UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

ALGORITMOS I - TP2

Breno Poggiali de Sousa
2018054800

October 25, 2019

Sumário

1 Sobre o Problema

O objetivo deste trabalho é, dado um valor inteiro N, que representa o valor máximo possível de ser gasto por cada menina e um conjunto M de ilhas com seus custos e pontuações por dia, desenvolver duas soluções:

1. Utilizando algoritmo guloso determinar: (a) a maior pontuação possível se as meninas escolherem um roteiro no qual pode haver repetições de ilhas (ficar mais de um dia na mesma ilha) e (b) a quantidade de dias que durará a viagem. O tempo de execução do algoritmo para esse problema não deve ser superior a O(m lg m);

2. Utilizando programação dinâmica determinar: (a) a maior pontuação possível se as meninas escolherem um roteiro no qual não pode haver repetições de ilhas e (b) a quantidade de dias que durará a viagem. O tempo de execução do algoritmo para esse problema não deve ser superior O($n * m$);

2 Princípios de projeto

Porque no primeiro problema é utilizado um algoritmo guloso e no segundo programação dinâmica?

No **primeiro problema**, utilizamos de um algoritmo guloso, pois temos um problema de otimização míope, ou seja, um problema que toma decisões baseadas unicamente na iteração corrente que ele está. Isso porque, como queremos a maior pontuação possível quando pode haver repetições de ilhas e não fazemos questão de retornar a solução ótima, basta ordenarmos pelo custo benefício e irmos encaixando quanto que cabe de cada ilha com o melhor custo benefício até a de pior custo benefício dentro do nosso N (valor que o nosso grupo tem para gastar).

Já no **segundo problema**, utilizamos de um algoritmo que utiliza programação dinâmica, pois precisamos necessariamente retornar a solução ótima. E, como para nossa solução, cada tomada de decisão depende de tomadas de decisões anteriores, utilizamos programação dinâmica para não termos que recalcular esses valores previamente calculados várias vezes. O que gasta mais espaço, mas nos economiza tempo!

Como a programação dinâmica foi construída? (como o problema foi subdivido em subproblemas)

A programação dinâmica foi construída baseada no Knapsack Problem (ou problema da mochila) adaptado. Esse algoritmo funciona da seguinte forma: o que queremos saber é, dados os valores e os pesos de cada objeto, qual a melhor combinação de elementos que eu consigo que maximize o valor dos objetos que eu coloquei na mochila sem passar o peso que a mochila suporta.

Na nossa abordagem, os valores dos objetos são os pontos que queremos maximizar e os pesos de cada objeto é a quantidade de dinheiro que temos para gastar, que queremos que caiba dentro. Dessa forma, basta explicarmos qual a ideia do Problema da Mochila. Ele consiste em construirmos uma matrix M x N, onde M é a a quantidade de ilhas e N é o valor máximo que temos para gastar. **OBS:** em nossa abordagem, dividimos o valor de cada ilha e o valor total que temos para gastar (N), pelo MDC entre todos eles, de forma que obtemos o mesmo resultado, mas conseguimos diminuir consideravelmente o tamanho de nossa matriz, o que deixa a nossa solução bem mais eficiente.

Após montarmos a nossa matriz, andamos por ela a partir da primeira linha (i = 0), da primeira (j = 0) até a última coluna(j = N), depois pela segunda linha (i = 1), da primeira (j = 0) para a última coluna (j = N) e assim vai. Nesse processo de andar pela matriz em N*M posições, utilizamos de otimizações (divisões em subproblemas) para que, ao chegarmos na casa [M][N], ou seja, na última posição de nossa matriz, necessariamente iremos possuir o valor ótimo.

Como são calculados nossos subproblemas: nossa função *getMaxPointsAndDaysWithoutRepeat* recebe 4 parâmetros: N (dinheiro para gastar - já dividido pelo MDC se for o caso), M (quantides de ilhas), vetor de custos e vetor de pontos, onde custos[i] é o custo da ilha *i* e pontos[i] é a quantidade de pontos que a ilha *i* possui. E iteramos por cada elemento. Caso estejamos em i == 0 ou j == 0, setamos points_days[i][j] = 0, se costs[i-1] $<=$ j, setamos points_days[i][j] para o máximo entre points_days[i-1][j] e points_days[i-1][j-costs[i-1] + (points[i-1], 1), senão, fazemos points_days[i][j] = points_days[i-1][j].

3 Análise de complexidade

Solução Gulosa

Tempo: O(M log M). Pois a complexidade que predomina nesse algoritmo em relação ao tempo é a parte de ordenar, que no caso foi utilziada a std::sort que, na versão utilziada para compilar do C++ (C++14), é garantido que possui complexidade O(M log M). O resto, só estou passando pelo vetor de ilhas ou pelo vetor de custo benefício, o que possui O(M) para ambos. De forma que O(M log M) é predominante.

Espaço: O(M). Pois preciso salvar um vetor de tuplas de tamanho M, que eu chamo de *cost_benefit* em meu código. Se trata de um vetor que contém o custo benefício, a quantidade de dias que consigo ficar numa ilha (em float) e que ilha se trata. Todo o resto de espaço que eu gasto é praticamente disprezível comparado com o vetor *cost_benefit*, pois todos possuem O(1) espaço.

Solução com Programação Dinâmica

Tempo: O(M*N). Pois a complexidade que predonomia nesse algoritmo em relação ao tempo é a parte de passar por todos os elementos da matriz que, no caso, possui M linhas por N colunas. **Obs:** é interessante observar aqui que, na verdade, a quantidade de colunas é N/MDC(MDC(entre valor de todas as ilhas), N), de forma que nossa complexidade, na verdade, graças à uma otimização, é O(M*(N/MDC(valor total, valores das ilhas))). Essa operação é predominante no tempo pois, nesse algoritmo, a única outra operação que fazemos é passar por todas as ilhas para criar os vetores *costs* e *points*. Operação essa que possui O(M) de complexidade, o que faz com que a nossa complexidade predominante continue sendo O(M*N).

Espaço: O(M*N). Pois o uso de espaço predominante é o utilizado para montarmos a matriz M*N para guardarmos valores previamente calculados. Além dessa matriz, usamos outros dois vetores (*costs* e *points*), que possuem tamanho M e, por isso, possuem O(M) complexidade de espaço, de forma que o O(M*N) predomine. Interessante observar aqui que, assim como explicamos na complexidade de tempo, esse O(M*N) é, na verdade O(M*(N/MDC(N, valores das ilhas))).

4 Prova de Corretude

Solução Gulosa

Para provamos a corretudo da solução gulosa, basta provarmos que ela retona o que é esperado dela. Ou seja, que ela retorna a maior pontuação possível se as meninas escolherem um roteiro no qual pode haver repetições de ilhas e a quantidade de dias para essa pontuação. Assim como foi explicado em sala de aula e pelo fórum, essa solução não precisa retornar necessariamente a solução ótimo. Isso ocorre porque não podemos ficar dias ”quebrados” em cada ilha.

De qualquer forma, o nosso raciocínio foi criar um vetor de custo benefício, que é um vetor de tuplas contendo o custo benefício daquela ilha (Quantidade de pontos da ilha/Preço da ilha), ou seja, a ilha que retorna mais pontos para 1 real é a ilha com o melhor custo-benefício. Além disso, nossa tupla contém também quantos dias cabem naquela ilha e qual ilha está sendo referenciada.

A partir disso, ordenamos o nosso vetor primeiro pelo custo benefício e depois pela quantidade de dias. Com esse vetor ordenado, basta iterarmos por cada ilha a partir do maior custo benefício até o menor e irmos subtraindo de N a quantidade de dias que cabem em N naquela ilha * o preço da ilha. Dessa forma, como nosso vetor está ordenado por melhor custo benefício, é garantido que obteremos a maior quantidade de pontos (pois estamos pegando primeiro as ilhas que oferecem uma maior pontuação por real gasto).

Além disso, como ordenamos depois por dia também, fica garantido que obteremos o melhor resultado de dias, ou seja, para aquela pontuação máxima, a quantidade de dias máxima que conseguimos ficar!

Solução com Programação Dinâmica

Para provamos a corretudo da solução com Programação Dinâmica, basta provarmos que o Problema da Mochila retorna a solução ótima, visto que ele foi a base da construção da nossa abordagem. Pensando no Problema da Mochila, ele retorna sempre a solução ótima e sabemos disso pelo fato de ele ser um dos 21 problemas NP-completos de Richard Karp exposto em 1972.

Não obstante, iremos mostrar o porquê dele retornar sempre a solução ótima.

Isso se deve ao fato de estarmos considerando apenas os subconjuntos onde o peso total é menor ou igual ao peso máximo (no nosso caso, o conjunto onde a soma dos preços das ilhas é menor ou igual ao preço N que podemos gastar).

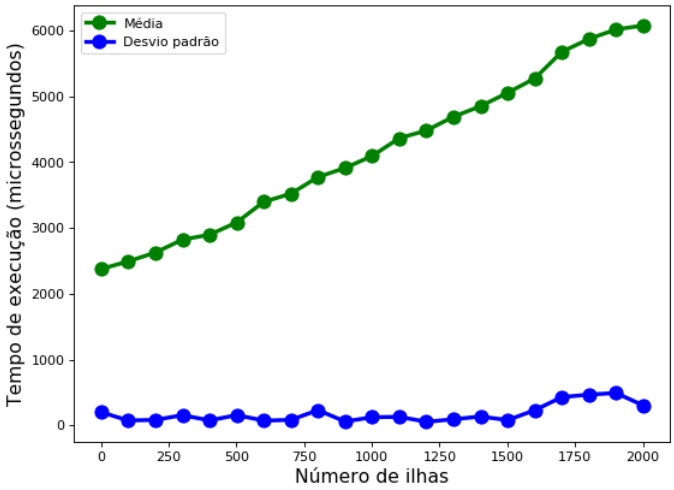
Pensando nesses subconjuntos de ilhas, podemos ter 2 casos para cada ilha: ou a ilha está incluída no subconjunto ideal ou ela não está incluída nesse subconjunto ideal (nossa resposta). Dessa forma, a quantidade de pontos máxima que pode ser obtido de M ilhas é, no máximo: ou a quantidade de pontos máxima obtido por M-1 ilhas e custo N (excluíndo a M-ésima ilha) ou a quantidade de pontos da M-ésima ilha mais a quantidade de pontos máxima obitida das M-1 ilhas e N-(custo da N-ésima ilha).

Dessa forma, se o custo da N-ésima ilha for maior que N, essa ilha não poderá ser incluída e, o primeiro caso (a quantidade de pontos máxima pode ser no máximo a quantidade de pontos máxima obtido por M-1) será a única possibilidade.

Por fim, como dividimos em subproblemas sempre com M-1 e começamos M variando de M = 0 até M = M(quantidade de ilhas), garantimos que, ao chegar no M = M (quantidade de ilhas), teremos a melhor solução até essa ilha que, no caso, será a melhor solução global do problema pois já teremos passados em todas as ilhas.

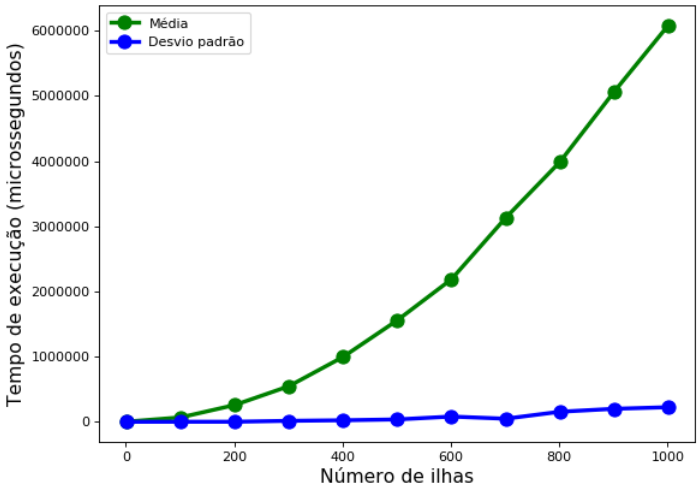
5 Avaliação Experimental

Tempo de execução dos algoritmos
Solução Gulosa



Média: No gráfico.
Desvio Padrão: No gráfico.

Solução com Programação Dinâmica



Média: No gráfico.
Desvio Padrão: No gráfico.

Com as saídas dos gráficos acima, podemos garantir que o nosso programa está rodando dentro da complexidade desejada. É interessante observar, porém, que no caso da Programação Dinâmica, o tempo deu consideravelmente alto, isso porque, desabilitei a divisão pelo MDC para que os nossos resultados fossem mais consistentes e conforme o esperado.

No caso, é importante salientar também que, em ambos os exemplos, N está variando em função de M. No caso, para todo M, rodamos exemplos aleatórios com N = M*100.

Discussão das saídas de ambas as abordagens: (qual é mais vantajosa para cada abordagem: conhecer mais lugares ou aumentar o tempo de estadia no local)

Após toda a discussão elaborada nesse trabalho e as análises de complexidade acima, podemos chegar na conclusão que não existe necessariamente uma abordagem melhor que a outra. Cada uma é mais eficiente para o determinado problema que quer resolver. No caso de querer conhecer o maior número de lugares possíveis, com certeza, a abordagem com Programação Dinâmica, utilizando o Knapsack Problem é a melhor abordagem, pois retorna a maior pontuação nesse cenário.

Já no caso do problema de aumentar o tempo de estadia em um local, a abordagem gulosa é claramente melhor, pois ela consegue favorecer que fiquemos mais dias nas ilhas que possuem o melhor custo benefício. Assim como vimos, apesar dela não retornar necessariamente a solução ótima, na maior parte dos casos teste ela retorna sim a solução ótima, além de retornar, quando não é esse o caso, uma solução bem próxima da ótima geralmente.

Bibliografia

ALMEIDA, J. M. (2019) *Slides da disciplina de Algoritmos I*

ZIVIANI, N. (2006). *Projetos de Algoritmos com Implementações em PASCAL e C: Capítulo 2: Paradigmas de Projeto de Algoritmos*. Editora Cengage.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. (2009) *Introduction to Algorithms*. MIT Press.