Algoritmos e Estrutura de Dados III
Paradigmas

Ingrid Rosselis Sant'Ana da Cunha

Julho 2017

1 Introdução

O último trabalho da disciplina Algoritmos e Estrutura de Dados III consiste em resolver um dado problema com três estratégias diferentes, sendo elas: *força bruta*, *guloso* e *dinâmico*. O problema fala de uma rua onde serão penduradas bandeirolas de donos de bares – onde cada dono mora do outro lado da rua, não necessariamente em frente ao seu estabelecimento – com a seguintes condições:

1. A bandeirola de um dono deve ser pendurada com uma ponta no bar e outra na casa da pessoa;
2. As linhas das bandeirolas não devem se cruzar.

2 Contextualização

2.1 Problema

O problema é, na verdade, uma variação do problema de **Subsequência Crescente Máxima** (do inglês *LIS - Longest Increasing Subsequence*) chamado **Contrução de Pontes**. Neste problema, existem n cidades divididas em pares separados por um rio, onde cada metade da dupla fica em uma margem, e deseja-se construir o máximo de pontes possíveis para ligar as cidades de cada par, sem que duas pontes se cruzem.

Já o *LIS* é mais simples: encontre uma subsequência em um vetor de números não ordenado que não seja necessariamente contínua, mas onde cada elemento inserido na subsequência seja maior que os anteriores . Ao final, deve-se obter a maior subsequência possível.

2.2 Força Bruta

Um algoritmo de **Força Bruta**, ou Tentativa e Erro, é uma técnica de solução de problemas trivial que consiste em enumerar todos os possíveis candidatos a solução e checar se cada um deles satisfaz o problema.

Possui implementação simples e solução sempre exata, porém testa todas as possibilidades para um candidato, isto é, sua complexidade é dada aproximadamente por p^n, onde p é o número de possibilidades e n é o tamanho da entrada. Por conta de sua complexidade temporal exponencial (na maioria dos casos), algoritmos de Força Bruta são normalmente usados quando o tamanho do problema é limitado e pequeno e quando uma solução exata é mais desejável que uma solução ótima, isto é, prioriza-se precisão em detrimento de desempenho.

2.3 Guloso

Um algoritmo **Guloso** é uma técnica de solução que tenta resolver o problema fazendo a escolha localmente ótima em cada iteração com a esperança de encontrar um ótimo global. Ele é amplamente utilizado quando deseja-se encontrar uma solução próxima à exata, mas em tempo ótimo, isto é, geralmente polinomial.

2.4 Programação Dinâmica

Um algoritmo que segue o paradigma de **Programação Dinâmica** (*PD*) tenta encontrar a resposta para um problema com uma espécie de solução iterativa inteligente, próxima a uma recursão que utiliza uma tabela para guardar passos intermediários. Assim como em um algoritmo recursivo, cada instância do problema é resolvida a partir de instâncias menores, subproblemas. O que o difere de uma estratégia recursiva é que as soluções dadas pelos subproblemas são memorizadas para serem reutilizadas na execução global do algoritmo, evitando que a roda seja "reinventada"em outros subproblemas.

Para que o paradigma possa ser aplicado, é necessário que o problema tenha duas características básicas:

- *Subestrutura ótima ou estrutura recursiva*: a solução ótima de toda instância do problema deve conter soluções ótimas de subproblemas;
- *Superposição de problemas*: acontece quando um algoritmo recursivo reexamina o mesmo problema muitas vezes. A superposição de problemas é a base do algoritmo recursivo, mas é tipicamente ineficiente porque refaz, muitas vezes, a solução de cada subproblema.

Um algoritmo de *PD* é construído tomando-se um algoritmo de recursão e montado uma tabela para ele com as soluções locais ótimas dos subproblemas, evitando que sejam recalculadas.

2.5 Guloso x Programação Dinâmica

A diferença entre um algoritmo que segue o paradigma de *PD* em relação a um algoritmo guloso é que o primeiro pode basear sua escolha na solução de instâncias menores, enquanto o segundo sempre tenta escolher a melhor solução naquela execução e sempre começa pelos maiores subproblemas.

3 Implementação

O projeto para este trabalho possui 10 arquivos com intuito de modularizar o código, são eles:

- 2 arquivos para funções de uso geral;
- 2 arquivos para funções que implementam o algoritmo de Força Bruta;
- 2 arquivos para funções que implementam o algoritmo Guloso;
- 2 arquivos para funções que implementam o algoritmo baseado em *PD*;
- 1 arquivo para função *main*;
- 1 arquivo de *Makefile*.

O grande foco deste trabalho são os algoritmos específicos para solução de problemas. Logo, os arquivos com funções de uso geral basicamente preparam o terreno para que um dos algoritmos seja executado.

A estrutura utilizada neste trabalho é uma adaptação do *pair* que existe em C++, ou seja, é uma estrutura que contém dois inteiros, sendo eles o número do bar e o número da casa.

Um pseudo-código aproximado do funcionamento geral do projeto seria:

```
1:  procedure TP3:
2:      recebe seletor de algoritmo
3:      recebe quantidade de pares
4:      cria estrutura
5:      recebe pares
6:      preenche estrutura
7:      if seletor ← g then
8:          executa algoritmo Guloso
9:      if seletor ← d then
10:         executa algoritmo baseado em PD
11:     if seletor ← b then
12:         executa algoritmo de Força Bruta
13:     return solução
```

Todos os algoritmos específicos são baseados no *LIS*. Para que eles funcionem, existem algumas alterações que devem ser feitas na entrada. São elas:

- Manter os índices pares e ímpares em lados separados. Foi pré-definido que os índices pares são bares e ímpares são casas, e na própria estrutura *pair* os inteiros são denominados como par ou ímpar;
- Para que o *LIS* possa ser aplicado a este problema, um dos lados deve ser ordenado em ordem crescente e o *LIS* deve ser obtido no outro lado. Assim, também por pré-definição, o lado de índices pares é ordenado usando a função *qsort()* e todos os algoritmos agem sobre o lado de índices ímpares.

3.1 Força Bruta

A estratégia abordada para a implementação do algoritmo de Força Bruta é o clássico algoritmo de recursão que testa todas as possibilidades para cada elemento da entrada. Para cada índice ímpar do par, são consideradas as possibilidades de que ele seja incluído na solução e de que ele não seja. Caso ele seja, aumenta-se um no valor total do *LIS* encontrado até aquele ponto. O algoritmo "queima"elemento a elemento do vetor até que todos tenham sido visitados e suas possibilidades consideradas. Segue pseudo-código usado para implementar o algoritmo de Força Bruta:

```
1:  procedure LISFB(A[0...n], anterior, esquerda, direita):
2:      if esquerda > direita then
3:          return 0
4:      else
5:          if A[esquerda] ≤ anterior then
6:              return LISFB(A[0...n], anterior, esquerda + 1, direita)
7:          else
8:              x ← LISFB(A[0...n], anterior, esquerda + 1, direita)
9:              y ← LISFB(A[0...n], A[esquerda], esquerda + 1, direita)
10:             return max(x, y)
```

3.2 Guloso

O algoritmo guloso implementado neste trabalho basea-se em um importante resultado do método de ordenação *Patience Sort*: o número de pilhas final obtido é o *LIS* para aquela sequência passada. Vamos entender melhor o que isso significa e como funciona:

O *Patience Sort* é um algoritmo de ordenação baseado no jogo paciência. Ele começa com um *deck* de cartas em ordem aleatória. No decorrer de sua execução, o algoritmo pega as cartas uma a uma e as coloca em pilhas baseado na regra de que **uma carta de menor valor deve ser colocada em cima de uma de maior valor ou colocada em sua própria pilha**. O objetivo é terminar a execução com o menor número de pilhas, sempre tentando colocar as cartas na pilha mais à esquerda possível.
Por exemplo, suponha que temos o *deck* **8 3 7 9 2 5 4 1 10 6**. O algoritmo guloso agirá da seguinte forma:

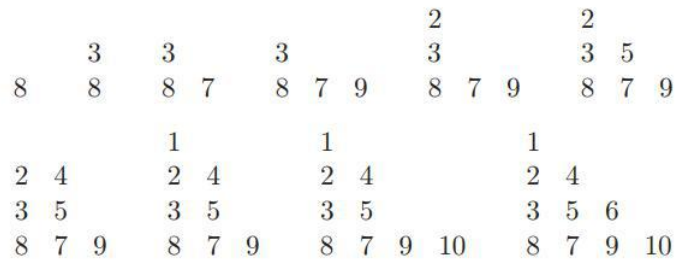


Figura 1: Execução do *Patience Sort*

3.2.1 Prova

Seja Π um *deck* de cartas enumeradas $1, 2, ..., n$. Podemos provar que usando o algoritmo guloso do *Patience Sort* obtemos pelo menos o *LIS* de Π em pilhas. Seja $i_1 < i_2 < ... < i_k$ uma subsequência crescente em Π. Então $\Pi(i_{j_{+1}})$ deve sempre estar em uma pilha mais à direita de $\Pi(i_j)$ já que uma carta só pode ser colocada em cima de outra se possuir menor valor. Portanto, qualquer estratégia válida irá resultar em, no mínimo, *LIS* pilhas.

Agora usando o algoritmo para achar uma instância de uma subsequência crescente, podemos mostrar que temos no máximo o *LIS* em número de pilhas. Toda vez que uma carta c é colocada em uma pilha i e não é a primeira da pilha, desenhe uma seta de c para a carta do topo d na pilha à esquerda de i. É sabido que $d < c$ pois, caso contrário, c estaria em cima de d. Se nós temos k pilhas, podemos achar a carta do topo da a_k pilha mais à direita. Seguindo sua seta estamos criando a subsequência crescente de Π. Logo, nós temos no máximo *LIS* pilhas.

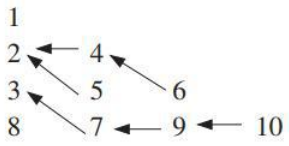


Figura 2: Prova de que o algoritmo possui *LIS* pilhas

Segue abaixo o pseudo-código usado para implementar o algoritmo guloso:

```
1: procedure LISG(A[0...n], n, conjunto de pilhas):
2:     insere o primeiro elemento do vetor na primeira pilha
3:     for 1..n do:
4:         seleciona subparte do conjuntos de pilhas e encontra meio
5:         if elemento_atual > pilha[meio] then
6:             altera índice da esquerda
7:         else
8:             altera índice da direta
9:         if esquerda maior que o número de pilhas anterior then
10:            nova pilha foi gerada, incrementa-se o LIS
11:            insere elemento atual na pilha mais à esquerda
12:    return LIS
```

Pode-se verificar que a estratégia utilizada pelo *Patience Sort* é gulosa baseado no pseudo-código. A cada vez que o algoritmo insere nas pilhas, ele procura uma solução local ótima para o problema, mas não guarda histórico de outras soluções. É claro que, deve-se ressaltar, o pseudo-código utilizado aqui é referente ao problema de encontrar o maior *LIS* e vem de um efeito colateral do algoritmo possibilitado pela estratégia gulosa.

3.3 Programação Dinâmica

A implementação do algoritmo dinâmico deste trabalho também se baseia em encontrar o *LIS* de uma sequência. Ele foi feito sobre o algoritmo recursivo do Força Bruta, mas utiliza a tabela que define programação dinâmica (ou memória, como será chamada nesse subseção) e algumas otimizações. O algoritmo basicamente confere cada elemento da lista e decide se ele é um candidato a fazer parte da solução. Quando um novo menor elemento é encontrado no vetor, ele é um candidato potencial para começar uma nova subsequência. Por isso, utiliza os seguintes critérios:

1. Se o elemento atual é o menor entre os candidatos que se encontram nas posições finais das listas, podemos começar uma nova lista de tamanho 1;

2. Se o elemento atual é o maior entre os candidatos que se encontram nas posições finais das listas, devemos clonar a maior lista e extendê-se com o elemento atual;

3. Se o elemento atual não encaixou em nenhum dos itens anteriores, devemos encontrar a lista com o maior elemento que é menor do que o atual e extendê-la com o atual. Todas as outras listas de mesmo tamanho devem ser descartadas.

Segue abaixo o pseudo-código usado para implementar o este algoritmo:

```
1: procedure LISPD(A[0...n], n, memória):
2:     insere o primeiro elemento do vetor na primeira posição da memória
3:     for 1..n do:
4:         if atual < menor_memoria then
5:             menor_memoria ← atual
6:         else
7:             if atual > maior_memoria then
8:                 memoria++ && maior_memoria ← atual
9:             else
10:                procura memória que possui maior_memoria < atual
11:                substitui
12:    return LIS
```

O algoritmo baseado em *PD* possui uma estrutura recursiva que pode ser representada por uma recorrência, e esta pode ser traduzida em um algoritmo recursivo.

Seja $LIS(i, j)$ o comprimento da maior subsequência de $A[j..n]$, com todos seus elementos maiores que $A[i]$. A partir dessa definição, podemos definir a relação de recorrência do algoritmo baseado em *DP* como:

$$\text{LIS(i,j)} = \begin{cases} 0, & se \;\; j > n \\ LIS(i, j+1), & se \;\; A[i] \geq A[j] \\ max(LIS(i, j+1), LIS(j, j+1)), & caso \;\; contrario \end{cases}$$

4 Análise de Complexidade

4.1 Espacial

O projeto só precisa armazenar um vetor de n pares durante toda sua execução, somando um total de $2n$ elementos armazenados. Logo, sua complexidade espacial é linar em todos os casos:

$$O(n) \tag{1}$$

4.2 Temporal

Com exceção dos algoritmos específicos, o procedimento que contém um custo maior de complexidade é o *qsort*() usado para ordenar a entrada. Assim a complexidade do programa é dada pela complexidade da função de ordenamento.

Melhor Caso:

$$O(n) \tag{2}$$

Caso Médio:

$$O(n \cdot \log n) \tag{3}$$

Pior Caso:

$$O(n^2) \tag{4}$$

Onde n é o tamanho da entrada.

4.2.1 Força Bruta

O algoritmo de Força Bruta testa todas as possibilidades para cada elemento da entrada. No caso deste trabalho, cada elemento pode ou não ser incluído na solução final. Logo, existem 2^n possibilidades testadas, o que gera um tempo exponencial.

Melhor, Pior e Médio Casos:

$$O(2^n) \tag{5}$$

4.2.2 Guloso

Para verificar a complexidade do algoritmo guloso, devemos lembrar que seu intuito é a ordenação de um vetor. O algoritmo divide os elementos em pilhas e, teoricamente, como a solução se baseia apenas no número de pilhas, nosso trabalho estaria concluído. Mas na ordenação, após a separação em pilhas, os elementos são reunidos de forma que ficam totalmente ordenados. O algoritmo passa por cada elemento do vetor uma vez, então tem complexidade $O(n)$ e divide os elementos em grupos, o que gera a complexidade $O(logn)$ para visitar um grupo. Neste trabalho, o custo relativo a $O(logn)$ vem da pesquisa binária que é realizada internamente no algoritmo

Melhor, Pior e Médio Casos:

$$O(n \cdot \log n) \tag{6}$$

4.2.3 Programação Dinâmica

O algoritmo baseado em Programação Dinâmica percorre o vetor uma vez apenas, custo de $O(n)$, mas utiliza pesquisa binária para encontrar a lista desejada, com custo $O(logn)$. Logo, seu custo total é $O(nlogn)$.

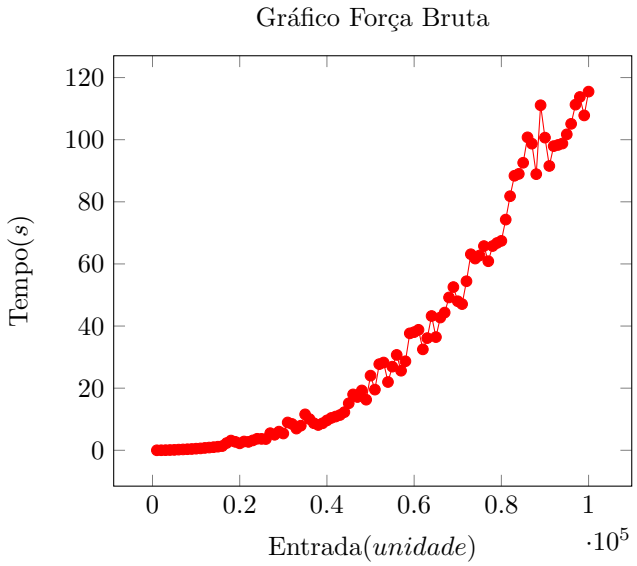
Melhor, Pior e Médio Casos:

$$\theta(n \cdot \log n) \tag{7}$$

5 Avaliação Experimental

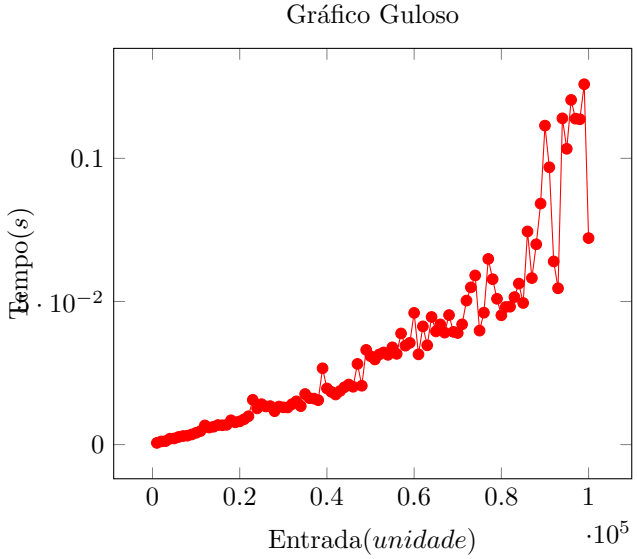
Para gerar os casos de teste, foi utilizado um gerador produzidor por um colega de classe. Nele foram gerados 100 casos de teste para cada estratégia. Os teste começam com um tamanho de entrada 1000 e terminam com 100000, incrementando de 1000 em 1000. O valor teto utilizado baseou-se no fato de que testes maiores permitem a melhor visualização dos gráficos e que o valor de 100000 foi aproximadamente o limite estipulado pelos monitores da disciplina.

5.1 Força Bruta



Pode-se verificar pelo gráfico que o tempo aumenta exponencialmente de acordo com a entrada e explode mais ou menos a partir de $5 \cdot 10^4$.

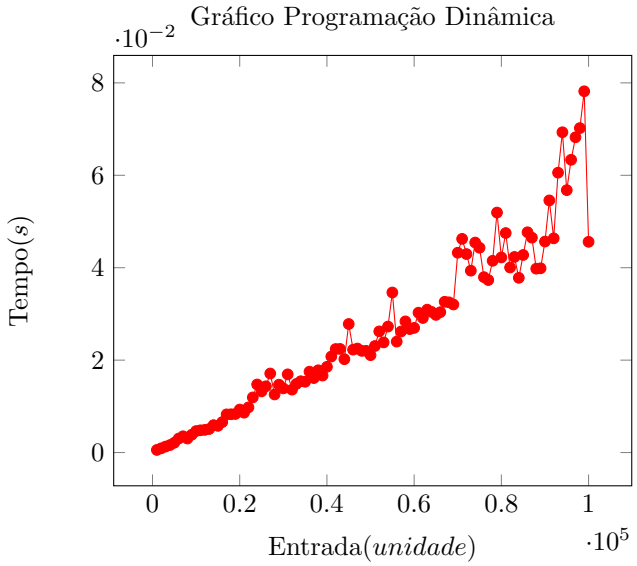
5.2 Guloso



O algoritmo guloso é, baseando-se apenas nos gráficos apresentados aqui, o de melhor velocidade de execução. Os picos que existem depois de $9 \cdot 10^4$ podem ter sido causados por variações na entrada que desfavoreceram o algoritmo.

Nota-se que até um certo ponto o gráfico parece linear, porque a complexidade linear pesa mais que a logarítmica no início.

5.3 Programação Dinâmica



O algoritmo que utiliza o paradigma de Programação Dinâmica ficou para trás nos testes em relação ao algoritmo guloso. Podemos assumir que o guloso foi mais eficiente para este caso, desde que fique claro que as entradas são aleatórias, o que pode influenciar o resultado final.

6 Conclusão

O objetivo deste trabalho é implementar três algoritmos baseados em grandes estratégias de programação. De modo geral, os algoritmos tiveram funcionamentos de acordo com o esperado, atendendo aos requisitos pré-especificados e obtendo resultados satisfatórios.

Como último trabalho prático da disciplina, mais uma vez o entendimento acerca de tópicos específicos de programação foi reforçado como a escolha entre estratégias de programação, os diferentes algoritmos que poderiam ser usados, as diferentes formas de resolver o mesmo problema, as diferenças de desempenho das estratégias e as diferenças entre Algoritmos Gulosos em relação aos que se baseiam em Programação Dinâmica.