 descreve o funcionamento da solução gulosa.

Algoritmo 2

```
function greedy(options, n)
1       max_flags ← 0
2       S ← ∅
3       for i = 0 to (n-1)
4               for j = 0 to (n-1)
5                       c = checkCandidate(options, j)
6                       if candidate is valid then
7                               S ← S + c
8                               max_flags ← max_flags + 1
9       return max_flags
```

A função *checkCandidate()* recebe como parâmetro um par (uma casa e um bar) e verifica se ele não possui nenhum conflito com os pares que já estão na solução *S*. Se ele não possuir conflitos, ele é um candidato válido, logo é adicionado ao conjunto *S*. A cada candidato incluído em *S*, significa que é mais uma linha de bandeirola adicionada, e ao final do algoritmo, *max_flags* irá conter a solução final.

- **Algoritmo que utiliza programação dinâmica**

A resolução do problema se assemelha à de um problema já conhecido da computação: LIS (*Longest increasing subsequence*). Este problema consiste em, dado um vetor de inteiros X[1...n], qual é o **tamanho** da maior subsequência crescente nele, podendo esta não ser necessariamente contígua. Exemplo:

X = {2, 4, 13, 8, 12, 1, 20}

Maior subsequência possível: 2, 4, 8, 12, 20

Tamanho: 5

Como este problema se assemelha com a quantidade máxima de linhas de bandeirolas na Rua Pai no Bar? Sabemos que os números das casas e bares são ordenados de forma crescente em ambos os lados da rua. Suponha que tenhamos somente um lado ordenado crescentemente, como por exemplo o que possui números pares. A partir desse lado ordenado, coloque cada casa (ou bar) correspondente em frente ao seu complementar. O resultado seria semelhante a esse:

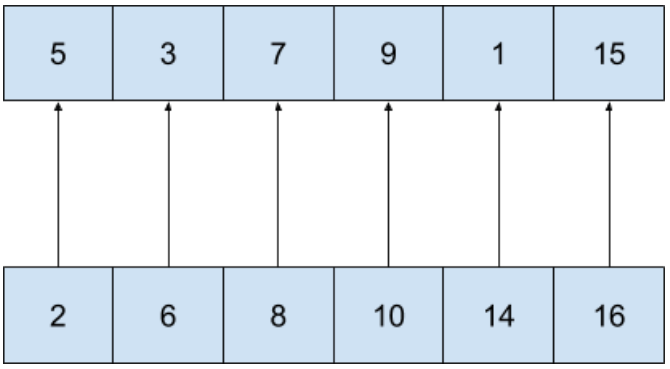

Figura 2: Exemplo para os pares (2,5), (6,3), (8,7), (10,9), (14,1) e (16,15).

Se aplicarmos o algoritmo que resolve o LIS no vetor de números ímpares, encontraremos o maior número de linhas de bandeirolas que não se cruzam. No exemplo da Figura 2, a maior subsequência crescente é (5, 7, 9, 15) ou (3, 7, 9, 15), que possuem tamanho **4**. Se organizarmos os vetores da forma como a rua é estruturada, ou seja, com ambos os lados crescentes, podemos observar que, realmente, o número máximo de linhas de bandeirolas que não se cruzam é **4**, através dos pares de casas/ bares {(2,5), (8,7), (10,9), (16,15)} ou {(6,3), (8,7), (10,9), (16,15)}.

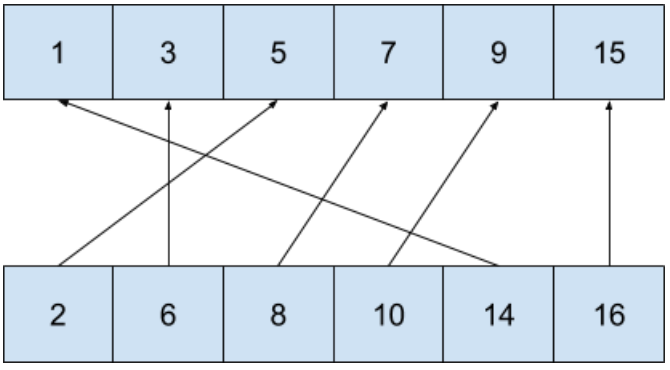

Figura 3: Real estrutura da rua para o exemplo da Figura 2. O número máximo de setas que não se cruzam é 4.

A programação dinâmica está no algoritmo para resolver o problema LIS. Para obter-se um algoritmo que utiliza programação dinâmica, devemos encontrar no problema uma subestrutura ótima. Procuramos, inicialmente, achar o tamanho da maior subsequência crescente em um vetor X de tamanho *n*. Podemos dividir esse problema calculando a maior subsequência crescente para partes do vetor X, ou seja, encontrar essa subsequência até um índice $i \leq n$ do vetor. Logo, podemos encontrar a maior subsequência crescente para *i =1, 2, 3, ..., n*, o que nos levará ao objetivo, que é *LIS(n)*. Escrevendo o exposto acima como uma estrutura recursiva, essa seria:

$$LIS(i) = \begin{cases} 1 & se\ i = 0 \\ max(LIS(j) + 1),\ para\ j = 0, 1, \ldots, i\text{-}1\ e\ X[j] < X[i] & c.c. \end{cases}$$

Encontrada a subestrutura ótima, e determinada a estrutura recursiva do problema, utilizando a estratégia *Bottom-up*, podemos escrever um algoritmo que utiliza programação dinâmica. Este algoritmo se encontra entre as linhas 4 e 13 do Algoritmo 3.

Algoritmo 3

```
function dynamic(options, n)
1       sort(evenArray)
2       sort(oddArray) // According the positions in evenArray (explained in Figure 2)
3       // Execution of LIS algorithm in the oddArray
4       LIS[0] ← 1
5       max_flags ← 0
6       for i ← 1 to n
7               aux ← 0
8               for j ← 0 to i
9                       if oddArray[i] > oddArray[j] AND LIS[j] > aux then
10                              aux ← LIS[j]
11              LIS[i] ← aux + 1
12              if LIS[i] > max_flags then
13                      max_flags ← LIS[i]
14      // End of LIS
15      return max_flags
```

Considerando primeiramente que *LIS(0) = 1*, pois para zero elementos, a maior subsequência é 1, ou seja, a opção de "nenhum elemento". A partir deste, seguindo a equação de recorrência definida, calculamos *LIS(1), LIS(2), ... LIS(n)*. Calculando "de baixo para cima" (*Bottom-up*),

não precisamos recalcular *LIS(j) para j = 0, 1, 2, ..., i-1*, o que reduz a complexidade do algoritmo.

3. ANÁLISE DE COMPLEXIDADE

As análises de complexidade serão feitas sob o número de comparações realizadas entre os vetores *evenArray* e *oddArray*, que possuem os números dos bares e das casas. Sendo *n* o número de casas e bares, seguem as análises de complexidade para cada algoritmo implementado:

- **Algoritmo de Força Bruta**

Primeiramente, os vetores que possuem os números pares e ímpares (vetores *evenArray* e *oddArray*) são ordenados com o *Quicksort*, que possui complexidade $O(n \log n)$. O algoritmo possui um laço que roda 2^n vezes para testar todas as opções possíveis, e possui inserido nele, outros dois laços que rodam, no pior caso, *n* vezes (laços das linhas 6 e 8 do Algoritmo 1). As outras operações do algoritmo são $O(1)$. Portanto, sua complexidade temporal é $O(n \log n) + O(2^n)\ (O(n) + O(n)) = O(2^n n)$. Em relação à complexidade espacial, o algoritmo possui três vetores de tamanho *n*: *possibilities*, *evenArray* e *oddArray*. Logo, sua complexidade espacial é $O(n)$.

- **Algoritmo guloso**

Semelhante ao algoritmo de Força Bruta, primeiramente os vetores *evenArray* e *oddArray* são ordenados em $O(n \log n)$ pelo *Quicksort*. O algoritmo guloso possui dois loops que são executados *n* vezes, que podem ser observados nas linhas 3 e 4 do Algoritmo 2. O loop mais interno executa operações de atribuição, que são $O(1)$, e faz a chamada da função *checkCandidate()*. Esta última, possui somente um loop que é percorrido *n* vezes, e dentro deste, chama a função *findCorrespondent()* que pesquisa sequencialmente o complementar do número, ou seja, é $O(n)$. A complexidade temporal final do algoritmo é $O(n^4) + O(n \log n) = O(n^4)$. O algoritmo faz o uso de quatro vetores, de tamanho *n*, que são os vetores *distances*, *solution*, *evenArray* e *oddArray*. Portanto, a complexidade espacial do algoritmo é $O(n)$.

- **Algoritmo que utiliza programação dinâmica**

A partir do Algoritmo 3, podemos analisar as 3 partes principais do algoritmo: *sort(evenArray)*; *sort(oddArray)* e *LIS algorithm*.

- Para ordenar o vetor de números pares, será utilizado o *Quicksort*, que possui complexidade $O(n \log n)$.
- O vetor *oddArray* é de tamanho *n*. Os elementos são escritos no vetor de forma crescente: primeiro o elemento da posição 0, segundo o da posição 1, e assim consecutivamente até *n-1*. Para encontrar qual elemento colocar em qual posição, é usada a função *findCorrespondent()*, que faz uma busca sequencial no vetor *options* da struct Numbers (uma struct que possui a relação de qual número par é o complementar de um número ímpar). Logo, temos dois loops que são executados *n* vezes, o que torna a complexidade dessa parte $O(n^2)$.
- O algoritmo que resolve o problema da *LIS* possui um loop externo que é executado *n* vezes e um interno que é executado $1, 2, 3, ..., n-1 = n(n+1)/2$ vezes. A complexidade desta última parte fica, portanto, $O(n^2)$.

Por fim, a complexidade temporal final do algoritmo de programação dinâmica é $O(n \log n) + O(n^2) + O(n^2) = O(n^2)$. O algoritmo possui três vetores de tamanho *n*: *evenArray*, *oddArray* e *LIS*. A complexidade espacial do algoritmo é $O(n)$.

O programa principal faz a leitura dos números das casas e bares através da função *readNumbers()*, que possui somente um loop que é executado *n* vezes, e após isto, chama um dos algoritmos para encontrar o número máximo de linhas de bandeirolas. Como a complexidade dos três algoritmos apresentados é superior assintoticamente à função *readNumbers()*, então a complexidade do programa principal será a complexidade do algoritmo escolhido para resolver o problema.

4. AVALIAÇÃO EXPERIMENTAL

Os testes foram feitos em um ambiente Linux utilizando a distribuição Ubuntu 16.04 LTS, em um computador com processador core i3 de 2.1 GHz e 6GB de RAM. Os testes

exaustivos foram realizados com o intuito de verificar qual o comportamento dos algoritmos frente à variação do número de bares e casas.

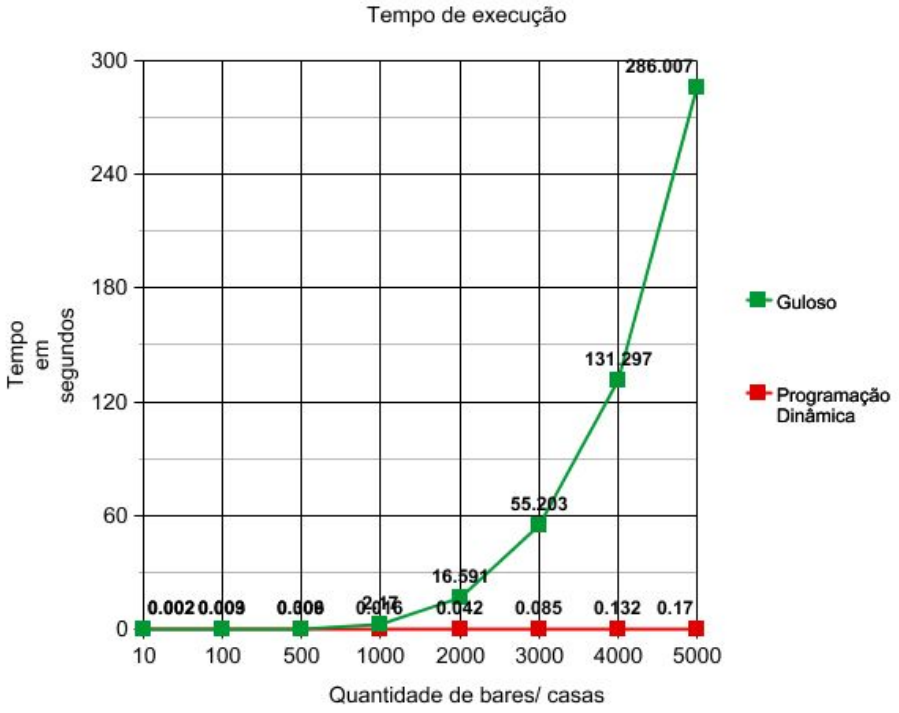

Gráfico 1: Tempo de execução dos algoritmos guloso e que utiliza programação dinâmica.

Pode-se observar no Gráfico 1 que, apesar das complexidades dos dois algoritmos serem polinomiais, $O(n^4)$ e $O(n^2)$, há uma diferença muito grande no tempo de execução a partir de $n = 3000$. Além do algoritmo de programação dinâmica sempre informar a solução ótima - o que não ocorre no algoritmo guloso -, ele extremamente mais eficiente.

Gráfico 2: Tempo de execução dos algoritmos de força bruta e que utiliza programação dinâmica.

Se para a comparação feita no Gráfico 1 encontramos uma discrepância muito grande entre os dois algoritmos, no caso do Gráfico 2, essa discrepância é ainda maior. O comportamento do algoritmo de força bruta que possui complexidade $O(n2^n)$ se mostra ineficiente para $n > 30$. Já o algoritmo de programação dinâmica, possui um comportamento linear para poucos elementos.

Gráfico 3: Tempo de execução do algoritmo que utiliza programação dinâmica, com n variando entre 10 e 10^6 elementos.

O Gráfico 3 nos mostra a eficiência do algoritmo de programação dinâmica. Podemos perceber que para $n = 10^6$, um valor relativamente grande, o algoritmo ainda executa em tempo hábil. Para entradas maiores ou iguais a 10^7, o algoritmo já não é eficaz.

5. REFERÊNCIAS

ZIVIANI, N. Projeto de Algoritmos. Terceira edição. São Paulo. 2013. 639p.