Técnicas de Desenvolvimento de Algoritmos (parte 1)

Prof. Marcelo Rosa

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

Sumário

Sumário

Paradigmas de Projeto de Algoritmos

- 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
  - **Um problema é considerado intratável se não existe um algoritmo polinomial para resolvê-lo**
- Algoritmos recursivos *vs.* não-recursivos

Paradigmas de Projeto de Algoritmos

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

Sequência de Fibonacci

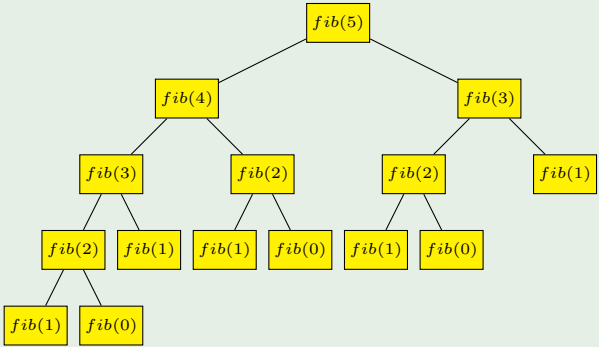
- $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...$

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

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

Sequência de Fibonacci

```
1  long int fib(int n){
2     if (n <= 0)
3        return 0;
4     else if (n == 1)
5        return 1;
6     else
7        return fib(n-1) + fib(n-2);
8  }
```



- 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

Sequência de Fibonacci

- Outra implementação da sequência de Fibonacci

```
1  long int fib2(int n){
2  int i, atual = 1;\\
3  int p = 0; // penúltimo
4  int u = 1; // último
5     if (n <= 0)
6        return 0;
7     for (i = 2; i <= n; i++){
8        atual = p + u;
9        p = u;
10       u = atual;
11    }
12    return atual;
13 }
```

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

Problema do Caixeiro Viajante (*Traveling Salesman Problem*)

Dado um conjunto de **cidades** e as **distâncias** (ou custos) entre cada par delas, determine o **menor caminho possível** que permite ao caixeiro:

1. **Visitar cada cidade exatamente uma vez**
2. **Retornar à cidade de origem**

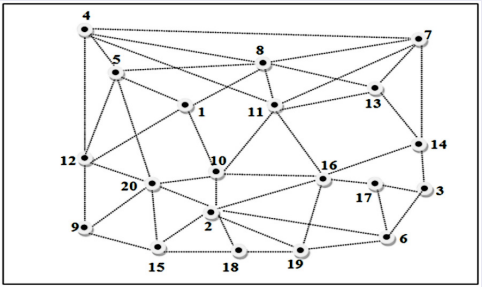


Figura 1 - Instância do PCV para 20 cidades

- 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 $O(n!)$

Paradigmas de Projeto de Algoritmos

- Métodos (Paradigmas) de desenvolvimento de algoritmos:
  - Força-bruta
  - Método guloso
  - Divisão e conquista
  - Programação dinâmica

Sumário

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.* $O(2^n)$)

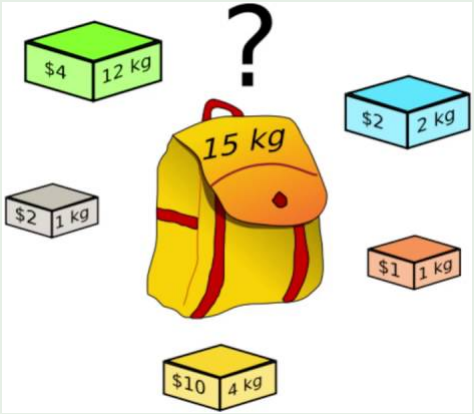
Problema da Mochila (0-1)

Dado um conjunto $I = \{1, 2, \ldots, n\}$ de n itens, onde cada item $i \in I$ possui um peso w_i e um valor v_i associados. Para uma dada mochila com capacidade limitada de peso W, selecionar um subconjunto de itens $I' \subseteq I$ tal que

- $\sum_{i \in I'} w_i \leq W$ (a soma dos pesos não ultrapassa a capacidade)
- $\sum_{i \in I'} v_i$ é máxima (**maximiza o valor total**)

Capacidade da mochila: 15 kg

Item	Peso (kg)	Valor (R$)
A	12	4
B	1	2
C	4	10
D	1	1
E	2	2



Força-Bruta

Problema da Mochila (*knapsack problem*)

Capacidade da mochila: 15 kg

- Espaço de estados: todas as combinações

A	B	C	D	E	Peso	Valor	Viável?
0	0	0	0	0	0	0	Sim
0	0	0	0	1	2	2	Sim
0	0	0	1	0	1	1	Sim
0	0	0	1	1	3	3	Sim
0	0	1	0	0	4	10	Sim
0	0	1	0	1	6	12	Sim
0	0	1	1	0	5	11	Sim
0	0	1	1	1	7	13	Sim
0	1	0	0	0	1	2	Sim
0	1	0	0	1	3	4	Sim
0	1	0	1	0	2	3	Sim
0	1	0	1	1	4	5	Sim
0	1	1	0	0	5	12	Sim
0	1	1	0	1	7	14	Sim
0	1	1	1	0	6	13	Sim
0	1	1	1	1	8	15	Sim

A	B	C	D	E	Peso	Valor	Viável?
1	0	0	0	0	12	4	Sim
1	0	0	0	1	14	6	Sim
1	0	0	1	0	13	5	Sim
1	0	0	1	1	15	7	Sim
1	0	1	0	0	16	14	Não
1	0	1	0	1	18	16	Não
1	0	1	1	0	17	15	Não
1	0	1	1	1	19	17	Não
1	1	0	0	0	13	6	Sim
1	1	0	0	1	15	8	Sim
1	1	0	1	0	14	7	Sim
1	1	0	1	1	16	9	Não
1	1	1	0	0	17	16	Não
1	1	1	0	1	19	18	Não
1	1	1	1	0	18	17	Não
1	1	1	1	1	20	19	Não

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

Força-Bruta

- Vantagens:
  - Simples implementação
  - Solução ótima
- Principal desvantagem:
  - Custo computacional pode ser proibitivo

Sumário

Método Guloso (*Greedy Algorithms*)

- Algoritmos gulosos são tipicamente usados para resolver problemas de otimização

Método Guloso

- Constrói uma solução por meio de uma sequência de decisões que visam o melhor cenário de curto prazo.
- 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
  - muita vezes, na verdade, não produz!

Método Guloso

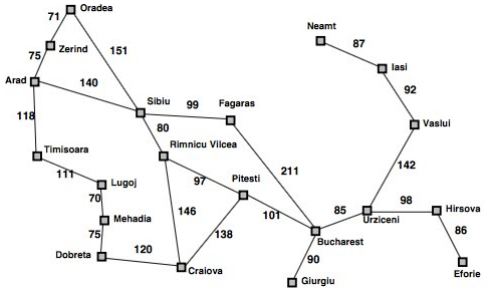
- Objetivo de um algoritmo guloso pode ser:
  - Minimizar
  - Maximizar

- Construir por etapas (iteração) uma solução (sub)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

Método Guloso

Exemplo: encontrar o caminho mais curto entre duas cidades

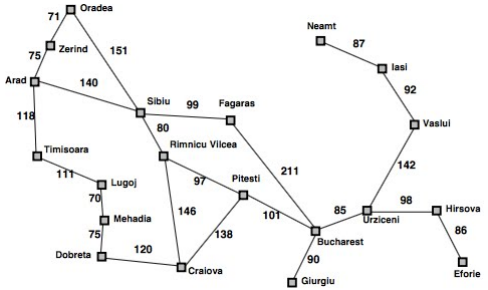
- **Problema**: Encontrar um caminho de **Arad** até **Bucharest**.
- **Estratégia Gulosa**: Escolhe sempre a estrada que parece mais promissora no momento atual.



Método Guloso

Exemplo: encontrar o caminho mais curto entre duas cidades

- **Problema**: Encontrar um caminho de **Arad** até **Bucharest**.
- **Estratégia Gulosa**: Escolhe sempre a estrada que parece mais promissora no momento atual.



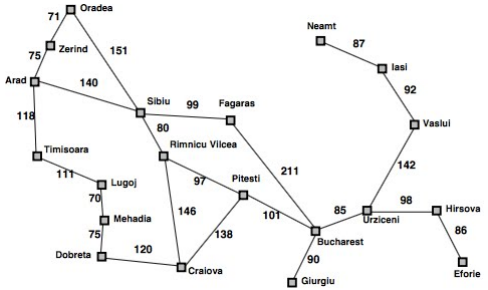
Algoritmo Gulosos

- Arad → Zerid → Oradea → Sibiu → Rimniu Vilcea → Pitesti → Bucharest.
- Total = 75 + 71 + 151 + 80 + 97 + 101 = 484 km

Método Guloso

Exemplo: encontrar o caminho mais curto entre duas cidades

- **Problema**: Encontrar um caminho de **Arad** até **Bucharest**.
- **Estratégia Gulosa**: Escolhe sempre a estrada que parece mais promissora no momento atual.



Algoritmo Gulosos

- Arad → Zerid → Oradea → Sibiu → Rimniu Vilcea → Pitesti → Bucharest.
- Total = 75 + 71 + 151 + 80 + 97 + 101 = 484 km

Solução ótima

- Arad → Sibiu → Rimniu Vilcea → Pitesti → Bucharest.
- Total = 140 + 80 + 97 + 101 = 418 km

- 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

Sejam:

- $M = \{m_1, m_2, \ldots, m_n\}$ o conjunto de valores de moedas disponíveis, com $m_i \in \mathbb{N}^+$;
- $V \in \mathbb{N}^+$ o valor total para o qual se deseja fornecer o troco.

Deseja-se encontrar um vetor de inteiros não negativos:

$$(x_1, x_2, \ldots, x_n) \in \mathbb{N}^n$$

tal que:

$$\sum_{i=1}^{n} x_i \cdot m_i = V$$

Objetivo: minimizar o número total de moedas usadas:

$$\min \left(\sum_{i=1}^{n} x_i \right)$$

- **Descrição**: seja $M = \{m_1, m_2, ..., m_n\}$, $m_1 > m_2 > ... > m_n$, um conjunto de n denominações de moedas (ou cédulas), e V um valor positivo que representa o troco
- **Problema**: fornecer o montante V com o menor número de moedas
- **Sequência de decisões**: escolher r_1, depois r_2, ...
  - $r_i = j$, tal que $m_j \leq M$ e $m_{j-1} > M$

- Seja $M = \{m_1, m_2, ..., m_n\}$, e V, um valor positivo que representa o troco
- Algoritmo, supondo que M esteja ordenado de forma decrescente
  - No passo i, escolher $r_i = j$, tal que $m_j \leq M$ e $m_{j-1} > M$
  - Dividir M por m_j
  - No próximo passo, utilizar o resto da divisão ($M\%m_j$)
  - 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 $M = \{100, 50, 25, 10, 5, 1\}$? 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 $M = \{100, 50, 25, 10, 5, 1\}$? Caso positivo, a resposta é ótima?
  - A estratégia, além de funcionar, retorna uma solução ótima: cinco (quatro moedas de 100 e uma de 50)

- Suponha que um valor de 450 deve ser devolvido como troco:

- Será que a estratégia gulosa apresentada funciona para o conjunto de moedas $M = \{100, 50, 25, 10, 5, 1\}$? 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?

- Suponha que um valor de 450 deve ser devolvido como troco:

- Será que a estratégia gulosa apresentada funciona para o conjunto de moedas $M = \{100, 50, 25, 10, 5, 1\}$? 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)
  - Solução ótima: 4 moedas (1 moeda de 250 e 2 de 100).

- Suponha que um valor de 450 deve ser devolvido como troco:

- Será que a estratégia gulosa apresentada funciona para o conjunto de moedas $M = \{100, 50, 25, 10, 5, 1\}$? 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)
  - Solução ótima: 4 moedas (1 moeda de 250 e 2 de 100).

- Dependendo do câmbio (*e.g.* real), a solução gulosa é ótima

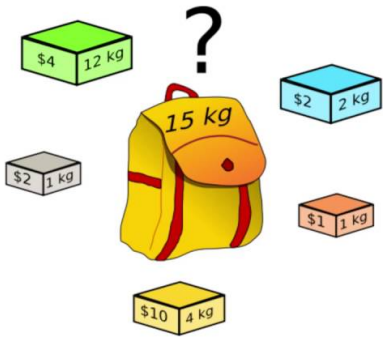
- Implementação do algoritmo troco mínimo

```
// 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: $O(n)$

- 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



Problema da Mochila (fracionária)

Dado um conjunto $I = \{1, 2, \ldots, n\}$ de n itens, onde cada item $i \in I$ possui um peso w_i e um valor v_i associados. Para uma dada mochila com capacidade limitada de peso W, selecionar uma sequência $S = (f_1, f_2, \ldots, f_n)$ em que $f_i \in [0, 1]$, para todo $i \in I$, tal que

- $\sum_{i \in I} f_i w_i \leq W$ (a soma dos pesos não ultrapassa a capacidade)
- $\sum_{i \in I} f_i v_i$ é máxima (**maximiza o valor total**)

Exemplo

Mochila: $W = 60$

Item 1: $v_1 = 60$, $w_1 = 10$

Item 2: $v_2 = 150$, $w_2 = 30$

Item 3: $v_3 = 120$, $w_3 = 30$

Item 4: $v_4 = 160$, $w_4 = 40$

Item 5: $v_5 = 200$, $w_5 = 50$

Item 6: $v_6 = 150$, $w_6 = 50$

Item 7: $v_7 = 60$, $w_7 = 60$

$S_1 = (0, 0, 0, 0, 0, 1, 0)$, peso $= 60$, valor $= 60$

$S_2 = (0, 0, 0, 0, 0, 1, 0)$, peso $= 50$, valor $= 150$

$S_3 = (0, 0, 0, 0, 1, 0, 0)$, peso $= 50$, valor $= 200$

$S_4 = (0, 0, 0, \frac{1}{4}, 1, 0, 0)$, peso $= 60$, valor $= 240$

$S_5 = (0, 1, 1, 0, 0, 0, 0)$, peso $= 60$, valor $= 270$

$S_6 = (1, 1, \frac{1}{3}, 0, \frac{1}{5}, 0, 0)$, peso $= 60$, valor $= 290$

$S_7 = (1, 1, 0, 0, \frac{2}{5}, 0, 0)$, peso $= 60$, valor $= 290$

$S_8 = (1, 1, \frac{2}{3}, 0, 0, 0, 0)$, peso $= 60$, valor $= 290$

Problema da mochila fracionária: algoritmo

1 Ordene os itens pela razão valor/peso e os renomeie de forma que $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$
2 $capacidade = W$
3 $i = 1$
4 **enquanto** $i \leq n$ *e* $capacidade \geq w_i$ **faça**
5 $\quad f_i = 1$
6 $\quad capacidade = capacidade - w_i$
7 $\quad i = i + 1$
8 **se** $i \leq n$ **então**
9 $\quad f_i = capacidade/w_i$
10 **para** $j = i + 1$ *até* n, *incrementando* **faça**
11 $\quad f_j = 0$
12 **devolve** $(f_1, \ldots, f_n)$

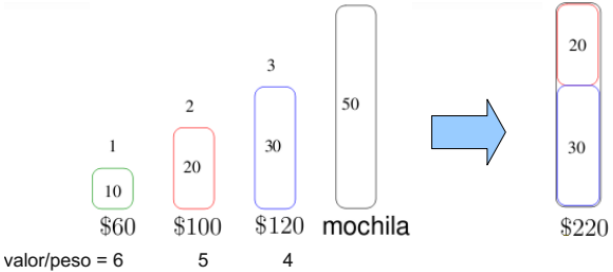
Problema da mochila fracionária: algoritmo

1 Ordene os itens pela razão valor/peso e os renomeie de forma que $v_1/w_1 \geq v_2/w_2 \geq \cdots \geq v_n/w_n$
2 $capacidade = W$
3 $i = 1$
4 **enquanto** $i \leq n$ *e* $capacidade \geq w_i$ **faça**
5 $\quad f_i = 1$
6 $\quad capacidade = capacidade - w_i$
7 $\quad i = i + 1$
8 **se** $i \leq n$ **então**
9 $\quad f_i = capacidade/w_i$
10 **para** $j = i + 1$ *até* n, *incrementando* **faça**
11 $\quad f_j = 0$
12 **devolve** $(f_1, \ldots, f_n)$

- O algoritmo guloso retorna solução ótima para o problema da mochila fracionária
- Complexidade de tempo: $O(n)$

- Exemplo

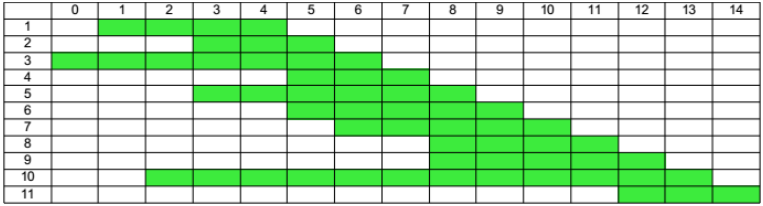
- Solução ótima do exemplo



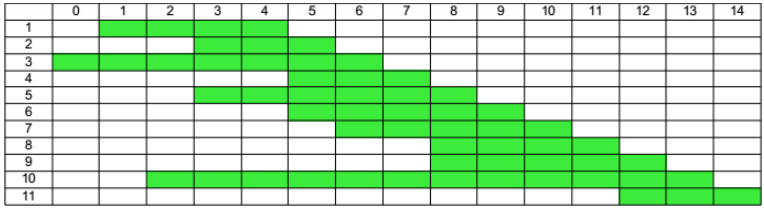
- O algoritmo guloso pode não gerar uma solução ótima para o problema da mochila binária

- 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

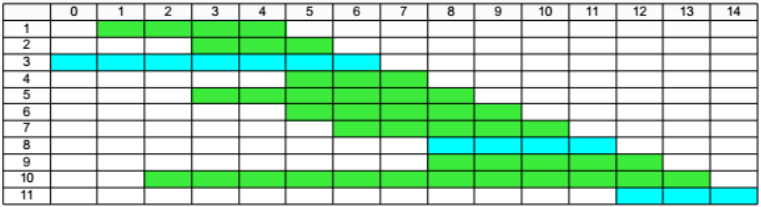
- Exemplo para 11 atividades e 14 unidades de tempo



- 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

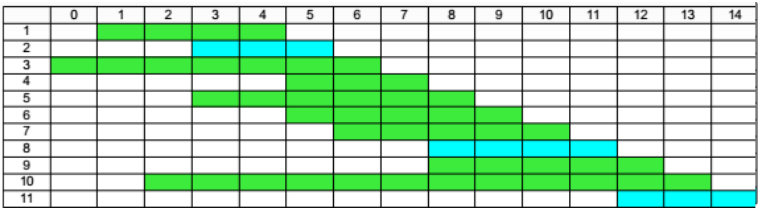


- Como fazer a seleção de atividades?
  - Estratégia 1: selecionar as atividades que começam primeiro



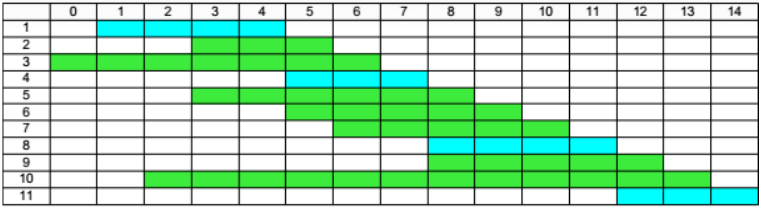
  - Solução não ótima

- 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

- 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

- Vantagens:
  - Simples implementação
  - Rápida execução
- Desvantagens:
  - Pode não gerar soluções ótimas
  - Pode entrar em *loop* infinito, se não detectar a expansão de estados repetido

Referências I


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


Oliva, J. T.
Árvores B. AE23CP - Algoritmos e Estrutura de Dados II.
Slides. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2025.