UNIVERSIDADE FEDERAL DE MINAS GERAIS

Trabalho Prático 3

Legado da Copa

Algoritmos e Estruturas de Dados III - 2017/1

Deiziane Natani da Silva

Introdução

Legado da Copa

O trabalho prático consiste em praticar os conceitos sobre paradigmas estudados em aula. O objetivo é ajudar os moradores de uma rua a pendurar o maior número de linhas de bandeirola dadas as seguintes condições:

- Na rua existem bares e casas, cada bar tem um dono que mora do lado oposto da rua.
- Dada a chegada da copa da Rússia, cada bar vai contribuir com uma linha de bandeirola.
- As bandeirolas serão penduradas da seguinte maneira: Uma ponta da linha no bar e a outra ponta na casa do dono do bar.
- Como a casa do dono do bar pode ficar em qualquer posição da rua oposta, pode acontecer algum caso em que as linhas iram se cruzar. Esse caso não é permitido.
- Um lado da rua é composto apenas por números ímpares e outro lado apenas por números pares que seguem a ordem crescente, como na numeração usual.

A solução do problema possui três estratégias: força bruta, algoritmos gulosos e programação dinâmica.

Solução do Problema

Programação Dinâmica

O problema apresentado pode ser visto como o problema Building Bridges que, por sua vez, é uma variação do problema Longest Increasing Subsequence (LIS). O Building Bridges é descrito da seguinte forma:

- Considere um mapa 2-D com um rio horizontal que passa por seu centro. Existem n cidades no lado sul do rio com coordenadas de a(1)...a(n) e n cidades no lado norte com coordenadas b(1) ... b(n). Você quer conectar tantos pares de cidades norte-sul quanto possível com pontes, de modo que não cruzem duas pontes. Ao conectar cidades, você só pode conectar a cidade a(i) no lado norte à cidade b(i) no lado sul. É preciso encontrar o número máximo de pontes que podem ser construídas para conectar pares norte-sul com as restrições acima mencionadas.

Esse problema se assemelha ao encontrado no tp, onde as pontes são as linhas de bandeirolas e as cidades são os bares e casas. O problema é resolvido encontrando a maior subsequência crescente, LIS, dada as coordenadas de cada item.

O problema LIS é caracterizado por encontrar uma subsequência de uma determinada sequência em que os elementos da subsequência estão em ordem ordenada, do menor para o maior, e em que a subsequência é a maior possível. Esta subsequência não é necessariamente contígua ou única. Exemplo:

arr[ ]	10	22	9	33	21	50	41	60	80
LIS	1	2		3		4		5	6

1- LIS = {10, 22, 33, 50, 60, 80}

A maior LIS será o maior número de pontes que poderemos construir, consequentemente, no problema a ser resolvido neste trabalho, o número de bandeirolas a serem penduradas.

Para facilitar a implementação e resolução do problema, é definido que todas os bares ficaram no lado par da rua e todas as casas estarão no lado ímpar da rua. Inicialmente, é feita uma ordenação pelo número dos bares e a LIS é calculada pelo número das casas.

Algoritmo Guloso

A solução utilizando a estratégia gulosa funciona com base no problema Patience Sorting e LIS (Longest Increasing Subsequence). É necessário que se faça algumas observações:

- Dado um array com os seguintes números: {2, 5, 3, 7, 11}. Temos duas LIS: {2, 3, 7, 11} e {2, 5, 7, 11}.

- Suponha que queremos adicionar o número 8 a este array. Percebe-se que adicionando este número, teremos que fazer a escolha de adicioná-lo ou não a nossa LIS, dado que, se adicionarmos, teremos que retirar o 11, pois 8 está em uma posição depois que 11 no array e é um número menor que 11.

- Caso o número 9 também seja adicionado ao array, a melhor escolha seria adicionar o 8 a LIS, já que assim poderíamos também adicionar o 9, o que não aconteceria se o 11 ainda fizesse parte da LIS.

- Considere também este array: {2, 5, 3}. Nesse caso, temos duas subsequências crescentes {2, 5} e {2,3}. Note que, como elas tem o mesmo tamanho, é mais proveitoso utilizar a subsequência {2,3} já que 3 é menor que 5, e caso tivéssemos um 4 no array, por exemplo, poderíamos adiciona-lo a {2,3}, o que não poderia ser feito em {2,5}.

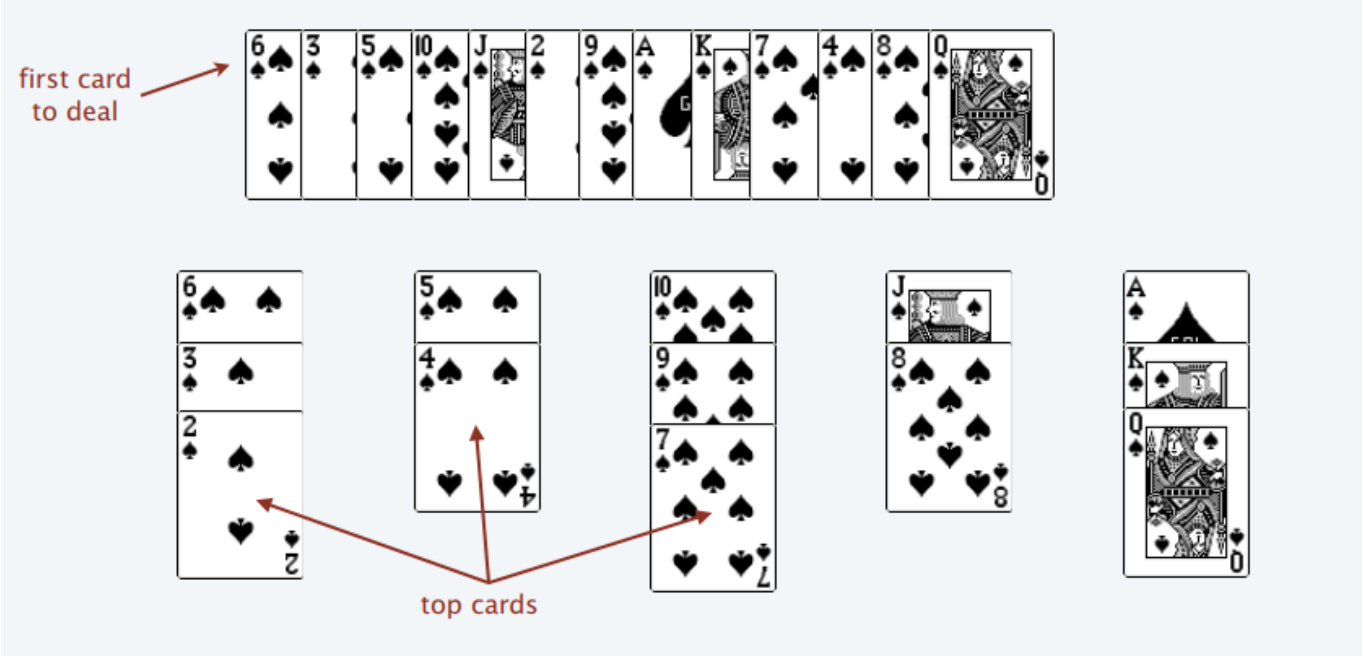
- Ainda considerando o array {2, 5, 3}, caso o próximo elemento seja 1, é constatado que ele não pode fazer parte de nenhuma das subsequências crescentes já encontradas {2,3} e {2,5}. Porém, o 1 pode formar uma nova subsequência {1}, essa subsequência pode ser uma potencial LIS.

Dadas as observações acima, podemos perceber que é necessário manter mais que uma subsequência, tendo assim uma lista de subsequências. A cada subsequência existente, verificamos seu elemento final e assim decidimos em qual subsequência existente o próximo número será adicionado, ou caso ele não possa fazer parte de nenhuma subsequência existente, é criada uma nova subsequência em que este elemento faça parte. A seleção é feita com base numa variação do problema Patience Sorting. Os critérios do Patience Sorting os seguintes:

Coloque as cartas c1, c2, ..., cn em pilhas de acordo com duas regras:

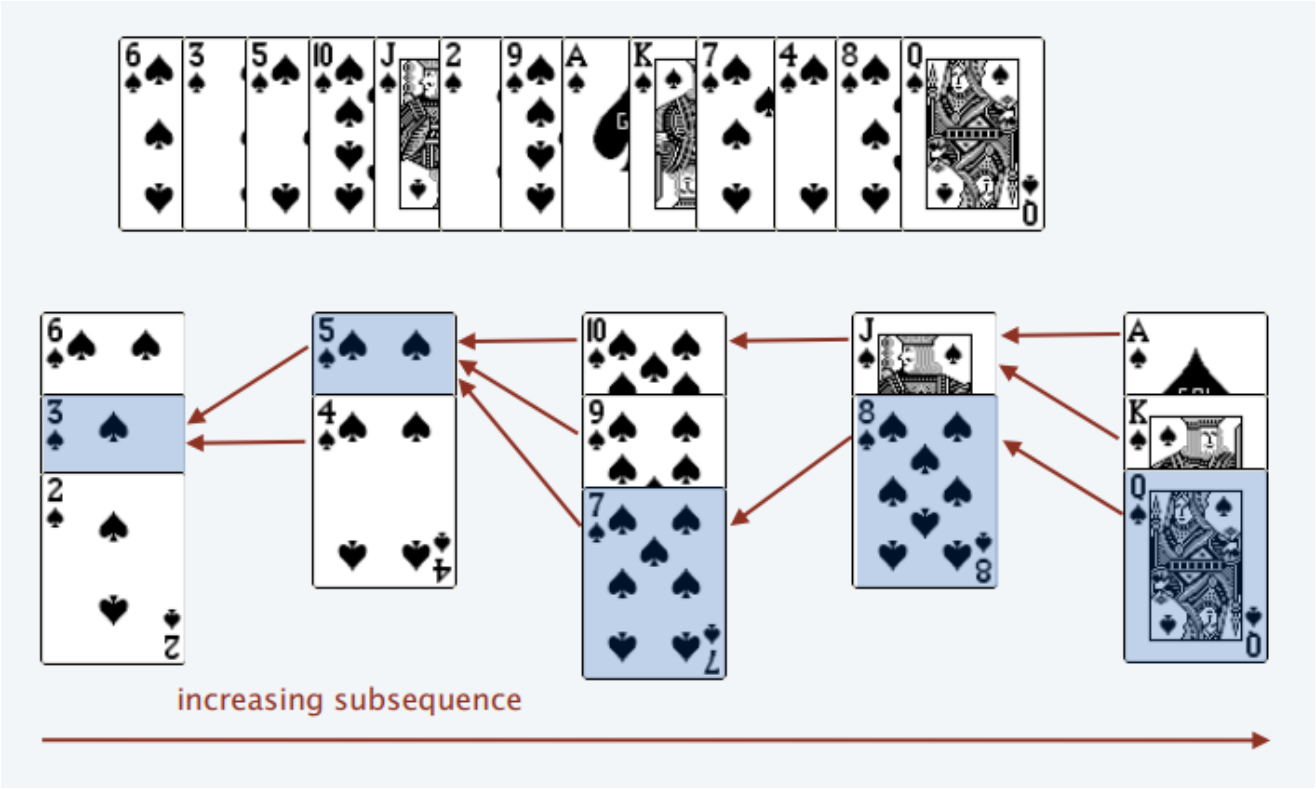
- ∇ Não é possível colocar uma carta de maior valor em uma carta de menor valor.
- ∇ Pode-se formar uma nova pilha e colocar uma carta nele.
- ∇ O objetivo é formar o mínimo de pilhas possível.

Esse algoritmo segue uma estratégia gulosa, ele sempre coloca cada carta na pilha mais à esquerda que ela se encaixa. Em qualquer fase durante o algoritmo, o número das cartas do topo das pilhas aumenta da esquerda para a direita. Exemplo:



As cartas de uma pilha formam uma subsequência decrescente. Qualquer sequência crescente pode usar no máximo uma carta de cada pilha.

Reconhecido por Hammersley em 1972, o Patience Sorting também pode ser usado para computar uma LIS. O número mínimo de pilhas = comprimento máximo de uma subsequência crescente.



A estratégia utilizada pelo algoritmo implementado é determinada pelas seguintes condições:

Seja A[] o array com os números a serem computados, e T[] o array que guarda todos os números do fim de cada subsequência encontrada.

- ∇ Caso 1: Se A[i] é menor entre todos os elementos de T criamos uma nova subsequência crescente formada por A[i].

- ∇ Caso 2: Se A [i] for maior entre todos os elementos de T, vamos clonar a maior subsequência e adicionar A[i] a ela.

- ∇ Caso 3: Se A [i] estiver no meio, encontraremos uma subsequência com o maior elemento final que é menor do que A[i]. Clonamos e adicionamos A[i] a esta subsequência. Descartamos todas as outras subsequências do mesmo comprimento que a da subsequência modificada.

  Caso 1: nenhuma subsequência existente, cria uma nova.

  A[0]

0	8	4	12	2	10

T[0] = 0;

0

Caso 2: 8 é maior que 0, portanto clonamos a subsequência {0} e adicionamos 8 a ela.

A[1]

0	8	4	12	2	10

T[1] = 8;

0	
0	8

Caso 3: 4 é menor que 8 e maior que 0, logo, clonamos a subsequência {0} e adicionamos 4 a ela. A subsequência {0,8} é substituída por {0,4}, já que elas têm o mesmo tamanho.

A[2]

0	8	4	12	2	10

T[1] = 4;

0	
0	4
0	8

Note que estamos lidando apenas com o elemento final das subsequências. Por isso, não é necessário manter todos os elementos da subsequência, apenas o último. Usaremos um array auxiliar para manter os elementos finais. Através dele, vamos definir em qual dos 3 casos citados acima o elemento se encaixa. Também mantemos um contador para acompanhar o comprimento do array auxiliar. Ao final da execução, este contador armazenará o número da LIS, que consequentemente

será o número máximo de bandeirolas que podemos pendurar na rua sem que elas se cruzem.

Força Bruta

A abordagem mais simples é tentar encontrar todas as subsequências crescentes e depois retornar o comprimento máximo da subsequência crescente mais longa. Para fazer isso, utilizamos uma função recursiva que retorna o comprimento do LIS possível do elemento atual (correspondente a posAtual) (incluindo o elemento atual). Dentro de cada chamada de função, consideramos dois casos:

1. O elemento atual é maior do que o elemento anterior incluído na LIS. Nesse caso, podemos incluir o elemento atual na LIS. Assim, descobrimos o comprimento do LIS obtido através da inclusão. Além disso, também descobrimos o comprimento do LIS possível ao não incluir o elemento atual no LIS. O valor retornado pela chamada de função atual é, portanto, o máximo entre os tamanhos das LIS encontradas incluindo e não incluindo o elemento atual.

2. O elemento atual é menor do que o elemento anterior incluído no LIS. Nesse caso, não podemos incluir o elemento atual na LIS. Assim, descobrimos apenas o comprimento da LIS possível ao não incluir o elemento atual na LIS, que é retornado pela chamada de função atual.

Ao final, teremos sempre a maior LIS dentre todas as possibilidades calculadas.

Análise de complexidade

Em toda leitura, é feita uma ordenação utilizando a função qsort no array de bares. Essa ordenação tem complexidade de tempo O(nlogn). Também são utilizados dois arrays para representar cada lado da rua. A complexidade de espaço é $O(n^2)$.

Programação Dinâmica

Complexidade de tempo: $O(n^2)$ Dois laços de tamanho n aninhados são executados.

Complexidade de espaço: O(n): apenas um array de tamanho n é usado para armazenar a LIS de cada número da sequência, sendo o resultado final o maior número armazenado nesse array.

Algoritmo Guloso

Complexidade de tempo: O(n log n). O loop é executado para N elementos e a busca binária, que leva log(n) de tempo, é executada todas as N vezes.

Complexidade de espaço: O(n): apenas um array de tamanho n é usado para armazenar o número final de cada subsequência criada.

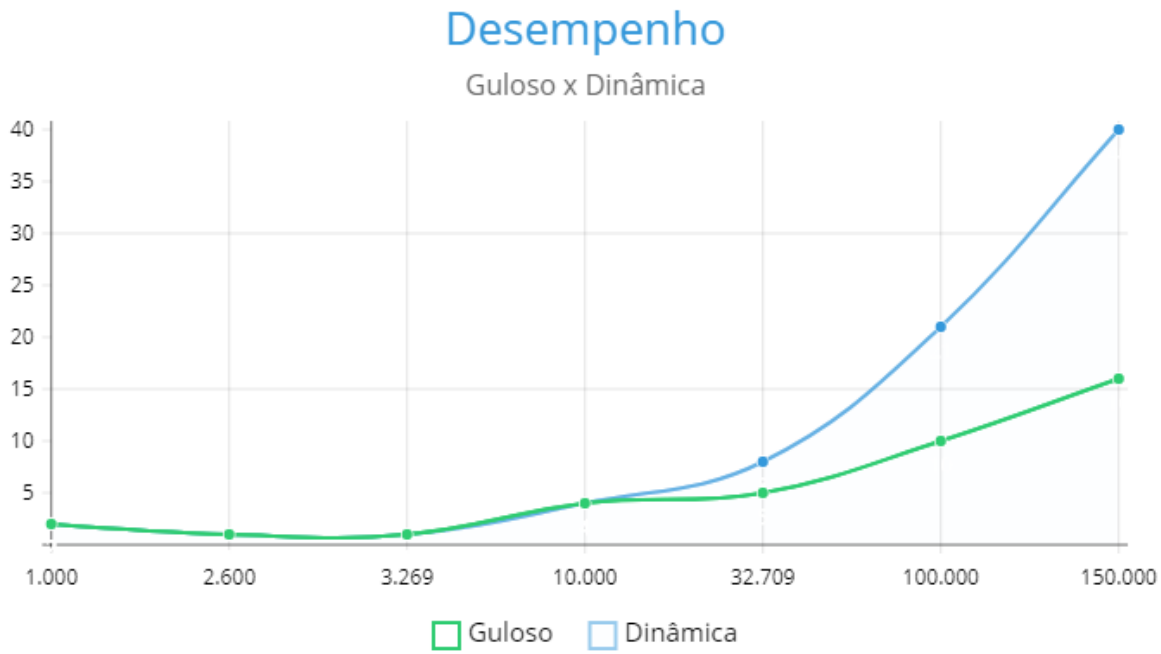
Força Bruta

Nesta abordagem, muitas chamadas recursivas tiveram que ser feitas repetidas vezes com os mesmos parâmetros, causando um crescimento exponencial. Essa redundância pode ser eliminada armazenando os resultados obtidos.

Complexidade de tempo: $O(2^n)$

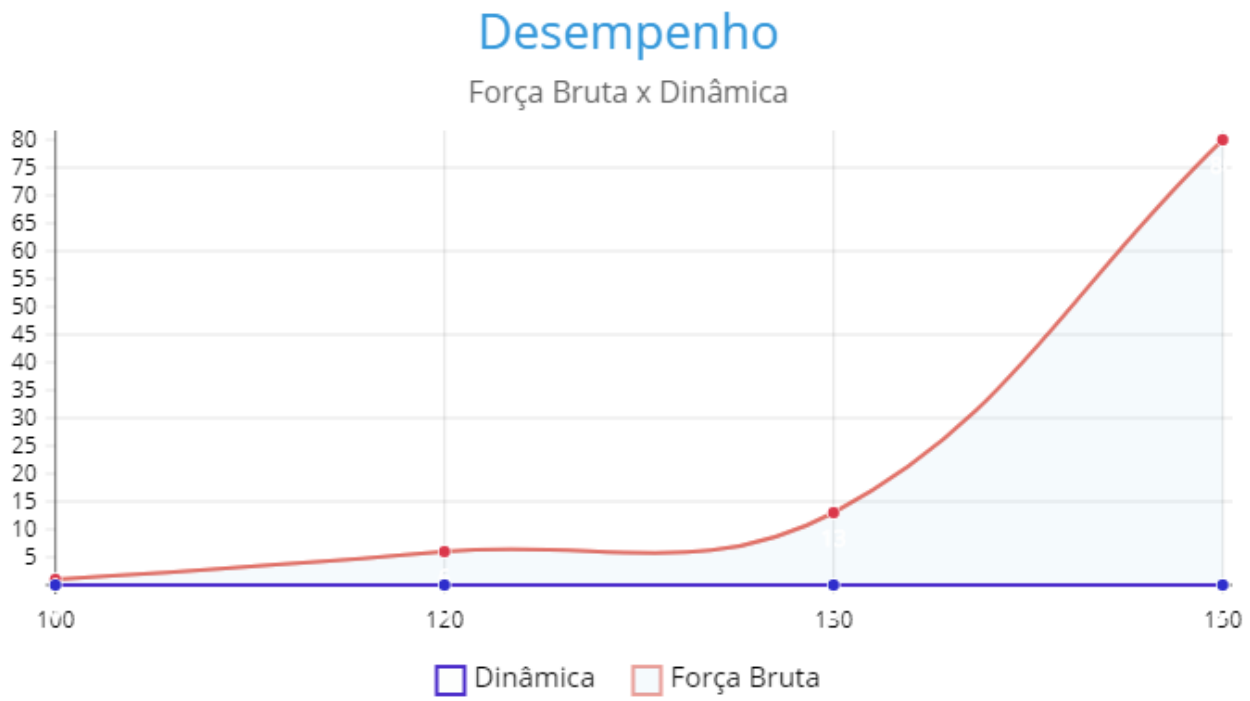
Complexidade de espaço: O(1): apenas uma memorização é feita, o máximo entre inclui e naoInclui.

Avaliação Experimental



Com os testes executados é possível perceber que com entradas a partir de 10.000, o algoritmo Guloso tem uma performance mais rápida do que o de Programação Dinâmica, atestando assim sua complexidade de tempo O(nlogn) menor que $O(n^2)$. Tanto o dinâmico quanto o guloso sempre dão as respostas ótimas.

O algoritmo de força bruta dá resultados apenas para entradas pequenas (menos de 200), em entradas muito grandes o algoritmo não consegue computar o resultado em um tempo razoável, o que atesta sua complexidade exponencial $O(2^n)$.



Os testes utilizados foram os disponibilizados por colegas no minhaUFMG e podem ser encontrados aqui.

Referências

https://www.techiedelight.com/longest-increasing-subsequence-using-dynamic-programming/

https://en.wikipedia.org/wiki/Longest_increasing_subsequence

http://www.geeksforgeeks.org/dynamic-programming-building-bridges/

https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/LongestIncreasingSubsequence.pdf