TP3 – Legado da Copa

Nome: Henrique Eustáquio Lopes Ferreira
Matrícula: 2015068990

1 Introdução

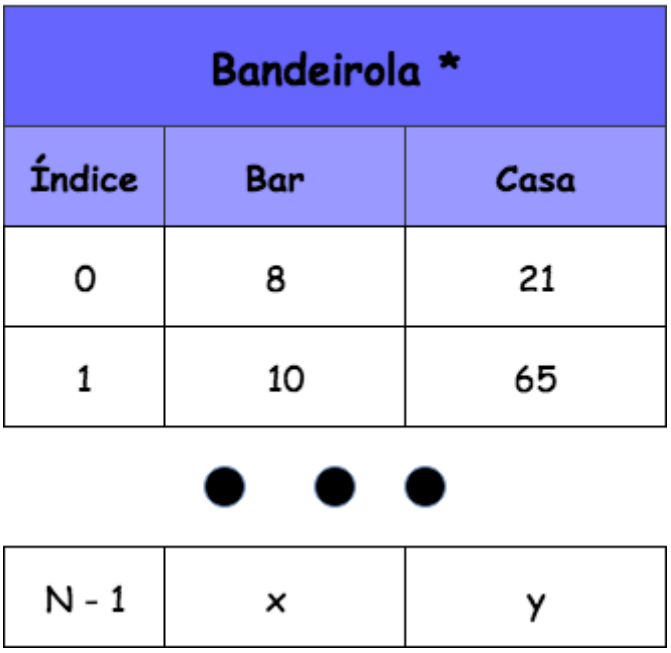
A Rua Pai no Bar, objeto central da situação-problema, é estruturada de forma que de um lado, há bares, enquanto do outro, somente as casas de seus respectivos donos. Também deve-se assumir que a numeração da rua é feita de maneira que de um lado os números são pares e no lado oposto, ímpares. Tais números seguem em ordem crescente em ambos os lados da rua e na mesma direção.

Precisa-se enfeitar a Rua Pai no Bar para a copa da Rússia e então, nesse cenário, o que deseja-se fazer é, dada uma tabela (provinda de um dos moradores, Sami) com os números dos bares e das casas de seus respectivos donos, pendure-se o máximo de bandeirolas possível, de acordo com as seguintes condições: cada bar contribui com uma linha da bandeirola, sendo pendurada do bar à casa do dito dono e as linhas não podem se cruzar. Como não existe necessidade de um bar estar logo à frente da casa de seu proprietário, pode-se lidar com essa questão a partir da maneira com a qual trata-se os dados dessa tabela de entrada e do processamento posterior que se faz com os mesmos, relativos aos números das residências e bares. Logo, consideremos os números como o dado bruto relevante para toda a análise do problema, que por sua vez partirá da maneira como eles são ordenados, como veremos na seção a seguir.

2 Solução do problema

2.1 Entendendo a entrada

Preliminarmente, já foi necessária a inferência de um Tipo Abstrato de Dados que fosse capaz de representar com exatidão a tabela informada na entrada. Daí, optou-se por um TAD simples que possui tão somente um par de inteiros: o número do bar e o número da casa do respectivo dono. Esse TAD foi chamado de bandeirola, a fim de representar legivelmente a situação hipotética do problema de amarrar-se uma bandeirola de um bar a uma casa. Exemplificando:



Assim, alocando-se memória para essa estrutura, de N posições conforme a entrada, trivialmente temos um vetor que é capaz de armazenar os dados de interesse passados de forma segura e suficientemente eficiente para o processamento posterior.

2.2 Trabalhando com o TAD

Partindo da *struct* bandeirola, encontra-se o problema de ela não corresponder necessariamente à maneira com a qual a rua é organizada na prática, pelo enunciado do problema, que não nos garante que um bar está à frente da casa do seu dono. Mas, também temos mais duas informações sobre a Rua Pai no Bar: como qualquer outra, segue uma ordenação crescente e de um lado se encontram números ímpares enquanto do outro, pares.

Dessa forma, é intuitivo encarar antes de mais nada dois primeiros subproblemas: separar aritmeticamente os bares das casas e ordenar a rua conforme um desses itens. Isso porque o problema não está em definir precisamente o caminho da bandeirola, e sim o número máximo de linhas que pode-se pendurar, desde que saindo de um bar e chegando à casa de seu dono, condição já satisfeita pela estrutura de pares, TAD bandeirola. Assim, como pré-processamento entendeu-se enquanto necessidades a separação de números pares e ímpares e o uso de um método de ordenação. Dessa maneira chega-se a um problema de computação famoso, o tamanho da Longest Increasing Subsequence (LIS), detalhado melhor mais adiante nesta seção.

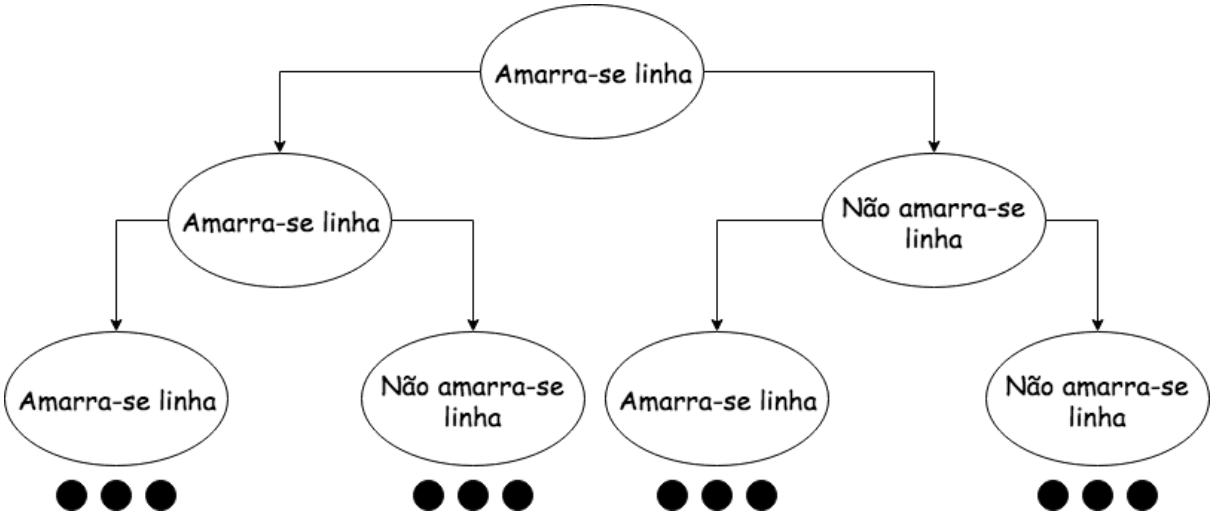
Prosseguindo, assumiu-se que os bares devem ser pares e as casas, por sua vez, ímpares (como não faz diferença para o cálculo do tamanho da LIS). Isso se faz em tempo de leitura e temos como resultado um “vetor de bandeirolas” correspondente. Logo após, o problema de ordenação foi abordado por meio do método QuickSort com partições de Hoare (no qual seleciona-se sempre o pivot como o primeiro elemento). Lembrando que os pares são ordenados a partir dos números dos bares então, agora sim, temos um “sub-vetor” não ordenado de números de casas sobre o qual devemos calcular o tamanho da maior subsequência crescente.

2.3 Longest Increasing Subsequence - LIS

O problema da maior subsequência crescente pode ser visto de maneira simples: dada um *array* não ordenado de inteiros, qual a maior subsequência que pode ser formada? No caso deste trabalho, precisa-se não da subsequência em si, mas sim do tamanho da mesma. Como já visto, a LIS deve, então, ser gerada a partir do “sub-vetor” não ordenado de números de casas, utilizando-se de três abordagens diferentes: força bruta, algoritmo guloso e programação dinâmica, cujas análises assintóticas constam na próxima seção. Por hora, descrevamos o funcionamento de cada uma delas.

2.3.1 Força Bruta

De implementação mais simples, o algoritmo de força bruta considera todas as possibilidades de uma LIS. Na prática, a cada iteração recursiva tem-se uma árvore de possíveis escolhas/árvore de decisão que se comporta similarmente ao seguinte esquema:



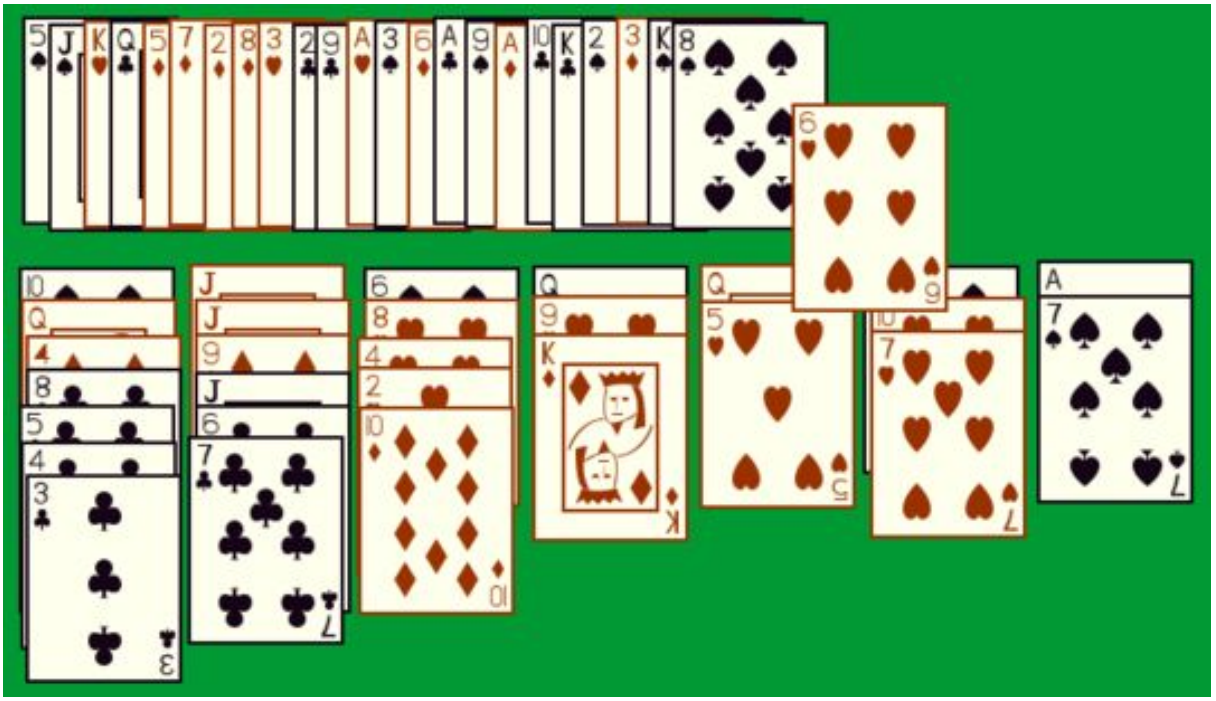
Em outras palavras, para cada linha da bandeirola de um bar a uma casa, consideramos amarrar a linha ao próximo bar, caso elas não se cruzem, e, ao mesmo tempo, não amarrá-la, computando o valor da LIS em todo e qualquer caso.

Basta, por fim, que decida-se qual o maior número entre a soma das possibilidades de prosseguir com a LIS ou não e tem-se o valor do tamanho da LIS.

2.3.2 Algoritmo Guloso

O método guloso utiliza-se da lógica do *Patience Sort*. Análogo ao raciocínio do jogo de cartas Paciência, de onde deriva seu nome, este algoritmo de ordenação parte de pilhas de cartas em ordem. Basicamente, ao montar-se uma pilha no jogo, procura-se que ela contenha uma subsequência crescente das cartas do baralho.

Sobre o funcionamento do algoritmo, iniciamos uma pilha de números de casas com a primeira casa possível. Daí, precisamos garantir que haverá uma pilha correspondente à LIS, o que é feito da seguinte forma: a cada iteração a pilha é atualizada com o número da casa atual, sendo que inicia-se da casa de menor número, sempre. Em outras palavras, consideramos em que posição o número da casa atual será alocado na pilha, sendo que excluímos sempre os elementos maiores, separando-os em “outras pilhas”. Como isso ocorre sempre que encontramos uma casa candidata, o tamanho da pilha corresponde ao tamanho da LIS, sendo que não é necessário “voltar atrás” no algoritmo.



Obs.: nota-se na figura o comportamento do *Patience Sort*: a carta 6 de copas, por ser menor, é acrescentada na pilha de forma que o 7 de copas “dá lugar” a ela, no raciocínio do algoritmo para o tamanho da LIS.

2.3.3 Programação Dinâmica

Aqui parte-se do fato de que a LIS até um determinado elemento independe dos elementos seguintes. Calculando-se esse subproblema, chega-se ao problema principal, da maior subsequência em todo o *array*.

Dessa forma percorre-se, para cada elemento (número da casa), todos os elementos anteriores e atualiza-se um vetor auxiliar com o valor máximo de números de casas menores do que o elemento fixado até o momento (o que significa na prática o valor correspondente no índice do vetor auxiliar). Em outras palavras, supõe-se, para cada elemento, a possibilidade de que ele esteja em cada LIS, garantindo que a nova sequência formada também é crescente e a maior possível.

3 Análise de complexidade

Nesta seção partiremos do princípio de analisar em alto nível e objetivamente somente aquilo crucial para a solução do problema de encontrar o tamanho da LIS para o “sub-vetor” de números das casas da rua, conforme já abordado. Dessa forma, trataremos da função de QuickSort, essencial para o pré-processamento, e das funções relacionadas a cada método de solução da seção 2.3 acima.

3.1 Hoare QuickSort

A partir da estratégia de divisão e conquista, conhece-se que a recursividade do QuickSort garante a ordenação do vetor pelo rearranjo dos elementos a partir da escolha do pivot. Quando o pivot divide a lista de maneira desbalanceada (1 elemento em uma partição e n-1 elementos em outra) para cada chamada da recursão, temos o pior caso, com complexidade O(n^2), já que trata se a cada chamada de um vetor de tamanho vetor anterior - 1. No melhor caso, produzimos partições de tamanho não maior do que n/2, sendo a equação respectiva de recorrência $T(n) \leq 2T(n/2) + \theta(n)$ que, pelo Teorema Mestre nos mostra complexidade O(n*log(n)).

3.2 Força Bruta

Porque a estratégia de força bruta forma, como demonstrado, forma uma árvore de decisão que, a cada nó, sempre considera as possibilidades de se ter ou não o número na LIS, a complexidade de tempo do algoritmo de força bruta corresponde ao tamanho da sua árvore recursiva, ou seja O(2^n). De outra maneira, a cada passo recursivo tem-se duas possibilidades, sendo que o algoritmo executa em cada um dos n passos dessa mesma maneira.

Claramente e como visto melhor nos testes, é a abordagem mais lenta dentre as três requeridas no trabalho.

3.3 Guloso

Devemos considerar que o algoritmo guloso possui um loop que percorre todos os n elementos. Para cada um desses, a opção mais cara é justamente verificar uma casa candidata à LIS, ou seja, uma casa a partir da qual deva-se excluir valores maiores. A decisão do índice onde deve-se adicionar essa casa se dá através de um algoritmo de busca binária, cujo sistema é idêntico à relação de particionamento do QuickSort já implementado, porém com o *pivot* escolhido como elemento do meio.

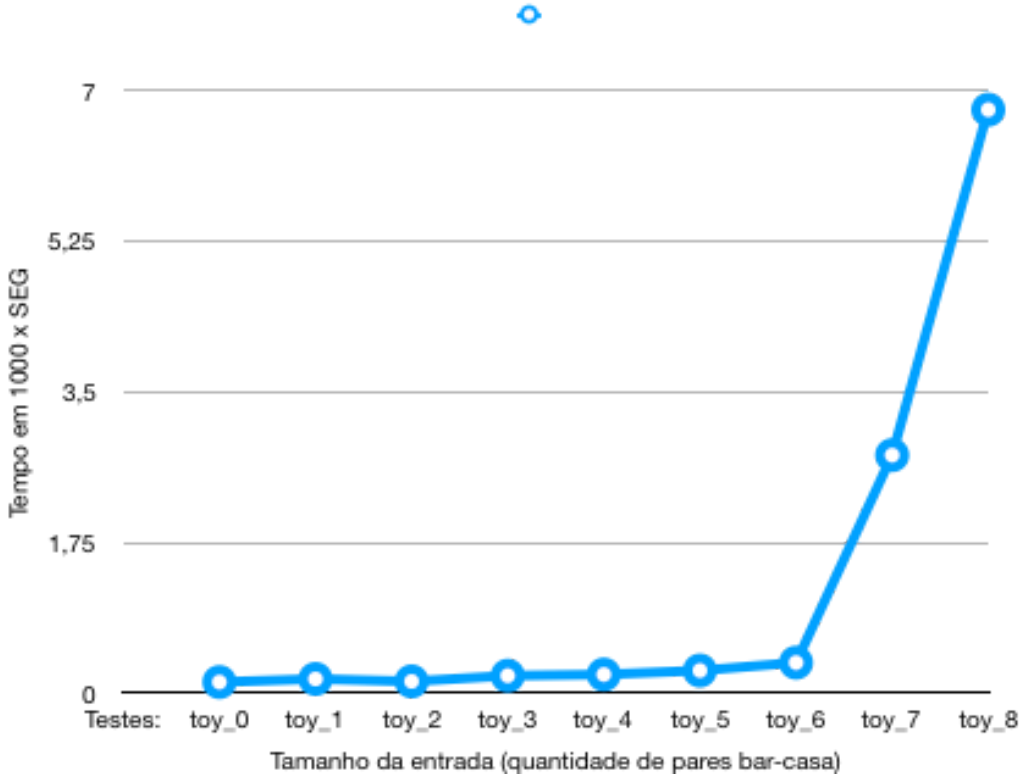
Enfim, a complexidade é dada, para cada iteração do loop, pela necessidade eventual da busca binária, O(log(n)). Assim, o método guloso dá a resposta em tempo O(n*log(n)).

3.4 Programação dinâmica

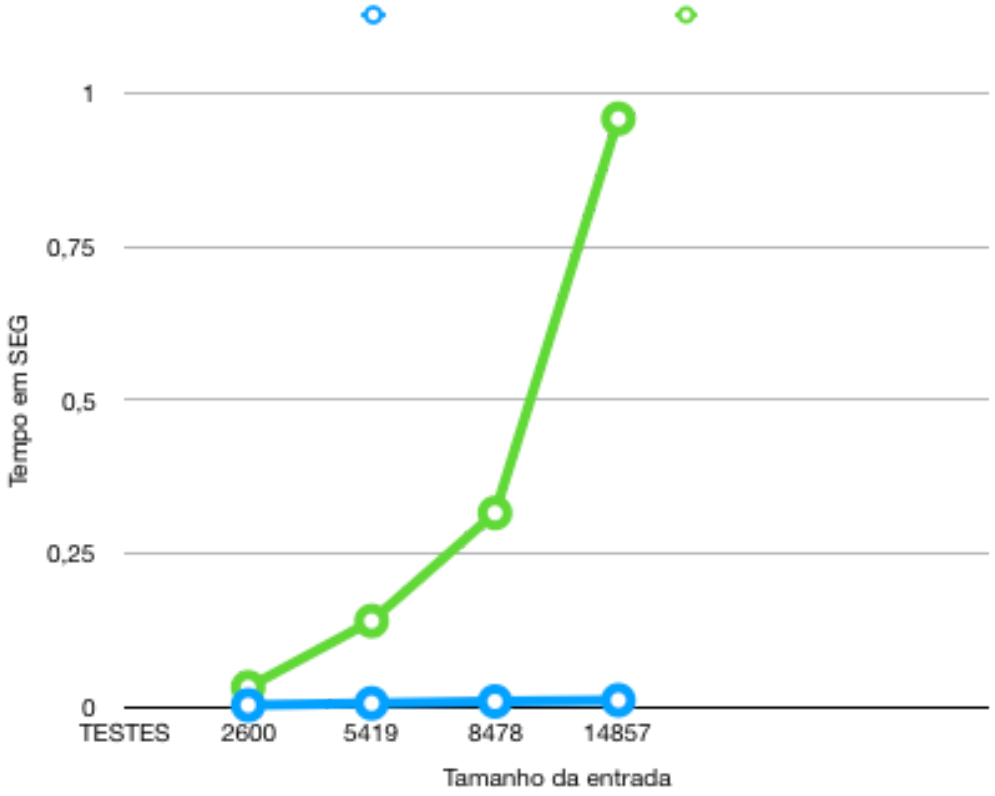
Finalmente, como devemos considerar cada subproblema, ou seja, cada LIS até um elemento de índice i, o algoritmo de programação dinâmica possui dois *loops* aninhados. Levando em conta a operação relevante enquanto escolher os valores máximos, conforme descrito em seu funcionamento, sabe-se que essa requisição, O(1), é feita em O(n^2), já que cada *loop* roda n vezes.

4 Avaliação experimental

Tendo em mente que o algoritmo de força bruta possui complexidade exponencial, para avaliá-lo optou-se pelos testes toy normais disponibilizados no moodle da disciplina. Assim para comparação de tempo e verificação da exponencialidade assintótica, gerou-se o gráfico abaixo:



Nos casos de solução por estratégia gulosa e força bruta, procurou-se não só casos de teste mais complicados como também comparar os tempos de execução. Assim, gerou-se o gráfico abaixo:



Legenda:

- Azul: guloso, O(n*log(n))
- Verde: dinâmico, O(n^2)

Percebe-se a grande discrepância matemática entre o logaritmo da entrada e seu valor linear na multiplicação por n. Em casos de teste maiores, com uma entrada da ordem de 100.000 pares casa-bar, a DP estoura facilmente os 40 segundos. Outra observação interessante é o fatídico comportamento da curva verde, muitíssimo semelhante à função quadrática à qual se refere.

Vale lembrar a média tirada dos tempos de execução a fim de evitar o *cold start*. Então, cada teste foi computado 5 vezes a fim de precisar seu tempo de execução.

5 Observações finais

O programa foi compilado em ambiente Linux Ubuntu 14.04 LTS 64 bits com 4GB de memória RAM disponíveis e um processador Intel® CoreTM i3-2350M CPU @ 2.30GHz.

Arquivos de código (*headers* devidamente comentados descritivamente):

- tp3.c (função *main*)
- trata_entrada.h
- trata_entrada.c (responsável pelo pré-processamento e leitura de dados)
- bandeirola.h
- bandeirola.c (responsável pelo processamento)

A compilação se deu através da linha:

- gcc -g -Wall -Wextra -Werror -std=c99 -pedantic trata_entrada.c tp3.c fila_bfs.c -o tp3

Todos os testes no valgrind foram efetuados com o comando:

- valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./tp3 < tp3.in

Obs.: tp3.in corresponde ao arquivo de entrada padrão, cujo conteúdo foi alterado diversas vezes conforme o caso de teste para verificar a corretude da resposta.