Departamento de Computação

Programação Competitiva unesp

Algoritmos Gulosos (Greedy) Divisão e Conquista

Laboratório de Programação Competitiva I

Pedro Henrique Paiola

Rene Pegoraro

Wilson M Yonezawa

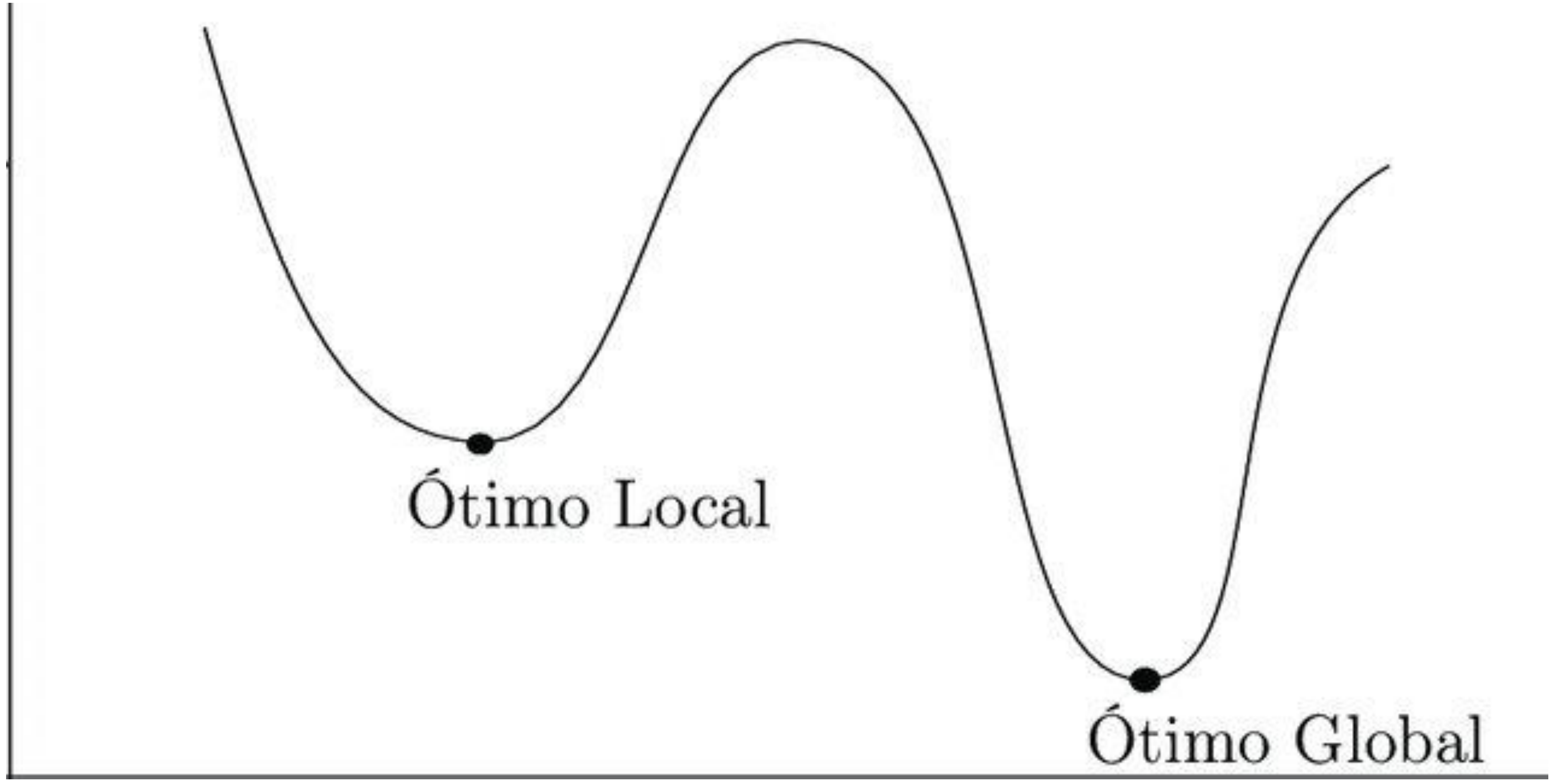
Algoritmos Gulosos

- Um algoritmo guloso (ou ganancioso, *greedy algorithm*) é um algoritmo que constrói uma solução para um problema, passo-a-passo, sempre fazendo as **escolhas que parecem mais vantajosas naquele momento**.
- Um algoritmo guloso nunca se arrepende, não desfaz escolhas já feitas
- É um algoritmo "míope", ele toma decisões com base nas informações disponíveis na iteração corrente, sem olhar as consequências que essas decisões terão no futuro.

Algoritmos Gulosos

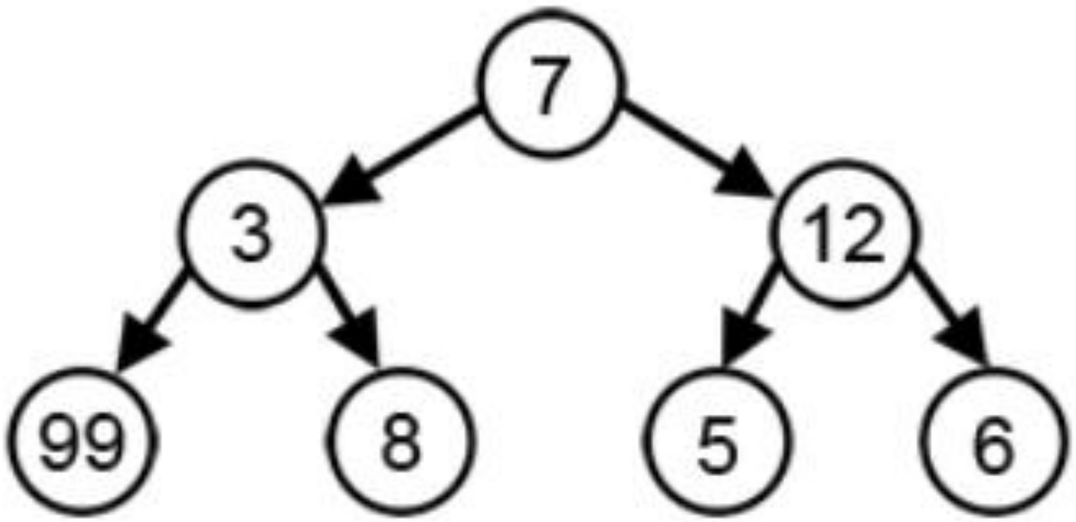
- Vantagens:
  - Implementações simples, normalmente
  - Algoritmos eficientes
- Desvantagens:
  - Nem sempre conduz a soluções ótimas globais
  - Quando conduz, a prova costuma ser difícil

Algoritmos Gulosos



Algoritmos Gulosos

- Problema do caminho de maior soma



Algoritmos Gulosos – Problema do Troco

- **Problema:** dar troco de um valor **x** com o menor número de moedas possíveis.
  - Já vimos a solução utilizando *backtracking*.
  - Nesta solução, vimos que era uma boa estratégia escolher sempre a maior moeda possível, pois isso levaria a solução que utiliza menos moedas mais rapidamente.
  - Porém, não tínhamos certeza se isso levaria a solução diretamente, por isso diversas outras possibilidades ainda eram avaliadas.

Algoritmos Gulosos – Problema do Troco

- **Problema:** dar troco de um valor **x** com o menor número de moedas possíveis.
  - Utilizando uma abordagem gulosa, vamos tentar considerar sempre uma **única opção**: escolher a moeda de maior valor possível

Algoritmos Gulosos – Problema do Troco

- Exemplos: Suponha que temos disponíveis moedas de 1, 5, 10 e 25 centavos.

41 centavos

41 - 25 = 16

16 - 10 = 6

6 - 5 = 1

1 - 1 = 0

41 = 25 + 10 + 5 + 1

Algoritmos Gulosos – Problema do Troco

- Exemplos: Suponha que temos disponíveis moedas de 1, 5, 10 e 25 centavos.

59 centavos

59 - 25 = 34

34 - 25 = 9

9 - 5 = 4

4 - 1 = 3

3 - 1 = 2

2 - 1 = 1

1 - 1 = 0

59 = 25 + 25 + 5 + 1 + 1 + 1 + 1

Algoritmos Gulosos – Problema do Troco

- Contra-exemplo: Suponha que estamos em um país onde existem apenas as moedas de 1, 5 e 8 centavos.

  11 centavos

  11 - 8 = 3

  3 - 1 = 2

  2 - 1 = 1

  1 - 1 = 0

  11 = 8 + 1 + 1 +1

- PORÉM, poderíamos obter um troco com 3 moedas: 11 = 5 + 5 + 1

Algoritmos Gulosos – Problema do Troco

- Quando o algoritmo guloso falha para o problema do troco?
- Quando existem moedas **x**, **y** tal que **x < y** e **2x > y**.

- Moedas: 1, 5, 10, 25, 50 OK
- Moedas: 1, 5, 8 Falha

Maximum product subset of an array

- **Problema:** encontrar o maior produto possível de um subconjunto de elementos de um vetor de inteiros.
- Exemplos

```
Entrada: a[] = {-5, 0, 2, 5, 5}
Saída: 50 = 2 * 5 * 5

Entrada: a[] = {-1, 0}
Saída: 0

Entrada: a[] = {-1, -1, -2, 4, 3}
Saída: 24 = (-1) * (-2) * 4 * 3
```

Maximum product subset of an array

- Algoritmo por força bruta ou *backtracking*: testar todos os subconjuntos possíveis.

- Algoritmo guloso: para o algoritmo guloso temos que nos basear nos seguintes fatos:
  a) Se temos números positivos: selecionamos todos eles.
  b) Se temos uma quantidade par de números negativos: selecionamos todos eles.
  c) Se temos uma quantidade ímpar de números negativos: selecionamos todos, com exceção do maior (com menor valor absoluto).
  d) Não escolhemos nenhum zero, a não ser que só tenhamos zeros, com no máximo um número negativo

Maximum product subset of an array

Entrada: a[] = {-5, 0, 2, 5, 5}

Saída: 50 = 2 * 5 * 5

Entrada: a[] = {-1, 0}

Saída: 0 = 0

Entrada: a[] = {-1, -1, -2, 4, 3}

Saída: 24 = (-1) * (-2) * 4 * 3

Entrada: a[] = {-4, -5, 0, 2, 3}

Saída: 120 = (-4) * (-5) * 2 * 3

Problema das Tarefas Compatíveis

- **Problema:** suponha um conjunto T = $\{t_1, t_2, ..., t_n\}$ de **n** tarefas propostas que desejam um recurso (como uma sala de conferências), o qual só pode ser utilizado por uma única tarefa de cada vez.

Problema das Tarefas Compatíveis

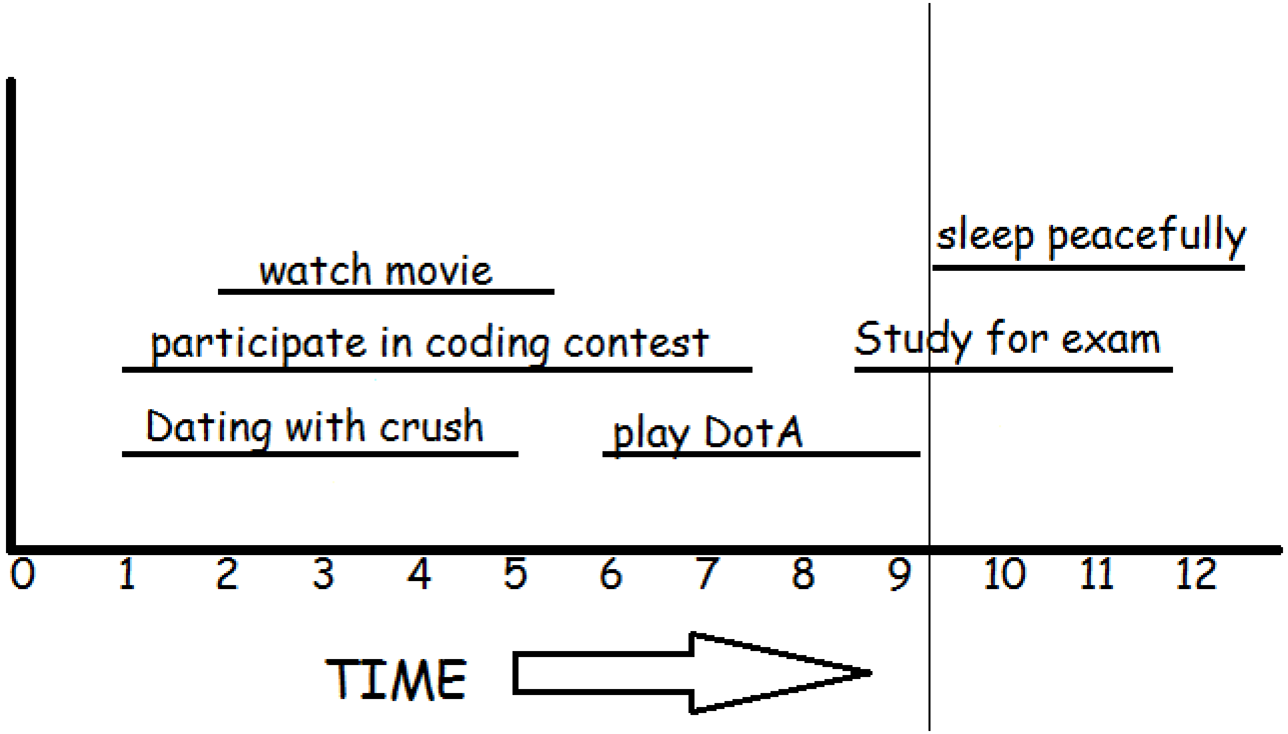
- **Problema:** suponha um conjunto $T = \{t_1, t_2, ..., t_n\}$ de **n** tarefas propostas que desejam um recurso (como uma sala de conferências), o qual só pode ser utilizado por uma única tarefa de cada vez.
  - Cada tarefa t_i tem um **tempo de início s_i** e um **tempo de término f_i**, em que $s_i < f_i$.
  - As tarefas t_i e t_j são compatíveis sse os intervalos $[s_i, f_j)$ e $[s_j, f_j)$ não se sobrepõem ($s_i >= f_j$ ou $s_j >= f_i$)

Problema das Tarefas Compatíveis

- **Problema:** suponha um conjunto $T = \{t_1, t_2, ..., t_n\}$ de **n** tarefas propostas que desejam um recurso (como uma sala de conferências), o qual só pode ser utilizado por uma única tarefa de cada vez.
  - Cada tarefa t_i tem um **tempo de início s_i** e um **tempo de término f_i**, em que $s_i < f_i$.
  - As tarefas t_i e t_j são compatíveis sse os intervalos $[s_i, f_i)$ e $[s_j, f_j)$ não se sobrepõem ($s_i >= f_j$ ou $s_j >= f_i$)
- **Objetivo**: selecionar um subconjunto de tamanho máximo de tarefas mutuamente compatíveis.

Problema das Tarefas Compatíveis

- BUSYMAN – I AM VERY BUSY (Spoj)



Problema das Tarefas Compatíveis

- **Solução por força bruta**: testar todos os possíveis subconjuntos.
- **Estratégia gulosa**: vamos tentar pensar em critérios simples de seleção de tarefas, e verificar o que acontece:
  - Selecionar a tarefa de menor duração
  - Selecionar a tarefa de menor s_i
  - Selecionar a tarefa de menor f_i

Problema das Tarefas Compatíveis

- Selecionar a tarefa de menor duração

- Selecionar a tarefa de menor s_i

- Selecionar a tarefa de menor f_i

Problema das Tarefas Compatíveis

- Selecionar a tarefa de menor duração

- Selecionar a tarefa de menor s_i

- Selecionar a tarefa de menor f_i

Problema das Tarefas Compatíveis

- De fato, este problema pode ser resolvido utilizando um algoritmo guloso em que a próxima atividade i selecionada é a que possui menor tempo f_i e é compatível com a anterior j ($s_i >= f_j$)

Problema das Tarefas Compatíveis

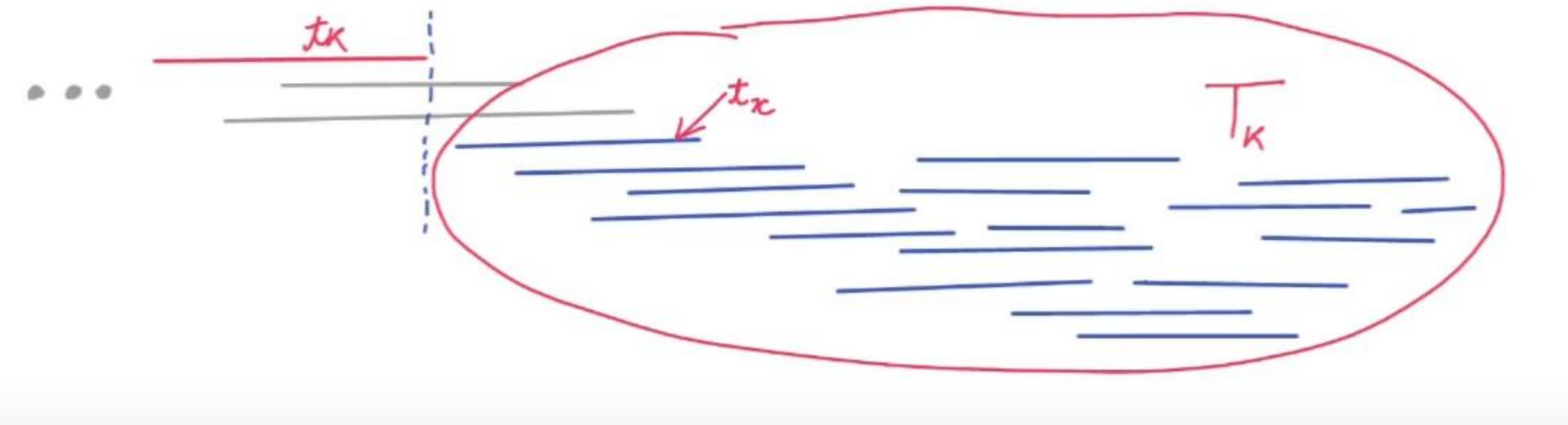
```
Escalona(T, s, f, n)
   Ordene as tarefas em ordem crescente de tempo final
   S = {t_1}
   k = 1
   para i = 2 até n faça
      se s_i >= f_k então
            S = S ∪ {t_i}
            k = i
   retorna S
```

Problema das Tarefas Compatíveis

- **Teorema**: dado um conjunto $T = \{t_1, t_2, ..., t_n\}$ com **n** tarefas onde cada t_i é realizado no intervalo $[s_i, f_i)$, o algoritmo Escalona(T, s, f, n) devolve uma solução ótima para o problema de tarefas compatíveis.
- Para demonstrar este teorema, tentaremos mostrar que qualquer tarefa escolhida por este algoritmo está em uma solução ótima. Sendo assim, por indução, ele sempre levará a uma solução ótima.

Problema das Tarefas Compatíveis

- **Demonstração:** para qualquer $t_k \in T$, seja $T_k = \{t_i \in T \mid s_i \geq f_k\}$. Seja $t_x \in T_k$ uma tarefa que termina primeiro em T_k.



Problema das Tarefas Compatíveis

- **Demonstração:** vamos supor que t_x não está em uma solução ótima. Seja então $S_k \subseteq T_k$ uma solução ótima para T_k e assuma que $t_x \notin S_k$.

Problema das Tarefas Compatíveis

- **Demonstração:** vamos supor que t_x não está em uma solução ótima. Seja então $S_k \subseteq T_k$ uma solução ótima para T_k e assuma que $t_x \notin S_k$.
- Seja $t_y \in S_k$ uma tarefa que termina primeiro em S_k. Podemos definir então o conjunto $S_k' = S_k - \{t_y\} \cup \{t_x\}$.

Problema das Tarefas Compatíveis

- **Demonstração:** vamos supor que t_x não está em uma solução ótima. Seja então $S_k \subseteq T_k$ uma solução ótima para T_k e assuma que $t_x \notin S_k$.
- Seja $t_y \in S_k$ uma tarefa que termina primeiro em S_k. Podemos definir então o conjunto $S'_k = S_k - \{t_y\} \cup \{t_x\}$.
- Como S_k é solução viável, $f_y \leq s_z$ para toda $t_z \in S_k$.

Problema das Tarefas Compatíveis

- **Demonstração:** vamos supor que t_x não está em uma solução ótima. Seja então $S_k \subseteq T_k$ uma solução ótima para T_k e assuma que $t_x \notin S_k$.
- Seja $t_y \in S_k$ uma tarefa que termina primeiro em S_k. Podemos definir então o conjunto $S'_k = S_k - \{t_y\} \cup \{t_x\}$.
- Como S_k é solução viável, $f_y \leq s_z$ para toda $t_z \in S_k$.
- Então, S'_k é solução viável também.

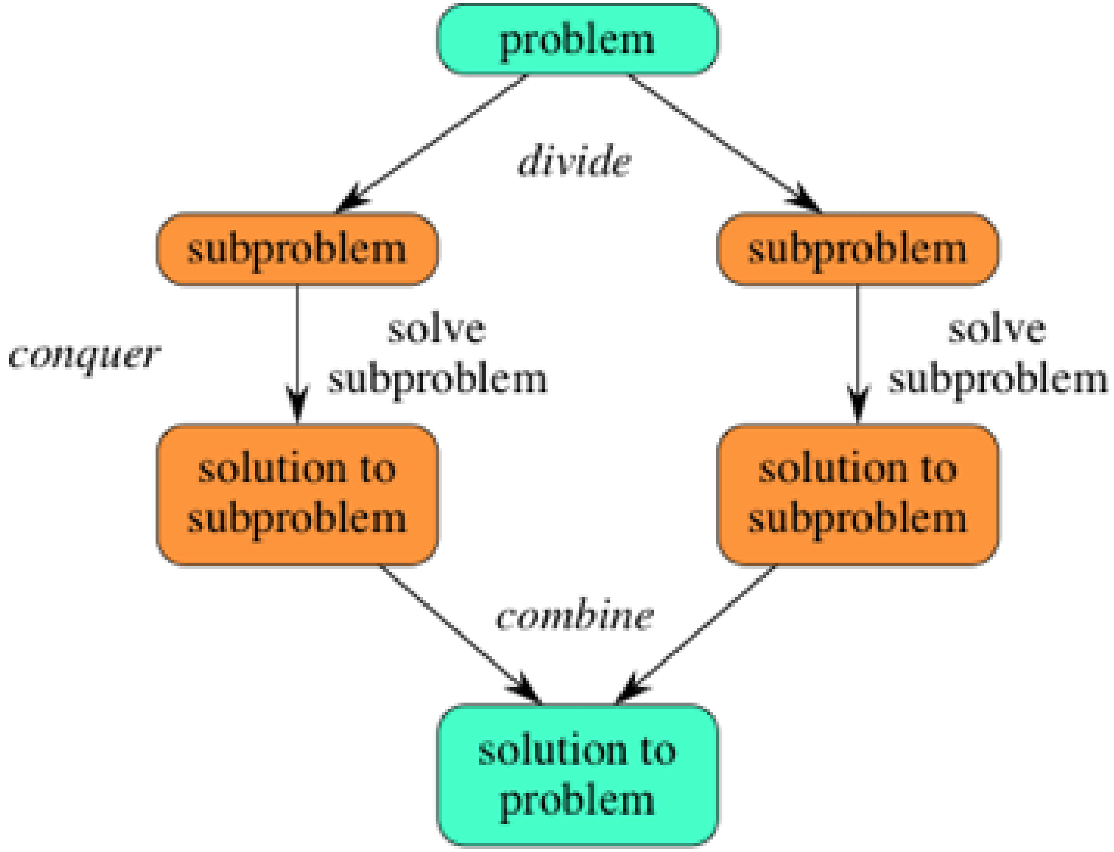
Problema das Tarefas Compatíveis

- **Demonstração:** vamos supor que t_x não está em uma solução ótima. Seja então $S_k \subseteq T_k$ uma solução ótima para T_k e assuma que $t_x \notin S_k$.
- Seja $t_y \in S_k$ uma tarefa que termina primeiro em S_k. Podemos definir então o conjunto $S'_k = S_k - \{t_y\} \cup \{t_x\}$.
- Como S_k é solução viável, $f_y \leq s_z$ para toda $t_z \in S_k$.
- Então, S'_k é solução viável também.
- Como $|S_k| = |S'_k|$, então S'_k é ótima também. Chegamos em uma contradição, pois havíamos suposto que t_x não fazia parte de uma solução ótima. ■

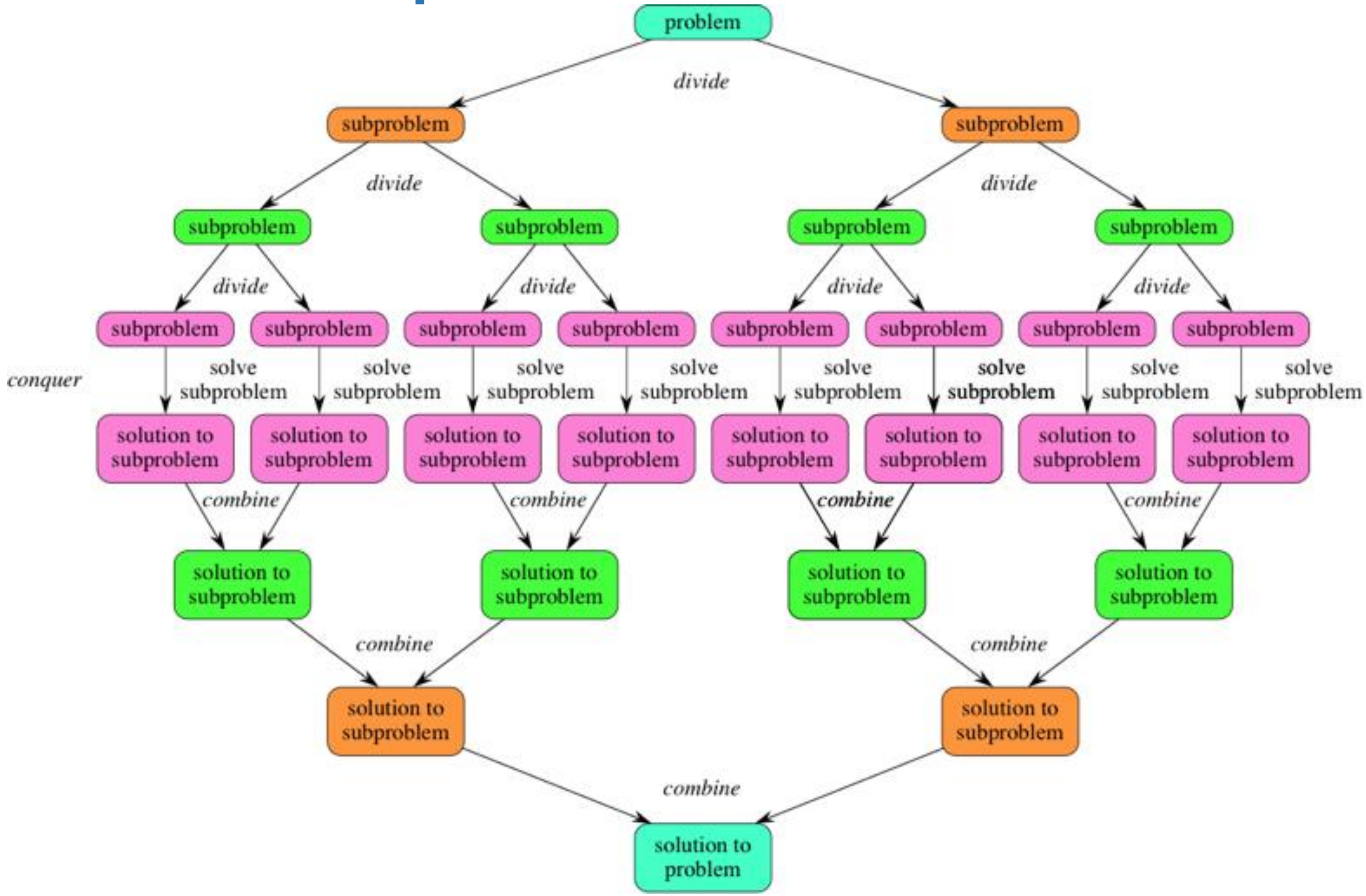
Divisão e Conquista

- Ideia geral:
  1. DIVIDIR: Dividir a instância do problema em duas ou mais instâncias menores;
  2. CONQUISTAR: Resolver as instâncias menores (geralmente recursivamente);
  3. COMBINAR: Obter a solução para as instâncias originais (maiores) através da combinação destas soluções
- Exemplos:
  - Mergesort
  - Quicksort
  - Busca binária

Divisão e Conquista



Divisão e Conquista



Divisão e Conquista

- Algoritmos baseados em divisão e conquista são, em geral, recursivos.
- A maioria dos algoritmos de divisão e conquista divide o problema em subproblemas da mesma natureza, de tamanho n/b.
- Existem três condições que indicam que a estratégia de divisão e conquista pode ser utilizada com sucesso:
  - Deve ser possível decompor uma instância em sub-instâncias
  - A combinação dos resultados deve ser eficiente (trivial se possível)
  - As sub-instâncias devem ser mais ou menos do mesmo tamanho

Divisão e Conquista

- Vantagens
  - Resolução de problemas difíceis (ex: Torre de Hanói)
  - Pode gerar algoritmos eficientes (forte tendência a complexidade logarítmica)
  - Facilmente paralelizável na fase da conquista (Em LPC isso não fará diferença)
- Desvantagens
  - Número de chamadas recursivas
  - Dificuldade na seleção dos casos bases
  - Repetição de sub-problemas (pode ser resolvido com Programação Dinâmica)

Divisão e Conquista - Exponenciação

- Exponenciação por força bruta em O(n):

```
int potencia(int x, int n) {
    int y = 1;
    for(int i = 0; i < n; i++)
        y *= x;
    return y;
}
```

Divisão e Conquista - Exponenciação

- Exponenciação com divisão e conquista em O(log n):

$$x^n = \begin{cases} 1, & n = 0 \\ x^{\frac{n}{2}} . x^{\frac{n}{2}}, & n \text{ é } par \\ x^{\frac{n}{2}} . x^{\frac{n}{2}} . x, & n \text{ é } ímpar \end{cases}$$

- Considerando que $\frac{n}{2}$ retorna o resultado da divisão inteira de n por 2.

Divisão e Conquista - Exponenciação

- Exponenciação com divisão e conquista em O(log n):

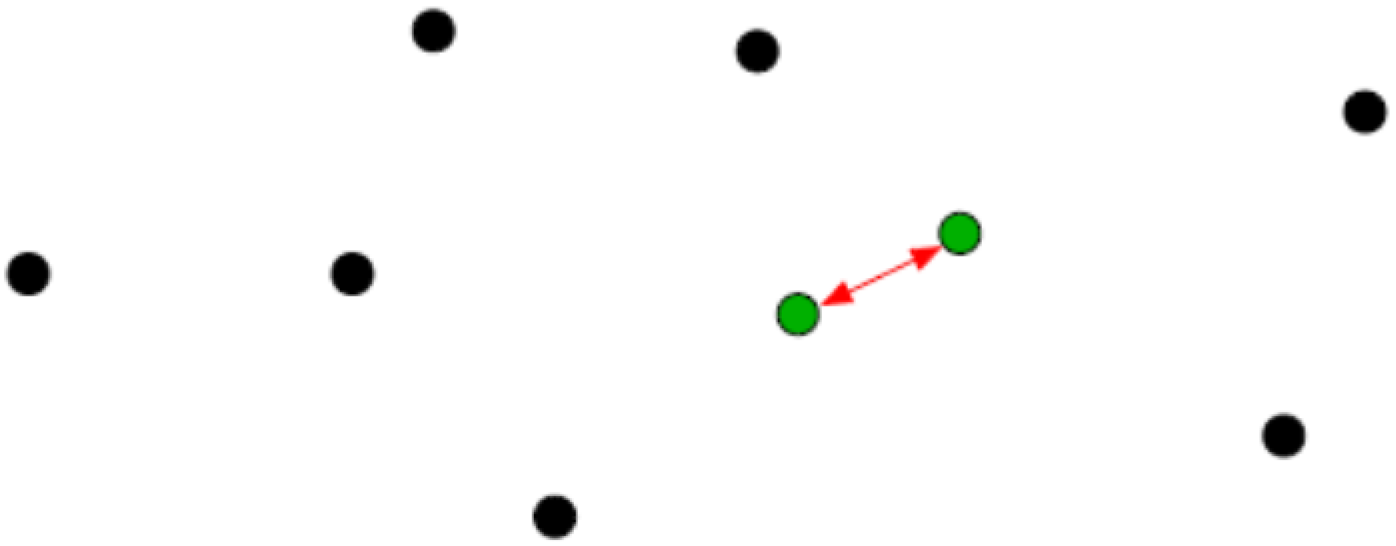
```
int potencia(int x, int n) {
    if (n == 0)
        return 1;
    int y = potencia(x, n/2);
    if (n % 2 == 0)
        return y*y;
    return y*y*x;
}
```

Departamento de Computação

Programação Competitiva unesp

Problema da Menor Distância

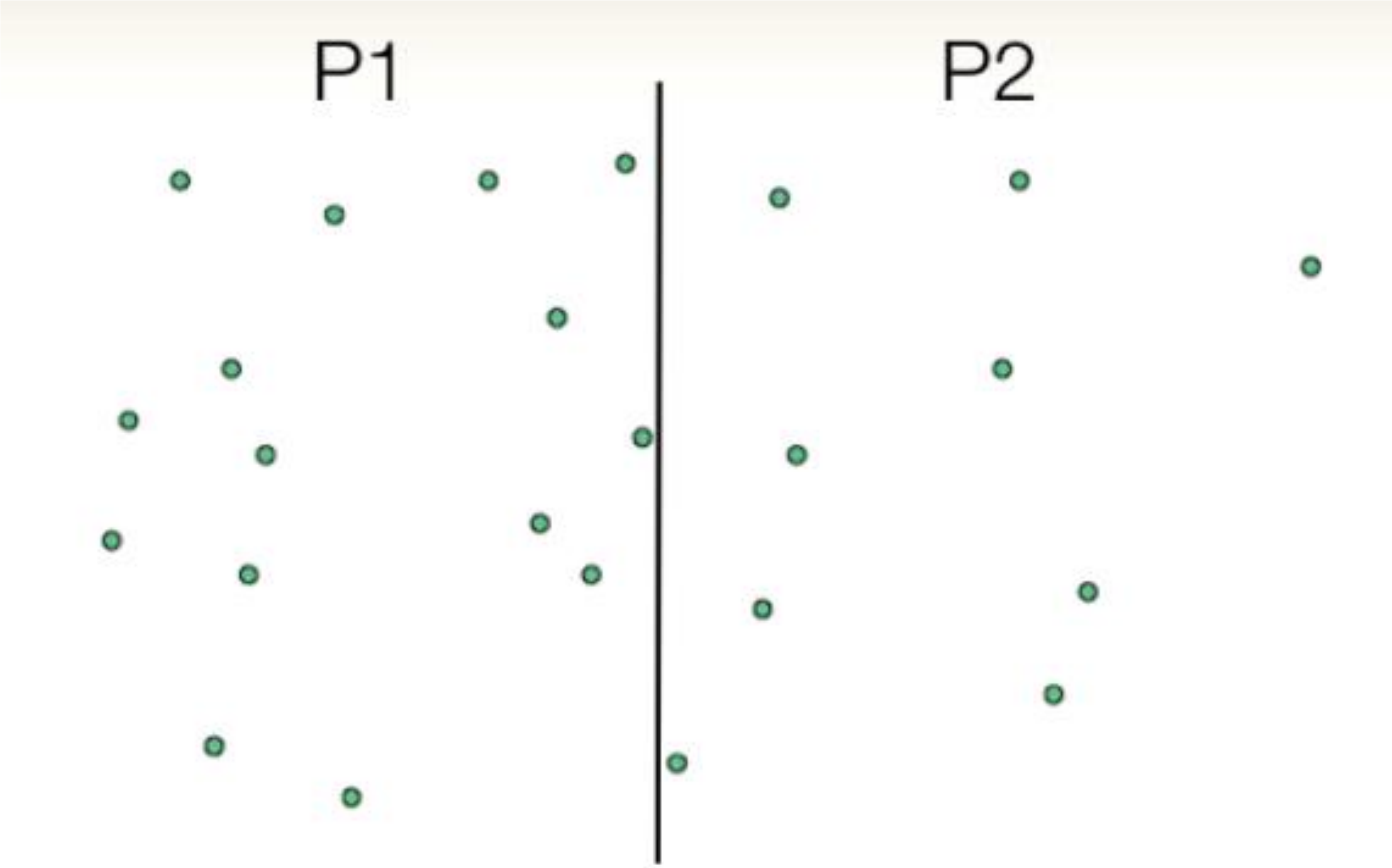
- Dados n pontos no plano, determinar a distância mínima entre qualquer par de pontos.



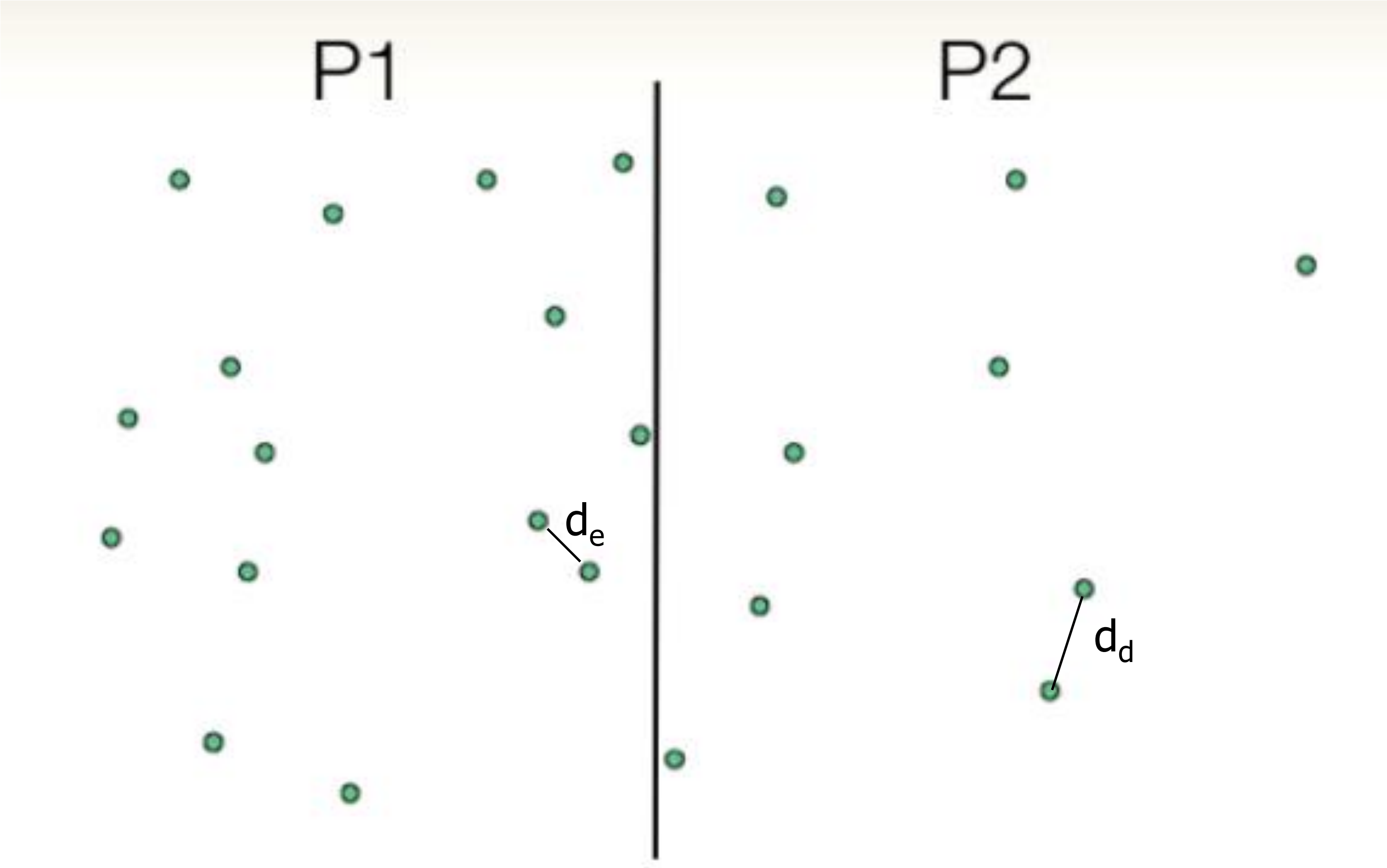
Problema da Menor Distância

- Solução por força bruta
  - Testar todos os possíveis pares de pontos. $O(n^2)$
- Como aplicar Divisão e Conquista?
  1. Vamos ordenar os pontos pela coordenada **x.**
  2. Dividir o problema em duas partes: esquerda e direita
  3. Resolver recursivamente os dois subproblemas gerados
  4. Combinar os subproblemas para obter a solução do problema inicial

Problema da Menor Distância



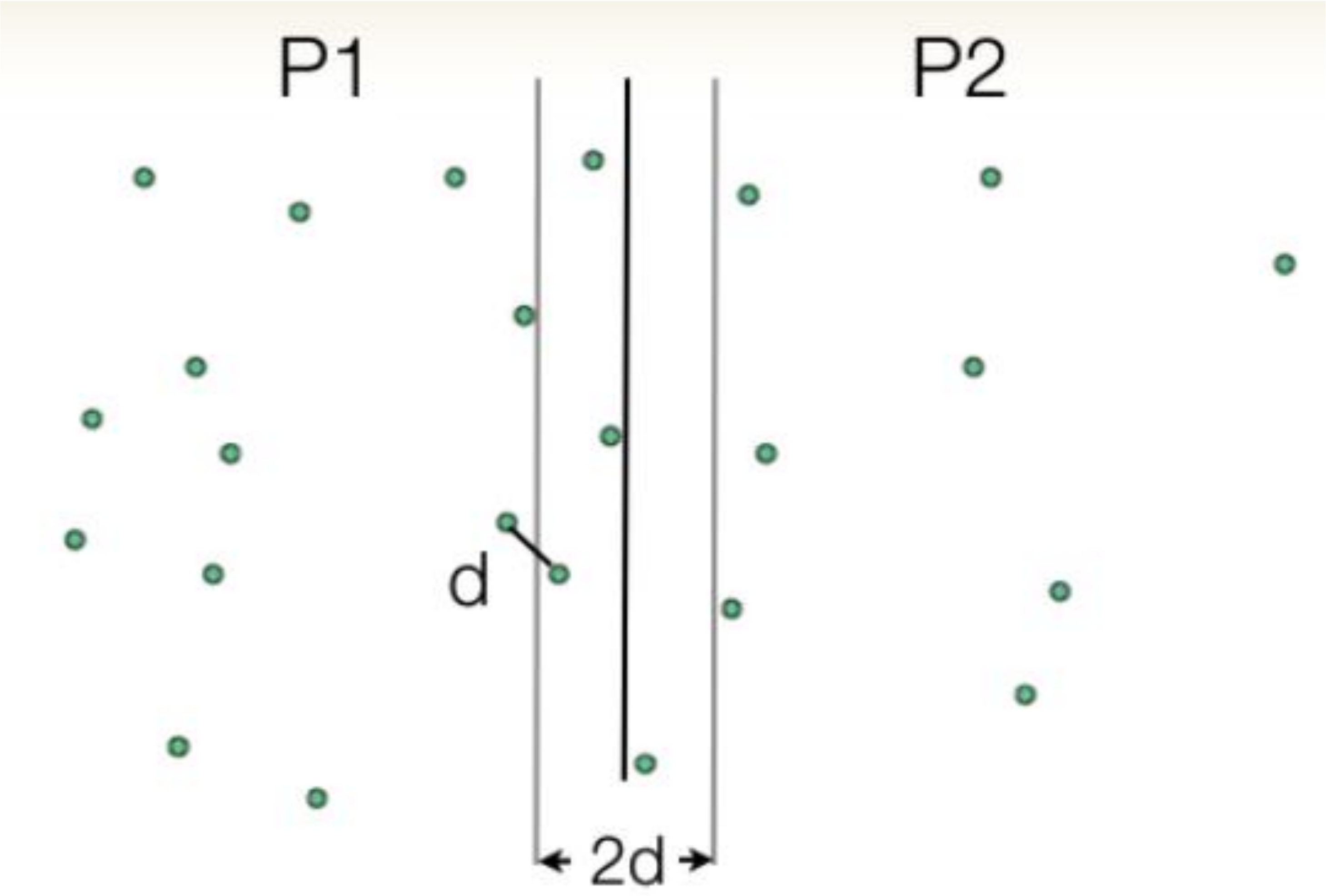
Problema da Menor Distância



Problema da Menor Distância

- Resolvendo os subproblemas P1 e P2 teremos a menor distância entre dois pontos nesses dois setores. Vamos chamar essas distâncias de d_e e d_d.
  - Com isso, podemos obter $d = \min(d_e, d_d)$
- Mas ainda falta analisar a distância entre pontos de sub-problemas distintos, ou seja, de pontos que estão no setor P1 com pontos que estão no setor P2.
- Devemos analisar TODOS os casos?
  - Não! Somente os pontos que se encontram em uma faixa 2d em torno da linha divisória, pontos além dessa linha não nos interessam, pois irão resultar em distâncias maiores que d.
- Complexidade: O(n.log n)

Problema da Menor Distância



Divisão e Conquista

- Outros problemas clássicos:
  - Multiplicação de Inteiros Grandes
  - Multiplicação de Matrizes (Algoritmo de Strassen)
- Sugestão: CodeForces 768B – Code For 1

Referências

Aulas de Técnicas de Programação do Prof. Dr. Renê Pegoraro.

LAAKSONEN, A. Competitive Programmer's Handbook.

Vídeo "Algoritmos gulosos e Problema das tarefas Compatíveis" da Profª. Drª. Carla Negri Lintzmayer. https://www.youtube.com/watch?v=PCMcGPknMwk

https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/guloso.html

https://www.geeksforgeeks.org/greedy-algorithms/

http://www3.decom.ufop.br/toffolo/site_media/uploads/2011-1/bcc402/slides/09._algoritmos_gulosos.pdf

https://docs.google.com/presentation/d/1rd4sxi2U6v3YNEJ0NRocFnVR64YBdJ2RsPffzjs9QFU/htmlpresent

Referências

https://pt.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms

http://www3.decom.ufop.br/toffolo/site_media/uploads/2011-1/bcc402/slides/08._divisao_e_conquista.pdf

http://www.dsc.ufcg.edu.br/~abrantes/CursosAnteriores/ATAL051/DivConq.pdf