Notas de Aula - AED2 – Técnicas e Análise de Algoritmos: método guloso
Prof. Jefferson T. Oliva

Nas aulas anteriores, vimos abordagens para a análise de complexidade de algoritmos, sejam interativos ou recursivos.

Hoje e nas próximas três aulas iremos ver métodos de desenvolvimento de algoritmos.

Para o desenvolvimento de algoritmos, abordagens adequadas devem ser utilizadas. Por mais que um bom hardware ajude no desempenho, um algoritmo ainda pode ser ineficiente, pois pode demandar maior quantidade de tempo, além do necessário (para um algoritmo eficiente), para a execução de uma determinada tarefa. Por exemplo, qual algoritmo é mais eficiente para resolver um mesmo problema: um que tem a complexidade de tempo O(2^n) ou um outro $O(n^2)$? Obviamente é o algoritmo que leva menor tempo, como é o caso do que tem a complexidade de O(n^2).

Um algoritmo eficiente geralmente possui complexidade de tempo polinomial e os ineficientes, são geralmente exponenciais. Mas há casos em que a solução é naturalmente exponencial ou que não existem soluções em tempo polinomial, como o problema do caixeiro viajante, o qual **tenta determinar a menor rota para percorrer uma série de cidades (visitando uma única vez cada uma delas), retornando à cidade de origem.**

Algoritmos exponenciais são, em geral, simples: variações de pesquisa exaustiva no espaço de soluções. Essa pesquisa exaustiva é conhecida como força bruta, onde todas as possibilidades possíveis são testadas para encontrar uma solução.

Algoritmos polinomiais são obtidos através de um entendimento mais profundo da estrutura do problema.

Um problema é considerado intratável se não existe um algoritmo polinomial para resolvê-lo. Quando um problema é bem resolvido ou tratável, significa que existe uma solução polinomial para resolvê-lo. Assim, esses algoritmos são considerados eficientes.

Problemas polinomiais estão divididos em duas classes: P (polinomial determinístico) e NP (polinomial não determinístico). Um problema é dito ser de classe P caso exista um algoritmo determinístico para resolvê-lo, ou seja, além de rodar em tempo polinomial, para uma mesma entrada sempre irá produzir o mesmo resultado. A classe NP é composta por problemas que podem ser resolvidos em tempo polinomial, mas não deterministicamente. Diversos problemas famosos estão na classe NP, como o caixeiro viajante, satisfatibilidade booleana (exemplo de aplicação: avaliação de circuitos combinatórios), entre outras.

Por fim, um ou fator a ser considerado para a implementação de algoritmos é o uso da recursividade. Por mais que o uso da recursividade seja muitas vezes mais “elegante”, pode demandar uma grande quantidade de recursos computacionais, pois em cada chamada recursiva, o algoritmo é instanciado na pilha, ou seja, várias cópias de variáveis são feitas e empilhadas até que o caso base seja atingido (critério de parada). O mal uso da recursividade pode acarretar em algoritmos ineficientes.

Para projetar um algoritmo, é necessária a preocupação com a complexidade de tempo, ou seja, a preferência são as que sejam executadas em tempo polinomial.

Um exemplo é a sequência de Fibonacci, que é uma sequência de números inteiros, começando normalmente por 0 e 1, na qual, cada termo subsequente corresponde à soma dos dois anteriores.

Na equação para definir essa sequência, quando n é igual a zero, o valor da função é zero e quando n é igual a 1, o resultado é um. Em outras palavras, n = 0 e n = 1 são os casos bases da sequência de Fibonacci. Quando n > 1 (caso iterativo), o resultado é a soma dos dois últimos valores da sequência (n – 1 e n – 2).

A sequência de Fibonacci pode ser resolvida de duas formas: recursivamente e iterativamente.

O exemplo recursivo apresentado no slide 5 por mais que seja fácil de implementar e de certa forma elegante, o custo computacional é muito alto: na ordem de O(2^n), tanto para tempo quanto para espaço, ou seja, possui custo exponencial! Para n = 100, o algoritmo levaria vários anos para terminar a operação. Há pesquisadores que afirmam que esse cálculo levaria 30 milhões de anos para ser feito!

No slide 6 é apresentada uma pilha de recursões da algoritmo apresentado no slide 5. É possível perceber que são realizadas várias operações repetidas.

Em uma implementação iterativa dessa sequência, a complexidade do algoritmo é O(n)!

Apesar das desvantagens da recursão, existem problemas que são naturalmente recursivos, como a torre de Hanói.

A Torre de Hanói é também bem conhecida como torre bramanismo ou quebra-cabeças do fim do mundo. Foi publicada em 1883 pelo matemático francês Edouard Lucas, informando que o jogo era original do Vietnã e popular também da China e no Japão. O autor oferecia mais de um milhão de francos para quem resolvesse o problema da Torre de Hanói com 64 níveis, seguindo as regras do jogo. Nesse jogo, o objetivo é transferir todos os discos de uma torre para uma outra, utilizando uma outra torre como auxiliar. Nesse jogo, os discos maiores não podem ficar em cima dos menores.

Considerando a figura do slide 8, os discos devem estar na torre A, inicialmente.

No caso base desse problema, quando n = 1, a solução é simples, ou seja, basta transferi-lo para para a torre C.

No caso iterativo, se existe uma solução para n discos, então há para n – 1.

Um algoritmo para o problema é apresentado no slide 9, cujos parâmetros são as torres (“de”, “para” e “meio”) e o disco a ser movimentado (“n”). Esse algoritmo imprime todos os movimentos de discos. No algoritmo, o caso base é trivial, ou seja, basta imprimir o movimento do n-ésimo disco da torre “de” para “para”. No caso iterativo, a primeira chamada recursiva simula a movimentação do disco da torre de origem (“de”) para uma intermediária (“meio”). Na segunda chamada, é simulada a movimentação do disco da torre intermediária (“meio”) para o destino (“para”).

O número mínimo de "movimentos" para conseguir transferir todos os discos da primeira estaca à terceira é 2^n-1, sendo n o número de discos. Logo: O(2^n).

No slide 10 é apresentada a quantidade de movimentos para 5, 7, 15, 32 e 64 discos. Note que para movimentar 64 discos é necessária uma quantidade absurda de movimentações (por volta de 18 quintilhões)!

Um outro problema famoso é o Caixeiro viajante, onde o espaço de busca é o conjunto de permutações das n cidades, sendo cada permutação uma sequência de cidades a serem visitadas. Imaginem fazer essa sequência em todas as combinações possíveis para encontrar uma com o custo mínimo, que é a solução ótima!

Cada tour pode ser representada de 2n maneiras diferentes para um modelo simétrico (exemplo, um caminho pode ser 1 2 3 4 1 ou 1 4 3 2 1).

Considerando-se que há n! formas de permutar n números, então a complexidade do problema é O(n!), ou seja, fatorial.

Há inúmeros outros problemas famosos relacionados ao paradigma de projeto de algoritmos: satisfatibilidade booleana, oito rainhas, passeio do cavalo, árvore geradora mínima, caminhos mínimos, etc.

Apesar de várias soluções serem exponenciais ou fatoriais em primeira vista, alguns problemas podem ter a complexidade reduzida por meio de uso de métodos/paradigmas para o desenvolvimento de algoritmos, como: **força-bruta, método guloso, divisão e conquista, programação dinâmica**, **backtracking, e branch and bound.**

Dessas abordagens, na aula de hoje serão abordadas as técnicas de força-bruta e de método guloso.

Força-Bruta

Essa abordagem, também conhecida como “busca exaustiva” e “tentativa e erro”, enumera todas as combinações possíveis de solução, não importando se as mesmas estão corretas ou erradas. No final, é escolhida uma solução ótima (a melhor resposta, se houver) para a resolução do problema.

Entretanto, algoritmos força-bruta comumente possuem custo computacional alto, já que enumera todas as soluções possíveis, sejam elas corretas ou incorretas. É comum uma solução força-bruta ter custo exponencial (*e.g.* $O(2^n)$).

Exemplo de aplicação: problema da mochila
- Dada uma mochila que admite um determinado peso (b) e n objetos com peso p_i e custo c_i.
- Objetivo: selecionar um subconjunto de objetos que caibam dentro da mochila de forma que o valor total dos objetos sejam maximizado.

Implementação da solução para o problema da mochila (exemplo de uso: investimento capital, carregamento de veículos, orçamento, corte de material, etc):

```
int mochila_fb(int c[], int p[], int n, int b, int i, int max){
   int c1, c2;

   if (i >= n){ // Verificar se todos os objetos foram explorados
      if (b < 0)
         return 0; // se b tiver valor negativo, então será retornado 0, já que a capacidade da mochila foi extrapolada.
      else
         return max; // caso contrário, será retornado o custo máximo acumulado
   }else{
      c1 = mochila_fb(c, p, n, b, i + 1, max);  // o objeto na posição i não foi colocado na solução

      c2 = mochila_fb(c, p, n, b - p[i], i + 1, max + c[i]); // o objeto na posição i foi colocado na solução

      return c1 > c2 ? c1 : c2;

   }
}

int mochila(int c[], int p[], int n, int b){
   return mochila_fb(c, p, n, b, 0, 0);
}
```

Onde:
- c: é o vetor de custos de objetos.
- p: é o vetor de pesos de objetos.
- n: é a quantidade de objetos.
- b: é a capacidade da mochila.
- i: é o i-ésimo objeto analisado.
- max: é o valor máximo de custo acumulado.

Recorrência:
$T(n) = c$, se $n = 0$ (não há mais objeto a ser explorado)
$T(n) = 2*T(n-1)$, se $n > 0$

Utilizando o método da árvore da árvore para a solução da recorrência, temos o custo de tempo e de espaço na ordem de $O(2^n)$ para solução do problema da mochila por força-bruta.

Apesar do custo computacional alto, em alguns problemas pode ser necessária a obtenção de todas as soluções possíveis, como em diversos problemas de detecção de padrões.

Para amenizar o custo da força-bruta, para a enumeração de todas as soluções podem ser utilizadas as abordagens *backtracking* ou *branch-and-bound*.

Vantagens da força-bruta: as soluções geralmente são simples de serem implementadas e sempre é encontrada a solução ótima.

Principal desvantagem: custo computacional pode ser proibitivo.

Método Guloso

Um exemplo clássico da aplicação do método guloso é o jogo do Pac-Man, onde os fantasmas procuram caminhos aparentemente mais curtos para chegar até Pac.

Na abordagem gulosa, em cada iteração, o algoritmo seleciona o objeto mais apetitoso no momento. Esse objeto seria uma parte da solução. Por exemplo, para ir de uma cidade de origem até o destino, diversos caminhos possíveis podem ser percorridos. A partir de cada cidade, o caminho aparentemente mais curto é escolhido. Suponhamos que faremos uma viagem de Pato Branco até Cascavel e as cidades mais próximas da origem seja Francisco Beltrão (55km de Pato Branco), Itapejara (37 km), e Coronel Vivida (35 km). O próximo passo é ir até Coronel Vivida (CV), que é a cidade mais perto. Em Coronel Vivida, além de Pato Branco, Itapejara (31 km) e São João (34 km) são as cidades mais perto. Então, o próximo passo é ir até Itapejara, que é mais perto (como já passamos em Pato Branco).

Como vocês podem notar, as decisões são tomadas com base nas informações disponíveis e as consequências futuras são desconsideradas.

Também, o algoritmo guloso nunca “volta atrás” (reconsidera uma solução). O que isso quer dizer? Se estamos em São João não voltaremos para Coronel Vivida ou Pato Branco, mas seguiremos em frente, até chegar em Cascavel. Em outras palavras, sempre seguiremos em frente.

Consequentemente, a abordagem gulosa nem sempre produz as melhores soluções: ao sairmos de Pato Branco e passar nas cidades Coronel Vivida e Itapejara, percorremos um total de 66 km. Se tivéssemos saído de Pato Branco para Itapejara, teríamos economizados 29 km.

Esse exemplo que falei para vocês é bem famoso e é conhecido como caminho mais curto entre duas cidades. Uma das formas de resolver o problema é por meio da busca gulosa (heurística), que será abordado em outra disciplina (inteligência artificial), onde vocês verão algumas formas para melhorar o desempenho dessa forma de busca.

Os métodos gulosos tomam decisão de acordo com o alcance local. Por exemplo, em qual cidade vizinha irei agora a partir da cidade atual?

A solução de problemas por métodos gulosos é realizada por etapas, para os quais, em cada uma é feita:
- A escolha do melhor elemento local.
- Marcação desse elemento para não considerá-lo novamente nos próximos estágios. Caso a partir desse elemento não seja levado a uma solução, o mesmo deve ser “descartado”.
- Verificação de sua viabilidade. Será que a partir cidade atual em que me encontro pode chegar ao destino? Será que eu já não passei nas cidades vizinhas?
- Decisão de que o elemento faça parte da solução. Ainda é possível chegar até a cidade de destino (há cidades a serem exploradas para chegar ao destino)? Em outras palavras, se o caminho é viável, o mesmo pode ser adicionado na solução.

Ao final do processo, deve ser verificado se uma solução foi encontrada. Caso positivo, o método teve sucesso na busca pela solução.

Nesse contexto, o objetivo de um método guloso é minimizar (reduzir custo) ou maximizar (aumentar o benefício) a solução.

Para uma solução baseada na abordagem gulosa, um dos segredos é a informação disponível e a ordenação do conjunto entrada. No exemplo apresentado até o momento, qual é a informação mais relevante? Neste caso é a distância entre uma cidade e a sua vizinhança. Os valores de distância são ordenados crescentemente. Assim, o primeiro elemento (o de menor custo) é testado inicialmente. Os outros elementos podem ser adicionados na solução se estiverem no caminho do primeiro elemento (e apresentarem menores custos).

Como dito anteriormente (mesmo de forma implícita), algoritmos são utilizados para resolver problemas de otimização que são solucionáveis a partir de uma sequência de passos. Em outras palavras, a solução ótima é construída por etapas. Exemplo do caminho mínimo.

Por mais que uma solução possa ser encontrada, nem sempre é ótima. Por onde o caminho entre Pato Branco e Cascavel passa? Francisco Beltrão, Itapejara ou CV?

Ingredientes chaves de um algoritmo guloso:

Subestrutura ótima: em uma solução ótima para o problema há soluções ótimas para os seus subproblemas. Se há uma solução ótima para o problema, dentro dele há soluções ótimas para seus subproblemas. No exemplo apresentado até agora, considerando apenas distância local não é possível, para este caso, a definição de um caminho ótimo (menor custo).

Característica gulosa: solução ótima global pode ser produzida a partir de uma escolha ótima local.

Existem diversos problemas em que a solução gulosa pode ser aplicada eficientemente, como: problema do troco, problema da mochila, e seleção de atividades.

Problema do troco

Imagine que você trabalha em um caixa de mercado e que você gosta muito de suas moedas. Como você tem que devolver o troco, você irá querer entregar a menor quantidade possível de moedas.

Objetivo: selecionar a menor quantidade possível de moedas para um troco no valor de N.

No slide 30 é apresentada a descrição desse problema de forma mais detalhada.

Como entrada para para a solução, devem ser considerados: o valor do troco e o conjunto de n denominações de moedas ordenadas de forma decrescente (exemplo: {100, 50, 10, 5, 1}).

A estratégia gulosa é apresentada no slide 31.

- No passo i, escolher r_i = j, tal que e_j <= M e e_{j-1} > M: em outras palavras, escolher uma moeda em que M seja divisível de forma que seja possível obter um valor inteiro.
- Dividir M por e_j, onde a parte inteira da divisão é a quantidade de moedas de valor e_j.

- No próximo passo, utilizar o resto da divisão (M % e_j) para atualizar M, ou seja, o novo valor para obter o troco. No exemplo do slide, ao dividirmos 450 (M) por 100 (e_0), teremos 4 moedas, mas ainda temos 50 para obtermos em moedas.
- Aplicar esse processo até o troco ser zerado (resto de divisão for zero).

Para o conjunto de moedas apresentado, a solução é ótima. Por exemplo: para o troco de R$ 450,00 reais, a função irá retornar 5 moedas (4 de 100 e uma de 50).

Será que essa estratégia gulosa funciona para o conjunto de moedas {300, 250, 100, 1}? Se sim, a solução é ótima?
Resposta: funciona sim e a função retornaria 52 moedas para esse caso. No entanto, a solução não é ótima. A solução ótima para esse caso seriam 3 moedas (1 de 250 e duas de 100).

Como vocês viram, dependendo do câmbio (real), essa solução gulosa é ótima.

A seguir é apresentada a implementação da solução proposta:
- As moedas devem ser ordenadas de forma decrescente.
- Começando com a primeira moeda (i = 0), divida o valor do troco pelo valor da moeda i (caso o valor do troco seja maior que o da moeda)
→ Adicione a parte inteira da divisão no conjunto da solução
→ Utilize o resto da divisão na próxima iteração

```
int qtd_moedas(int v[], int n, int troco){
   int qtd = 0;
   int i;

   for (i = 0; (i < n) && (troco > 0); i++){
      qtd += troco / v[i];
      troco = troco % v[i];
   }

   if (troco == 0)
      return qtd;
   else
      return -1;
}
```

A complexidade de tempo para a solução é O(n), já que o pior caso é quando todas as moedas são percorridas no vetor. A complexidade de espaço é de Theta(n), pois é o vetor que ocupa o maior espaço.

Problema da mochila

Um problema clássico na computação!

Dada uma mochila que admite um determinado peso e há um conjunto de objetos (cada um com um peso e um valor) para serem carregados. Objetivo: selecionar um subconjunto de objetos que caibam dentro da mochila de forma que o valor total dos objetos seja maximizado.

O problema da mochila é dividido em dois subproblemas distintos:
- Mochila fracionária: os objetos podem ser particionados (e o valor será proporcional à fração do objeto), ou seja, você pode colocar um pedaço do objeto dentro da mochila
- Mochila binária: os objetos não podem ser particionados (ou estarão dentro da mochila ou fora). Problema conhecido como Problema da Mochila Binária.

Uma estratégia para a solução do problema é a ordenação da entrada pelo valor da divisão entre o valor e o peso (custo benefício) de cada objeto. Assim, acarretaria em uma solução ótima.

Algoritmo para solução do problema da mochila fracionária:
- Ordene os itens por valor/peso de forma decrescente.
- A partir do primeiro objeto, enquanto a capacidade da mochila não for alcançada, coloque o máximo possível de objetos inteiros
- Caso a mochila não estiver cheia e ainda houver objetos, adicionar uma fração do objeto de forma que a capacidade máxima da mochila seja alcançada.

A explicação da solução está entre os slides 37 e 41.

Algoritmo:

```
int mochila_g(int p[], int c[], int n, int b){
    int i = 0;
    float valor = 0;

    while ((i < n) && (p[i] <= b)){
        valor += c[i];
        b -= p[i];
        i++;
    }

    if ((b > 0) && (i < n))
        valor += (b / p[i]) * c[i];

    return valor;
}
```

A complexidade de tempo para a solução é O(n), já que o pior caso é quando todas os itens são inseridos na mochila. A complexidade de espaço é de Theta(n), pois é o vetor que ocupa o maior espaço.

Será que a solução é ótima para a mochila binária? Não! **Ver exemplo no slide 43**.

Seleção de atividades

Diversas atividades podem requerer o uso de um mesmo recurso.

Considerando sala de aula como exemplo:
- Cada atividade (aula) tem um horário de início e um horário de fim
- Só existe uma sala disponível
- Duas aulas não podem ser ministradas na mesma sala ao mesmo

Objetivo: selecionar um conjunto máximo de atividades compatíveis sem sobreposição de tempo. Ou seja, criar o maior grupo de atividades sem a haja sobreposição de tempo.
Entre os slides 48 e 50 são apresentadas 3 estratégias gulosas para a solução do problema de seleção de atividades.

Algoritmo guloso (solução ótima)
- Receber a lista de atividades ordenadas pelo horário de término
- Em cada iteração, checar se a atividade atual é compatível
- Caso a atividade seja compatível, adicione-a no conjunto solução

Exercício no slide 51.

Considerações Finais

Outros problemas podem ser resolvidos por meio de algoritmos gulosos
Árvore de Huffman
Árvore geradora mínima
Busca gulosa
Distância mínima

Vantagens do método guloso: simples implementação e rapidez dos algoritmos.

Desvantagens do método guloso: nem sempre acarreta em soluções ótimas e se não tiver um critério de parada definido corretamente, poderá acarretar em loop infinito.

Referências

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Clifford, S. Algoritmos: teoria e prática. Elsevier, 2012.

Horowitz, E., Sahni, S. Rajasekaran, S. Computer Algorithms. Computer Science Press, 1998.

Szwarcfiter, J.; Markenzon, L. Estruturas de Dados e Seus Algoritmos. LTC, 2010.

Ziviani, M. Projetos de Algoritmos: com implementações em Pascal e C. Thomson, 2004.