UNIVERSIDADE FEDERAL DE MINAS GERAIS

ALGORITMOS 1
DOCUMENTAÇÃO TP2

MATHEUS TIAGO PIMENTA DE SOUZA - 2018054893
PROFESSORA JUSSARA M. ALMEIDA

Objetivo:

Este trabalho da disciplina de Algoritmos 1 tem como objetivo fazer com que os alunos apliquem os conceitos de "Algoritmos gulosos" e de "programação dinâmica" vistos em aula, além de praticarem e aperfeiçoarem técnicas de programação. Dessa forma, dado o problema de encontrar a melhor solução de ilhas a se visitar, deve-se implementar um algoritmo guloso e um usando programação dinâmica para resolvê-lo, dada as suas respectivas condições.

Ambiente de desenvolvimento, compilação e execução:

O projeto foi desenvolvido no ambiente de C++ e compilado nas máquinas do DCC utilizando o compilador G++. Para compilar, rodar *make* no diretório onde estão os arquivos. Em seguida, execute o executável gerado, passando por linha de comando o arquivo texto que será usado.

O problema: Viagem às ilhas:

Dada a informação de quanto dinheiro cada integrante do grupo possui, e uma lista de todas as ilhas desejáveis pelo grupo, contendo somente o quanto será gasto por dia e a nota de cada ilha (onde quanto maior a nota melhor é a ilha, na visão do grupo), deve-se apresentar duas soluções:

1. Usando um algoritmo guloso, determinar a maior pontuação possível de um roteiro no qual pode haver repetições de ilhas, apresentando também quantos dias ficaram viajando. O algoritmo tem que ter tempo de execução da ordem de $O(m*(\log m))$;

2. Usando programação dinâmica, determinar maior pontuação possível de um roteiro no qual não pode haver repetições de ilhas, apresentando também quantos dias ficaram viajando. Este algoritmo tem que ter tempo de execução da ordem de $O(n*m)$;

Análise do algoritmo:

1. *Guloso:*

Antes de falar do algoritmo guloso em si, vale pontuar algumas coisas. Para os dois algoritmos foram utilizados uma *struct* chamada "island" para as ilhas, que contém 3

parâmetros: o custo de cada dia, a pontuação da ilha e a razão entre os pontos e o custo. Assim dada a entrada das listas, é gerado um vetor dessa *struct* e usado durante todo o código.

A ideia de um algoritmo guloso é fazer a melhor escolha local, na esperança que leve a melhor escolha global. Traduzindo para o nosso problema, o algoritmo deve selecionar a melhor ilha e escolhê-la o máximo de vezes possível. Quando não for mais possível escolher a primeira, passar para a segunda melhor ilha e realizar o mesmo processo, e assim sucessivamente. Nesse momento surge a grande pergunta é: como saber a melhor ilha? Simples, basta olhar para a razão entre o custo de cada dia na ilha com os pontos da mesma, ou seja, quanto vale (em dinheiro) cada ponto daquela ilha. Como temos que realizar a melhor escolha local, temos que a melhor ilha é aquela que cada ponto seu vale menos, que é o mesmo que dizer que para cada 1 real, se consegue mais pontos naquela ilha.

Portanto, é calculada a razão de cada ilha e armazenada no campo "ratio" da *struct*. Porém, se para cada vez que formos buscar uma ilha precisarmos percorrer todo o vetor de ilhas, teríamos uma complexidade alta (da ordem de $O(n^2)$). Para evitar tais buscas, ordenamos o vetor pela razão, em ordem crescente, utilizando o *mergesort*. Dessa forma teremos nosso vetor ordenado na ordem das melhores ilhas.

O mergesort funciona da seguinte forma: dado um vetor, ele é dividido na metade, e para cada subvetor é chamada a mesma função recursivamente. Assim, o vetor será subdividido em subvetores até chegar a um elemento por subvetor. Nesse momento que acontece o *merge*, ou a junção desses subvetores, onde se une dois subvetores em um vetor maior, de forma ordenada (se compara cada elemento dos subvetores com o auxílio de dois iteradores, colocando o menor dessa comparação e andando com o iterador pelo subvetor). Na imagem abaixo vemos um exemplo do mergesort:

Imagem da transparência de mergesort do curso de Estrutura de Dados de 2018/2, ministrada pelos professores Luiz Chaimowicz e Raquel Prates

No mergesort utilizado foi adicionada uma otimização, onde em vez de passar para o merge dois vetores, passamos apenas um vetor maior onde inserimos o primeiro vetor em ordem crescente e logo em seguida segundo vetor na ordem inversa. É necessário passar também o índice da mudança dos vetores.

Com o vetor de ilhas ordenado em ordem de melhores ilhas, basta selecionar de acordo com o dinheiro total. Se o custo de uma ilha foi maior que o montante, essa mesma ilha não é adicionada a solução do algoritmo. Sendo menor ou igual, dividimos o montante total pelo custo da ilha, de forma que o quociente é o número de dias que deve-se ficar naquela ilha e o quociente vezes os pontos daquela ilha são os pontos totais feitos naquela ilha. Esse processo é realizado até acabar o dinheiro, ou as ilhas, ou não for possível pagar um dia nas ilhas restantes. Guardando os dias e os pontos feitos em cada ilha, ao final do processo teremos a solução do problema.

Apesar de não parecer trivial, os algoritmos gulosos nem sempre entregam a solução ótima, e esse não é diferente. Vamos pensar no caso onde são passadas 3 ilhas e 1000 reais:

	Custo	Pontos
1	800	80
2	300	25
3	700	65

A ilha com maior custo-benefício é a 1, e portanto o algoritmo vai escolher ela, apesar de a solução ótima seria escolher a 2 e a 3 vai gerar uma maior pontuação. Provamos portanto por contra-exemplo que o algoritmo guloso não é ótimo.

O problema de escolher as ilhas podendo repeti-las é tratado com um algoritmo guloso porque, apesar de não ser ótimo, é bem menos custoso. Se fossemos usar programação dinâmica por exemplo, teríamos um extenso espectro de operações, já que o número de possibilidades diferentes faria com que a as chamadas recursivas e a construção da tabela fossem muitas e, dessa forma. bastantes custosas em termos de processamento, execução e tempo.

2. *Programação Dinâmica:*

O paradigma de programação dinâmica está dentro da ideia de “dividir para conquistar”, que significa dividir o problema em subproblemas menores e mais fáceis de resolver, resolvê-los (conquistá-los) e a partir da solução desses subproblemas chegar na

solução do problema. Muitas das vezes essas divisões ocorrem recursivamente. Porém, a programação dinâmica tem a diferença de que seus subproblemas são sobrepostos: nas chamadas recursivas o computador vai fazer mais de uma vez a operação desejada para o mesmo subproblema, pois os subproblemas se repetem. Por exemplo, na sequência de Fibonacci, para calcular o F(n) é necessário saber o F(n-1) e o F(n-2). Usando divisão e conquista, é fácil perceber que será necessário calcular o F(n-2) duas vezes, no F(n-2) e no F(n-1) que depende do F(n-2). Temos portanto dois subproblemas sobrepostos.

Com os subproblemas sobrepostos, foi questionado se fazer as mesmas operações várias vezes era necessário. A resposta é não, e isso é a ideia chave da programação dinâmica: salvar os resultados já obtidos para não precisarem ser calculados novamente. Utilizando novamente o caso da sequência de Fibonacci, podemos salvar os valores já calculados num vetor, para depois só serem consultados.

Aplicando tais conceitos para resolver o problema, consideramos a escolha ou não da última ilha do vetor para a solução ótima como o grande problema. Sabemos que essa escolha depende da escolha ou não da ilha anterior, que por sua vez depende da ilha anterior e assim sucessivamente. Dessa forma, a escolha de uma ilha depende da escolha (ou não) de todas as anteriores, e consequentemente a escolha da última depende de todas as outras. Temos portanto uma estrutura de recursão, e os subproblemas se unem facilmente passando a melhor pontuação possível para o problema posterior.

Com esse pensamento, foi criada uma matriz M*N chamada Mem para ser a memória do código, que irá salvar a melhor pontuação para uma determinada ilha com um determinado dinheiro. É importante setar todos os valores com -1, para não confundir na hora da análise.

A estrutura recursiva foi montada da seguinte forma:

```
Dinamica(ilha,dinheiro):
    Se acabaram as ilhas ou acabou o dinheiro:
        return(0);
    Senão se o valor da matriz Mem para essa ilha e esse dinheiro != -1:
        return( Mem[ilha][dinheiro] );
    Senão se o custo dessa ilha for maior que o dinheiro:
        pontos = Dinamica( ilha - 1, dinheiro );
    Senão:
        ans1 = Dinamica( ilha - 1, dinheiro );
        ans2 = ilha.pontos + Dinamica( ilha - 1, dinheiro - ilha.custo);
        pontos = max( ans1, ans2);
    Mem[ilha][dinheiro] = pontos;
    return(pontos);
```

Temos que o primeiro *if* é o caso base. O segundo é consultar se tão situação já foi calculada, e caso tenha basta retornar o valor. O terceiro é caso o custo seja mais alto que o dinheiro atual. Nesse caso não se deve calcular nada, apenas passar para a próxima ilha. Por fim, o *else* calcula qual situação é melhor, não ficar nessa ilha (ans1) ou ficar (ans2). Perceba

que ao testar ficar na ilha nós adicionamos a pontuação obtida por ficar na ilha a solução e tiramos do dinheiro o custo da ilha. O resultado (pontos) é o maior dessas duas situações, e antes de retorná-lo, armazenamos esse valor na nossa *Mem*. Ao passarmos o dinheiro total e o número total de ilhas, no final da execução teremos a melhor pontuação possível.

Para obtermos o número de dias entretanto, temos que usar a *Mem*. Sabemos que o resultado final está armazenado na última linha e na última coluna. Agora, se andarmos uma linha, o que significa que estamos analisando o mesmo dinheiro para a ilha anterior, teremos duas situações: ou o valor é o mesmo da última casa, ou é diferente. Vamos analisar cada um dos casos.

Se o valor for igual, significa que a última ilha não foi selecionada. Nesse caso, o programa chama ele mesmo para a mesma quantidade de dinheiro e a ilha anterior, e assim o valor da ilha anterior vai ser repetido para a ilha seguinte (pois esse caso é melhor do que selecionar a última ilha). Para a análise dos dias, isso significa que não houve nenhum dia.

Agora a situação é diferente se o valor mudou, pois significa que foram adicionados pontos entre a penúltima e a última ilha, o que significa que a última ilha foi selecionada, e um dia a mais. Não obstante, sabemos que esses pontos foram o da última ilha, já que analisando como o algoritmo funciona, se uma ilha foi selecionada o programa soma os pontos da ilha atual e passa a anterior com o montante total menos o custo da ilha atual.

Portanto, para analisar quantos dias se passaram na melhor solução, basta andar na Mem, partindo da última casa, analisando se os valores mudam ou não entre as ilhas. Se mudam, deve-se andar uma linha para cima e andar nas colunas o equivalente ao custo da ilha para esquerda, somar 1 ao contador de dias e continuar a análise, até acabarem as ilhas. Por exemplo, vamos supor que a pontuação muda entre a ilha *i* e a *i-1*. Deve-se portanto ir para a linha *i-1*, mas subtrair do montante atual o custo da ilha *i*.

Vale ressaltar que esse algoritmo não cobre o caso onde a primeira ilha é selecionada e a segunda não, pois ele não tem uma ilha anterior para comparar se a ilha foi selecionada ou não. Assim, deve-se verificar se, para o montante que chegou na primeira ilha, se esse valor da matriz *Mem* for diferente de 0, deve-se adicionar um dia. Ao final, teremos o número de dias, e resolvemos o problema.

Diferente do algoritmo guloso, este, utilizando PD, é ótimo. Para provar, vamos supor *S* como sendo a solução ótima para um certo conjunto de ilhas, com seus pontos e custos, para um certo dinheiro. Agora vamos considerar uma ilha m.

Vamos supor, por contradição, que *S - {m}* não seja a solução ótima para o conjunto de ilhas menos a ilha *m*. Dizer isso significa dizer que existe outro conjunto, por exemplo *S'*, que também não contem *m* e é solução ótima, para esse caso. Assim:

$$pontos(S') > pontos(S- \{m\})$$

Adicionando a ilha *m* as duas soluções, e chamando o conjunto S' + {m} = S'', temos, com relação aos pontos:

$$pontos(S') + pontos(m) > pontos(S- \{m\}) + pontos(m)$$
$$pontos(S'') > pontos(S)$$

Ou seja, encontramos um conjunto que é melhor q S. Mas isso é uma contradição, pois partimos do pressuposto que S era a solução ótima para esse conjunto de ilhas. Provamos portanto que o algoritmo é ótimo.

Esse segundo problema é tratado com programação dinâmica porque o fato das ilhas não repetirem torna o algoritmo guloso mais propício a dar resultados errados, pois é mais simples explorar suas fraquezas, como a não análise de todas as possibilidades. Também, o fato de, ao subdividir o problema em subproblemas temos subproblemas sobrepostos auxilia e muito a escolha de PD. Assim, se faz necessário analisar todas as possibilidades, e como muitos casos vão usar os mesmos resultados, faz programação dinâmica ser a melhor escolha.

Complexidade do algoritmo:

1. *Guloso:*

    a. Complexidade de tempo: Temos no algoritmo guloso duas grandes funções, o *mergesort*, que tem complexidade da ordem de $O(m*log\ m)$, e o *greed*, que tem apenas um loop *while* que no pior caso vai varrer todo o vetor de ilhas e realizar operações de custo 1, ou seja, é O(m). Portanto, o algoritmo é $O(m*log\ m)$.

    b. Complexidade de espaço: O algoritmo cria um vetor para auxiliar em cada interação do merge, que terão no máximo tamanho M. Assim podemos dizer que teremos uma complexidade da ordem de $O(m)$.

2. *Programação Dinâmica:*

    a. Complexidade de tempo: Os maiores custos estão na criação e no preenchimento da matriz, que são da ordem de $O(n*m)$. As chamadas recursivas pelo fato de consultarem os valores da memória, terão nesse caso custo constante. A consulta do número de dias é da ordem de $O(m)$ apenas. Assim, a ordem do algoritmo é de $O(n*m) + O(m) = O(n*m)$.

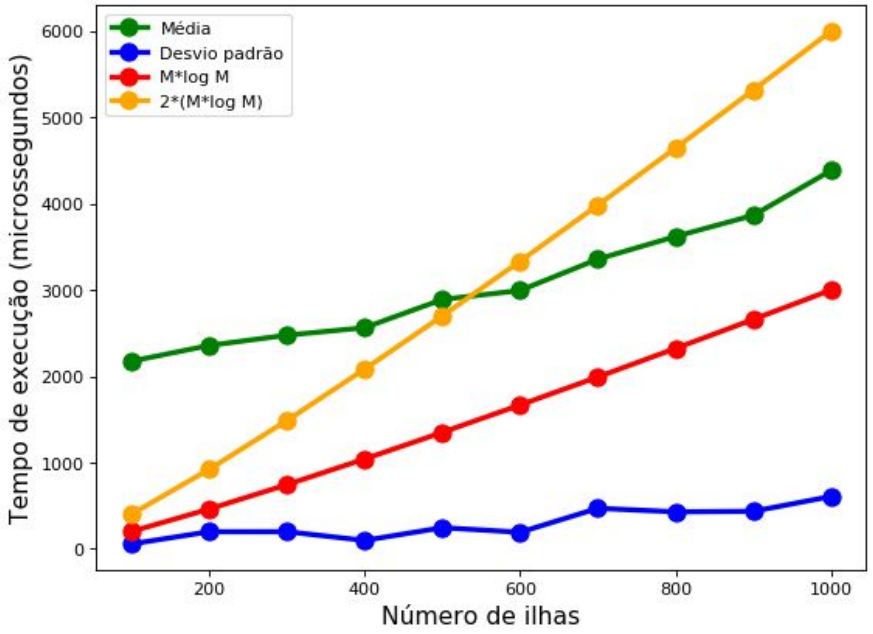
    b. Complexidade de espaço: O maior custo de espaço está na matriz *Mem*, que tem tamanho $N*M$. Assim, o algoritmo é da ordem de $O(n*m)$.

Avaliação experimental:

Para realizar a avaliação experimental, foram gerados automaticamentes casos que englobam diferentes números de ilhas e diferentes montantes iniciais. Utilizado um algoritmo em python que gera os gráficos, e um em bash para executar tudo coordenadamente.

Algoritmo Guloso:

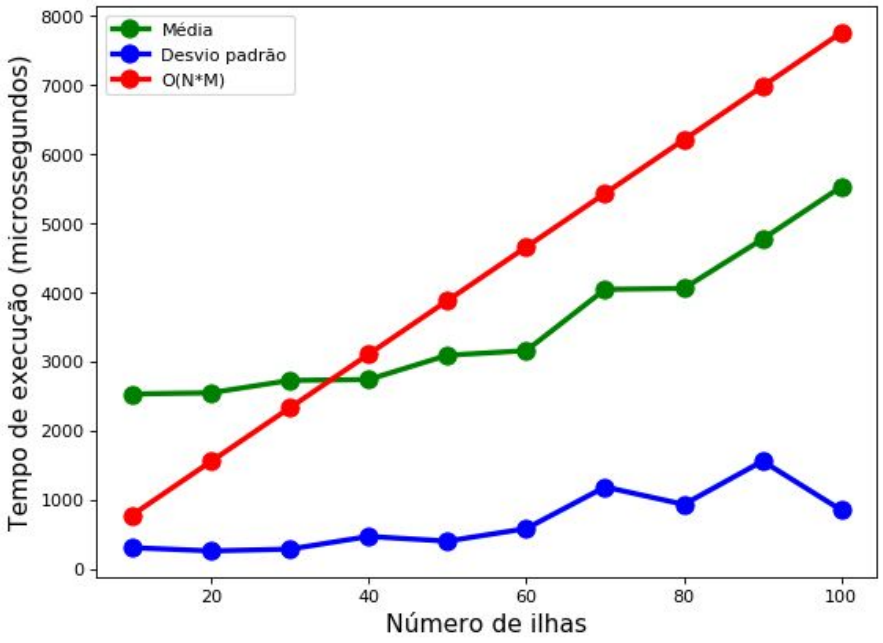
Para o primeiro algoritmo foram gerados testes variando o número de ilhas entre 100 e 1000, adicionando 100 ilhas por iteração. Cada teste foi repetido 20 vezes. O montante inicial N foi gerado aleatoriamente. Os resultados estão ilustrados no gráfico abaixo:



As linhas vermelhas e laranjas são curvas de *M*log(M)* vezes uma constante, onde a vermelha a constante é 1 e na laranja, 2. Percebemos que a média das iterações do algoritmo (linha verde), a partir de um valor entre 500 e 600, é superada pela linha laranja, o que mostra a dominância assintótica de *2*M*log(M)* no código. Temos portanto a demonstração gráfica que o algoritmo produzido tem complexidade assintótica de tempo da ordem de *O(M*log(M))*, pois, a partir de um certo valor, o algoritmo é dominado por vezes uma constante (definição de domínio assintótico).

Programação dinâmica:

Para o segundo algoritmo foram realizados testes variando o número M de ilhas entre 10 e 100, de 10 em 10. O valor de N gerado aleatoriamente foi de 7760. Cada teste foi realizado 20 vezes. Abaixo os resultados:



Diferente do gráfico anterior, a linha vermelha está como *O(N*M)* porque teve que ser multiplicada por 0.1 para ficar na escala do gráfico. Podemos perceber que, a partir de aproximadamente N = 40, a função *0.1*N*M* domina assintoticamente a curva das médias das medidas (curva verde), o que mostra que o algoritmo tem complexidade assintótica de tempo da ordem de *O(N*M)*.

Conclusões:

Com esse trabalho foi possível aprimorar os conhecimentos dos paradigmas "algoritmos gulosos" e "programação dinâmica", entendendo seus conceitos e suas aplicações. Também foi possível compreender mais sobre testes e análise de complexidade de algoritmos.

Analisando os problemas em si, podemos concluir que para conhecer mais lugares é recomendável o algoritmo de programação dinâmica, pois o mesmo vai otimizar a soma dos pontos das ilhas escolhidas podendo escolher só uma de cada, o que muito provavelmente vai gerar mais ilhas que o guloso. Este por sua vez é mais recomendável para se passar mais dias viajando, pois o mesmo vai pegar a ilha com maior custo-benefício e vai selecioná-la o máximo de vezes possível (e fazendo isso com a outras que for selecionando também), não

priorizando diretamente o número de pontos, gerando assim mais dias de viagem, conhecendo provavelmente menos ilhas.