Técnicas de Desenvolvimento de Algoritmos (parte 1)

Prof. Jefferson T. Oliva

Algoritmos e Estrutura de Dados 2 (AE43CP)
Engenharia de Computação
Departamento Acadêmico de Informática (Dainf)
Universidade Tecnológica Federal do Paraná (UTFPR)
Campus Pato Branco

DAINF DEPARTAMENTO ACADÊMICO DE INFORMÁTICA CÂMPUS PATO BRANCO

UTFPR UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Sumário

- Paradigmas de Projeto de Algoritmos
- Força-Bruta
  - Backtracking
  - Branch-and-bound
  - Exemplos
- Método Guloso
  - Implementação de algoritmos gulosos
  - Estratégia gulosa
  - Exemplos

- O projeto de algoritmos requer abordagens adequadas:
  - Dependendo da forma de tratamento do problema, o algoritmo pode ter desempenho ineficiente
  - Em certo casos, o algoritmo pode não conseguir resolver o problema em tempo viável
- Algoritmos polinomiais *vs.* exponenciais
- Problemas tratáveis *vs.* intratáveis
- P *vs.* NP
- Algoritmos recursivos *vs.* não-recursivos

- Para projetar um algoritmo eficiente, é fundamental a preocupação com a sua complexidade

- Exemplo: sequência de Fibonacci
    - $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...$

$$F(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F(n-1) + F(n-2) & \text{se } n > 1 \end{cases}$$

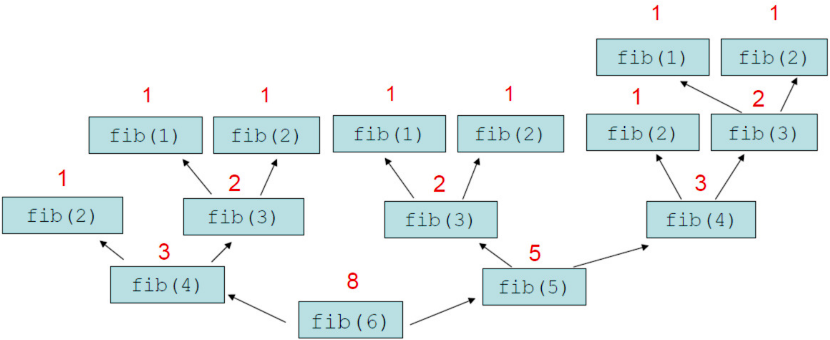
- Dado o valor de n, queremos obter o *n*-ésimo elemento da sequência

- Exemplo: sequência de Fibonacci

```
long fib(int n){
 if (n <= 0)
    return 0;
 else if (n == 1)
    return 1;
  else
    return fib(n - 1) + fib(n - 2);
}
```

  - A complexidade dessa solução é na ordem de $O(2^n)$, tanto para tempo quanto para espaço
  - Para $n = 100$, o algoritmo levaria muito tempo para executar 2^{100} operações

- Pilha de recursão para *fib*(6)



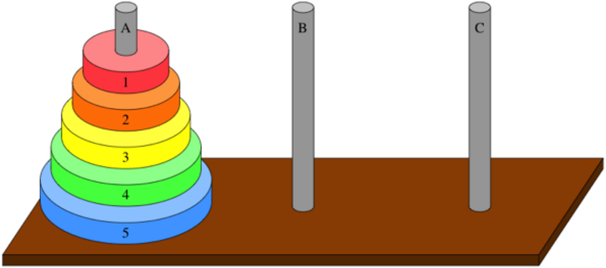
- Outra implementação da sequência de Fibonacci

```
long fib2(int n){
  int i, atual = 1;
  int p = 0; // penúltimo
  int u = 1; // último

  if (n <= 0) return 0;

  for (i = 2; i <= n; i++){
    atual = p + u;
    p = u;
    u = atual;
 }
  return atual;
}
```

- A complexidade de tempo: $O(n)$

- A complexidade de espaço: $\Theta(1)$

- Exemplo: torre de Hanói



- Dado três torres (A, B e C) e n discos de diâmetros diferentes
- O disco de menor diâmetro sempre deve estar em cima do disco de maior diâmetro
- Problema: como colocar todos os discos na torre C, utilizando a torre intermediária B, sem inverter a ordem dos diâmetros em nenhum torre?

- Exemplo: torre de Hanói
  - Inicialmente, todos discos deve estar na torre A
  - Se há solução para n discos, então há solução para $n - 1$ discos
  - No caso trivial de $n = 1$, a solução é simples
  - A solução para n discos é realizada em termos de $n - 1$

```
void hanoi(char de, char para, char meio, int
n){
  if (n <= 1)
   printf("Disco %d:  %c => %c\n", n, de, para);
  else{
   hanoi(de, meio, para, n - 1);
   printf("Disco %d:  %c => %c\n", n, de, para);
   hanoi(meio, para, de, n - 1);
  }
}
```

- Complexidade do algoritmo *Hanoi*
  - 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: $O(2^n)$
  - Para solucionar um Hanói de 5 discos, são necessários 31 movimentos
  - Hanói de 7 discos: 127 movimentos
  - Hanói de 15 discos: 32.767 movimentos
  - Hanói de 32 discos: 4.294.967.295 movimentos
  - Hanói de 64 discos: 18.446.744.073.709.551.615 movimentos!

Paradigmas de Projeto de Algoritmos

- Exemplo: problema do caixeiro viajante (PCV)

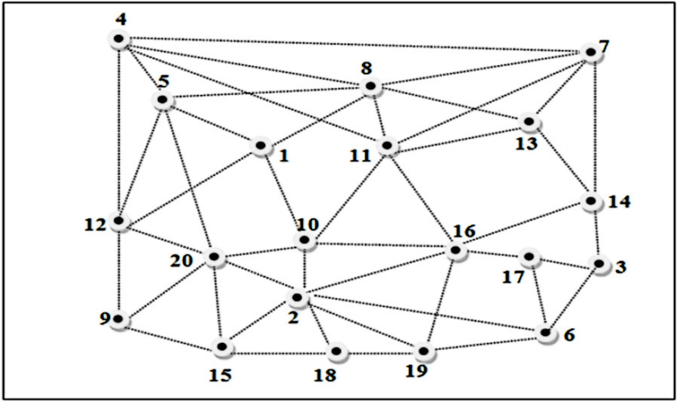


Figura 1 - Instância do PCV para 20 cidades

- Exemplo: problema do caixeiro viajante
  - Espaço de busca é um conjunto de permutação das n cidades
  - Cada permutação das n cidades caracteriza-se como uma lista ordenada que define a sequência das cidades a serem visitadas
  - A solução ótima é uma permutação que corresponda a uma *tour* (ou passeio) de caminho mínimo
  - Cada *tour* pode ser representada de $2n$ maneiras diferentes (para um modelo simétrico)
  - Considerando-se que há $n!$ formas de permutar n números, o tamanho do espaço de busca é $|S| = \frac{n!}{2n} = \frac{(n-1)!}{2}$
  - Logo, a complexidade é $O(n!)$

- Exemplos de outros problemas
  - Satisfatibilidade booleana
  - Oito rainhas
  - Passeio do cavalo
  - Árvore geradora mínima
  - Caminhos mínimos
  - ...

Paradigmas de Projeto de Algoritmos

- Métodos (Paradigmas) de desenvolvimento de algoritmos:
  - Força-bruta
  - Método guloso
  - Divisão e conquista
  - Programação dinâmica
  - *Backtracking**
  - *Branch and bound**

Força-Bruta

Força-Bruta

- Também conhecida como "busca exaustiva" e "tentativa e erro"

- É a estratégia mais trivial e intuitiva para a solução de problemas

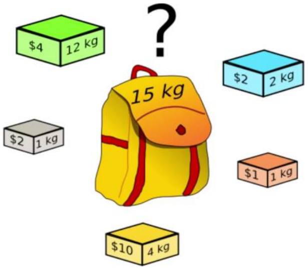
- Essa abordagem enumera todas as combinações possíveis de soluções
    - No final, é escolhida uma solução, se houver, que satisfaça o problema

    - A melhor solução é escolhida

- Entretanto, algoritmos força-bruta comumente possuem custo computacional alto
    - Muitas vezes exponenciais (*e.g.* $O(2^n)$)

Força-Bruta

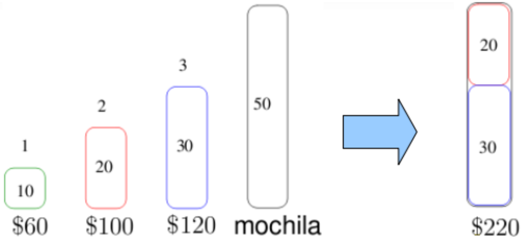
- Exemplo: problema da mochila



  - 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

Força-Bruta

- Exemplo: problema da mochila
  - Solução ótima para um conjunto de entrada



- Implementação do algoritmo mochila

```
static int mochila_fb(int c[], int p[], int n, int b, int
i, int max){
  int c1, c2;
  if (i >= n)
    return b < 0 ?  0 :  max;
  else{
    c1 = mochila_fb(c, p, n, b, i + 1, max);
    c2 = mochila_fb(c, p, n, b - p[i], i + 1, max + c[i]);
    return c1 > c2 ?  c1 :  c2;
  }
}

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

- Complexidade de tempo e de espaço: $O(2^n)$

Força-Bruta

- Apesar do custo computacional alto, em alguns problemas pode ser necessária a obtenção de todas as soluções possíveis
  - 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:
  - Simples implementação
  - Solução ótima
- Principal desvantagem:
  - Custo computacional pode ser proibitivo

Método Guloso

- Algoritmos gulosos são tipicamente usados para resolver problemas de otimização

Método Guloso

- Um algoritmo guloso escolhe, em cada iteração, o objeto mais apetitoso que vê pela frente
- O objeto selecionado passa a fazer parte da solução do problema
- As decisões são tomadas com base em informações disponíveis na iteração corrente, desconsiderando as consequências futuras
- Nunca reconsidera uma solução, independentemente das consequências
- Pode não produzir a melhor solução

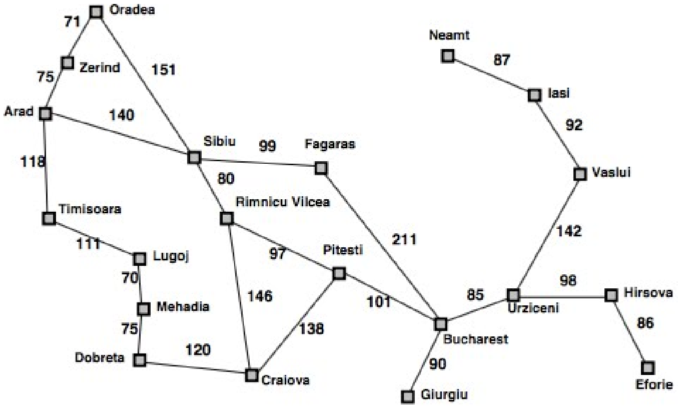
Método Guloso

- Objetivo de um algoritmo guloso pode ser:
  - Minimizar

  - Maximizar

- Construir por etapas (iteração) uma solução ótima
- Em cada iteração:
  - Selecione um elemento conforme uma função gulosa (o melhor local)
  - Marque-o para não considerá-lo novamente nos próximos estágios
  - Examine o elemento selecionado quanto sua viabilidade
  - Decida a sua participação ou não na solução
- No final, verifique se a solução foi encontrada

- Exemplo: encontrar o caminho mais curto entre duas cidades



- Um dos "segredos" dos algoritmos gulosos é a forma da ordenação/organização do conjunto de entrada
- Algoritmos gulosos são utilizados para resolver problemas de otimização que funcionem através de uma sequência de passos

- Problema do troco
- Problema da mochila
- Seleção de atividades

- Imagine que você trabalha no caixa de um supermercado
  - Após o pagamento da compra pelo cliente, você deve entregar o troco em moedas
  - Você gosta dessas moedas e quer entregar o menor número de moedas possíveis
- Objetivo: selecionar a menor quantidade possível de moedas para um troco de valor N

- **Descrição**: seja $E = \{e_1, e_2, ..., e_n\}$, $e_1 > e_2 > ... > e_n$, um conjunto de n denominações de moedas (ou cédulas), e M um valor positivo que representa o troco

- **Problema**: fornecer o montante M com o menor número de moedas

- **Sequência de decisões**: escolher r_1, depois r_2, ...
  - $r_i = j$, tal que $e_j \leq M$ e $e_{j-1} > M$

- Seja $E = \{e_1, e_2, ..., e_n\}$, e M, um valor positivo que representa o troco

- Algoritmo, supondo que E esteja ordenado de forma decrescente
  - No passo i, escolher $r_i = j$, tal que $e_j \leq M$ e $e_{j-1} > M$
  - Dividir M por e_j
  - No próximo passo, utilizar o resto da divisão ($M\%e_j$)
  - Aplicar esse processo até o troco ser zerado (resto de divisão for zero) ou todas as moedas terem sido percorridas

- Suponha que um valor de 450 deve ser devolvido como troco:
  - Será que a estratégia gulosa apresentada funciona para o conjunto de moedas $E = \{100, 50, 25, 10, 5, 1\}$? Caso positivo, a resposta é ótima?
  - Será que a estratégia gulosa apresentada funciona para o conjunto de moedas $E = \{300, 250, 100, 1\}$? Caso positivo, a resposta é ótima?

- Suponha que um valor de 450 deve ser devolvido como troco:
  - Será que a estratégia gulosa apresentada funciona para o conjunto de moedas $E = \{100, 50, 25, 10, 5, 1\}$? Caso positivo, a resposta é ótima?
    - A estratégia, além de funcionar, retorna uma solução ótima: cinco (quatro moedas de 100 e uma de 50)
  - Será que a estratégia gulosa apresentada funciona para o conjunto de moedas $E = \{300, 250, 100, 1\}$? Caso positivo, a resposta é ótima?
    - A estratégia funciona (encontra uma solução), mas não retorna uma solução ótima: 52 moedas (1 moeda de 300, uma de 100 e 50 de 1)
- Dependendo do câmbio (*e.g.* real), a solução gulosa é ótima

- Algoritmo
  - Ordene o valor das moedas 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

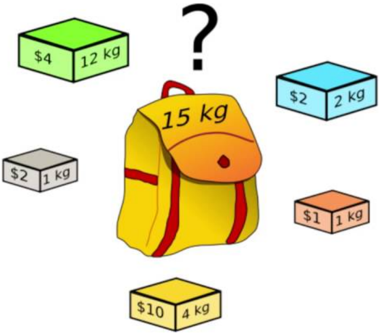
- Implementação do algoritmo troco mínimo

```
// Supõe-se que o vetor moedas esteja ordenado
int qtd_moedas(int moedas[], int n, int troco){
  int i, n_moedas = 0;
  for (i = 0; i < n && troco > 0; i++){
    // atualizar a quantidade de moedas de troco
    n_moedas += troco / moedas[i];
    // atualizar o valor do troco faltante
    troco = troco % moedas[i];
  }
  if (troco > 0)
    return n_moedas; // solução encontrada
  else
    return -1; // solução não encontrada
}
```

- Complexidade de tempo: $O(n)$
- Complexidade de espaço (se considerador a entrada): $\Theta(n)$
- Complexidade de espaço (espaço extra): $\Theta(1)$

- Exercício: adapte o algoritmo anterior para retornar o conjunto de moedas utilizadas para o troco. Por exemplo, para $moedas = \{100, 50, 10, 5, 1\}$ e $troco = 450$ deve ser retornada a seguinte sequência: $\{100, 100, 100, 100, 50\}$.

- Dados
  - Uma mochila que admite um determinado peso
  - Um conjunto de objetos, sendo cada com um valor e um peso



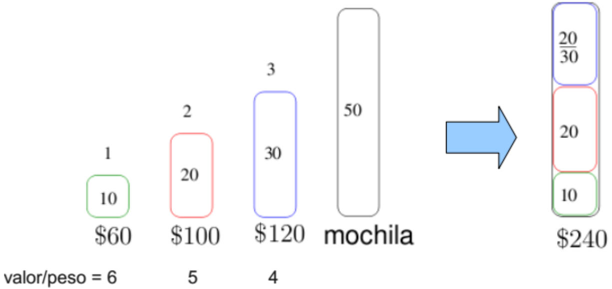
- Objetivo: selecionar um subconjunto de objetos que caibam dentro da mochila de forma que o valor total dos objetos seja maximizado

- O problema da mochila é dividido em dois subproblemas distintos
  - Mochila fracionária: os objetos podem ser particionados de forma proporcional
  - Mochila binária: os objetos não podem ser particionados

- A entrada pode ser ordenada por valor/peso
- Solução gulosa é ótima
- Algoritmo
  - Ordene os itens por valor/peso de forma decrescente
  - Começando com o primeiro objeto ($i = 0$), coloque o máximo do mesmo que estiver disponível e for possível
  - Se a mochila ainda não estiver cheia, passe para o próximo item

- Exemplo

- Exemplo



- O algoritmo guloso retorna solução ótima para o problema da mochila fracionária

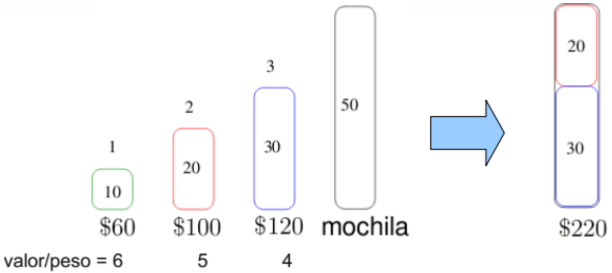
- Implementação do algoritmo mochila fracionária

```
int mochila_g(int peso[], int custo[], int n, int
capacidade){
  int i = 0;
  float valor = 0;
  while ((i < n) && (peso[i] <= capacidade)){
    valor += custo[i];
    capacidade -= peso[i];     i++;
  }
  if ((capacidade > 0) && (i < n))
    valor += (capacidade / peso[i]) * custo[i];

  return valor;
}
```

- Complexidade de tempo: $O(n)$
- Complexidade de espaço (considerando a entrada): $\Theta(n)$
- Complexidade de espaço (espaço extra): $\Theta(1)$

- Exemplo

- Solução ótima do exemplo

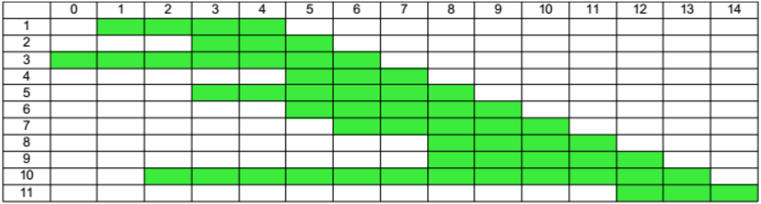


- O algoritmo guloso pode não gerar uma solução ótima para o problema da mochila binária

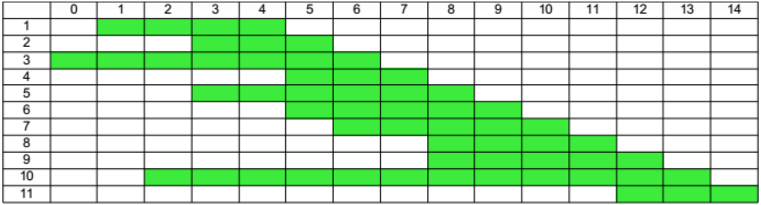
- Diversas atividades podem requerer o uso de um mesmo recurso

- Considerando 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 tempo

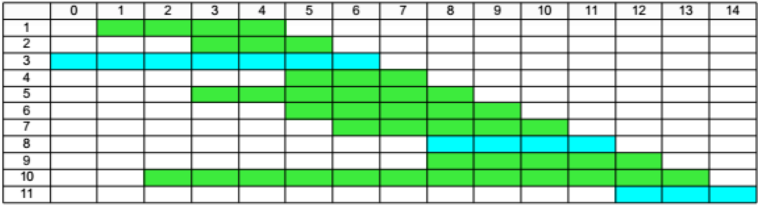
- Exemplo para 11 atividades e 14 unidades de tempo



- Considerando aula como exemplo:
  - Objetivo: selecionar um conjunto máximo de atividades compatíveis
    - Sem sobreposição de tempo
    - Criação do maior grupo de atividades sem sobreposição de tempo

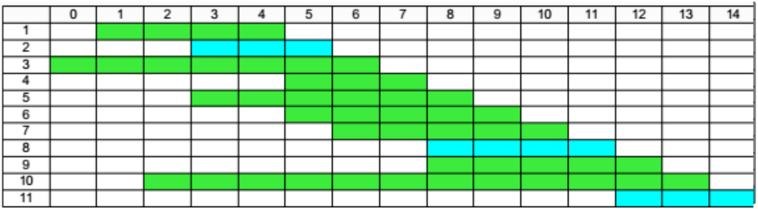


- Como fazer a seleção de atividades?
  - Estratégia 1: selecionar as atividades que começam primeiro



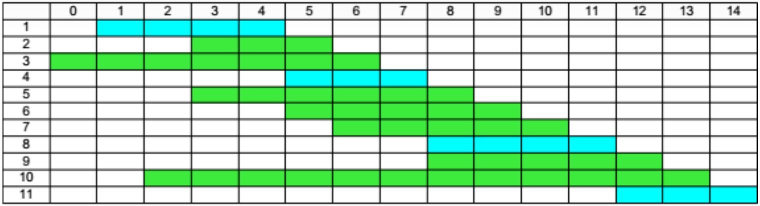
  - Solução não ótima

- Como fazer a seleção de atividades?
  - Estratégia 2: selecionar as atividades que são executadas em menos tempo



  - Solução não ótima

- Como fazer a seleção de atividades?
    - Estratégia 3: escolher as atividades que terminam primeiro



- Solução ótima

- Algoritmo guloso
  - 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**: implemente uma solução gulosa para o problema de seleção de atividades. A função deverá receber como entrada: vetor de horário de início, vetor de horário de término, tamanho dos vetores (obs.: os vetores poderão ser de números inteiros, em vez de itens no formato hh:mm). Como saída, a função deverá retornar a quantidade de atividades alocadas.

Considerações Finais

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

Considerações Finais

- Vantagens:
  - Simples implementação
  - Rápida execução
- Desvantagens:
  - Pode não gerar soluções ótimas
  - Pode entrar em *loop* infinito

Referências I

- 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, N.
  Projeto de Algoritmos - com implementações em Java e C++.
  Thomson, 2007.