SIN110 Algoritmos e Grafos

aula 06

Técnicas de Projeto de Algoritmos

- Projetos de algoritmos por indução (E02)
- Divisão e Conquista
- Programação Dinâmica

Projetos por Indução

E02 - solução

1) Considere o problema de *encontrar o maior e menor elemento de um conjunto **S*** que contem ***n*** números inteiros. Projete um algoritmo por indução que realiza até ***2n - 3*** comparações para resolver o problema.

```
Busca(S,n)
se n < 3
   então se S[n] > S[n-1]
             então M ← S[n]
                   m ← S[n-1]
             senão M ← S[n-1]
                   m ← S[n]
  senão (M, m) ← Busca(S,n-1)
        se S[n] > M
           então M ← S[n]
           senão se S[n] < m
                    então m ← S[n]
 devolve (M, m)
```

```
Recorrência de comparações
n > 2 :   T(n)= T(n-1)+ 2
n = 2 :   T(2)= 1

resolvendo:  T(n) = T(n-1)+2 = T(n-2)+2+2 = ... = T(2)+2+2+...+2
             T(n) = 2(n-2) + 1 = 2n - 4 + 1= 2n -3
```

2) Dados um vetor ordenado $\boldsymbol{A}$ de $\boldsymbol{n}$ números reais, $\boldsymbol{n \geq 1}$, e um número real $\boldsymbol{x}$, queremos determinar se existem $\boldsymbol{A[i]}$ e $\boldsymbol{A[j]}$, $\boldsymbol{1 \leq i, j \leq n}$, tais que $\boldsymbol{x = A[i] + A[j]}$. Projete um algoritmo por indução de complexidade $\boldsymbol{O(n)}$ para esse algoritmo. Escreva a relação de recorrência $\boldsymbol{T(n)}$ para seu algoritmo e prove que o resultado da recorrência é de fato $\boldsymbol{O(n)}$.

```
{   entrada: vetor A[1..n], e=1, d=n, x      }
{  pesquisa: x = A[i] + A[j] ?               }
{     saída: devolve (i,j) ou (-1,-1)        }

Pesquisa(A,e,d,x)
    se e > d
       então devolve (-1,-1)
       senão se x = A[e]+ A[d]
                 então devolve (e,d)
                 senão se x < A[e]+ A[d]
                          então e ← e+1
                          senão d ← d-1
                       devolve Pesquisa(A,e,d,x)
```

Recorrência

$n \geq 1$: $T(n) = T(n-1) + 5$

$n < 1$: $T(0) = 2$

Resolvendo:

$$T(n) = T(n-1)+5 = T(n-2)+5+5 = \ldots = T(0)+5+5+\ldots+5 = 2 + 5n$$

$$T(n) = O(n)$$

Projeto por Indução : o problema da celebridade

Definição

Num conjunto S de n pessoas, uma *celebridade* é alguém que é conhecido por todas as pessoas de S mas que não conhece ninguém. (Celebridades são pessoas de difícil convívio...).

Note que pode existir no máximo uma celebridade em S!

Problema:

Determinar se existe uma celebridade em um conjunto S de n pessoas.

o problema da celebridade

Vamos formalizar melhor: para um conjunto de n pessoas, associamos uma matriz $n \times n$ M tal que $M[i,j] = 1$ se a pessoa i conhece a pessoa j e $M[i,j] = 0$ caso contrário.
Por convenção, $M[i,i] = 0$ para todo i.

Problema:

Dado um conjunto de n pessoas e a matriz associada M encontrar (se existir) uma celebridade no conjunto.
Isto é, determinar um k tal que todos os elementos da coluna k (exceto $M[k,k]$) são 1s e todos os elementos da linha k são 0s.

Existe uma solução simples mas laboriosa: para cada pessoa i, verifique todos os outros elementos da linha i e da coluna i.
O custo dessa solução é $2(n-1)n$.

O problema da celebridade

A segunda tentativa baseia-se em um fato muito simples:

> Dadas duas pessoas i e j, é possível determinar se uma delas **não** é uma celebridade com apenas uma comparação: se $M[i,j] = 1$, então i não é celebridade; caso contrário j não é celebridade.

Vamos usar esse argumento aplicando a hipótese de indução sobre o conjunto de $n-1$ pessoas obtidas removendo de S uma pessoa que sabemos não ser celebridade.

- O caso base e a hipótese de indução são os mesmos que anteriormente.

O problema da celebridade

Tome então um conjunto arbitrário de $n > 2$ pessoas e a matriz M associada.

Sejam i e j quaisquer duas pessoas e suponha que j não é celebridade (usando o argumento acima).

Seja $S' = S \setminus \{j\}$ e considere os dois casos possíveis:

1. Existe uma celebridade em S', digamos a pessoa k. Se $M[j,k] = 1$ e $M[k,j] = 0$, então k é celebridade em S; caso contrário não há uma celebridade em S.
2. Não existe celebridade em S'; então não existe uma celebridade em S.

O problema da celebridade

Celebridade(S, M)

$\triangleright$ **Entrada:** conjunto de pessoas $S = \{1, 2, \ldots, n\}$;
M, a matriz que define quem conhece quem em S.

$\triangleright$ **Saída:** Um inteiro $k \leq n$ que é celebridade em S ou $k = 0$

1. **se** $|S| = 1$ **então** $k \leftarrow$ elemento em S
2. **senão**
3. sejam i, j quaisquer duas pessoas em S
4. **se** $M[i, j] = 1$ **então** $s \leftarrow i$ **senão** $s \leftarrow j$
5. $S' \leftarrow S \setminus \{s\}$
6. $k \leftarrow$ Celebridade(S', M)
7. **se** $k > 0$ **então**
8. **se** $(M[s, k] \neq 1)$ **ou** $(M[k, s] \neq 0)$ **então** $k \leftarrow 0$
9. **devolva** k

O problema da celebridade

A recorrência $T(n)$ para o número de operações executadas pelo algoritmo é:

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ T(n-1) + \Theta(1), & n > 1. \end{cases}$$

A solução desta recorrência é

$$\sum_{1}^{n} \Theta(1) = n\Theta(1) = \Theta(n).$$

3) Num conjunto ***S*** de ***n*** pessoas, uma celebridade é alguém que é conhecido por todas as pessoas de ***S***, mas que não conhece ninguém. Isso implica que pode existir somente uma celebridade em ***S***. (Celebridades são pessoas de difícil convívio...). Problema: projetar por indução, um algoritmo linear em ***n*** que determine se existe uma celebridade em ***S***.

```
Celebridade (M[e..d, e..d])
se d-e = 0
   então k ← d
   senão  se M[e , d] = 1
                 então s ← e;  k ← Celebridade(M[e+1 .. d, e+1..d])
                 senão s ← d;  k ← Celebridade(M[e .. d-1, e .. d-1])
          se k > 0
                 então se M[s, k] ≠ 1 OU M[k, s] ≠ 0
                             então k ← 0
devolve k
```

simulando

Cel(M[1..4, 1..4])

0	1	1	1	S=1, k=4
0	0	1	1	dev k=3
0	0	0	0	
0	0	1	0	

Cel(M[2..4, 2..4])

0	1	1	1	
0	0	1	1	S=2, k=3
0	0	0	0	dev k=3
0	0	1	0	

Cel(M[3..4, 3..4])

0	1	1	1	
0	0	1	1	
0	0	0	0	s=4, k=3
0	0	1	0	dev k=3

Cel(M[3..3, 3..3])

0	1	1	1	
0	0	1	1	
0	0	0	0	
0	0	1	0	dev k=3

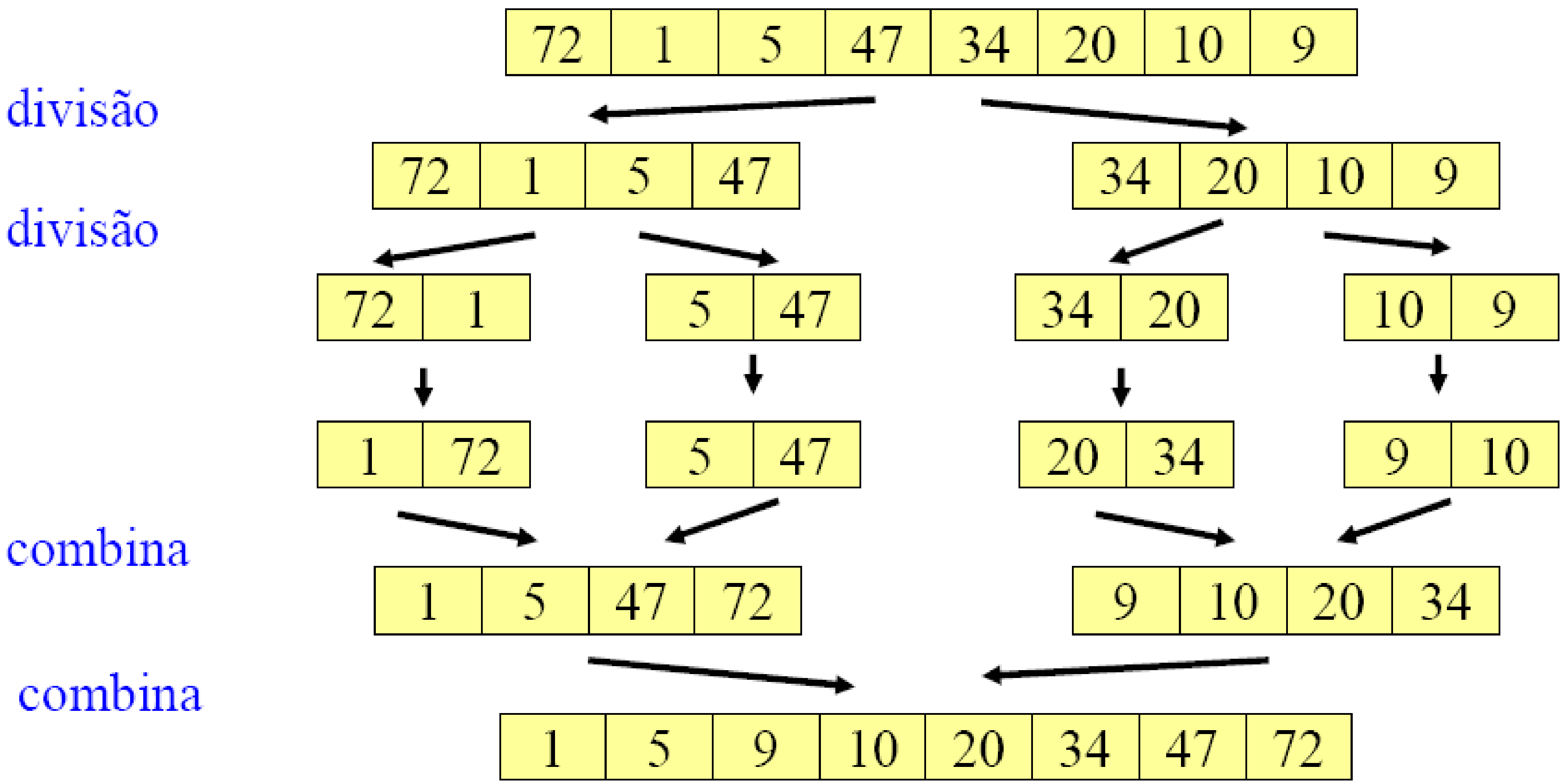
Divisão e Conquista

Divisão e Conquista

Motivação:

- Tomar um problema de “entrada grande”:
- Quebrar a entrada em pedaços menores (DIVISÃO)
- Resolver cada pedaço separadamente. (CONQUISTA)
- Combinar os resultados.

MergeSort



72 1 5 47 34 20 10 9
divisão
72 1 5 47
34 20 10 9
divisão
72 1
5 47
34 20
10 9
1 72
5 47
20 34
9 10
combina
1 5 47 72
9 10 20 34
combina
1 5 9 10 20 34 47 72

Recorrências de Divisão e Conquista

Todo algoritmo eficiente de Divisão e Conquista divide os problemas em subproblemas, onde cada um é uma fração do problema original, e então realiza algum trabalho adicional para computar a resposta final, resultando na expressão geral de recorrência:

$$f(n) = af(n/b) + g(n).$$

Recorrências de Divisão e Conquista

O teorema Mestre pode ser usado para determinar esse tempo para a maioria dos algoritmos de divisão e conquista.

Teorema: a solução para a equação $f(n) = af(n/b) + O(n^k)$, *onde* $a \geq 1$ e $b > 1$ *, é*

$$f(n) = \begin{cases} O(n^{\log_b a}) \text{ se } a > b^k \\ O(n^k \log n) \text{ se } a = b^k \\ O(n^k) \text{ se } a < b^k \end{cases}$$

Recorrências de Divisão e Conquista

Existem recorrências onde não podemos aplicar o Teorema Mestre, alguns exemplos:

i) $T(1) = 1$

$T(n) = T(n-1) + n$

ii) $T(b) = 1$

$T(n) = T(n-a) + T(a) + n$ (para $a \geq 1$, $b \leq a$, a e b inteiros)

iii) $T(1) = 1$

$T(n) = T(\alpha n) + T((1-\alpha)n) + n$ (para $0 < \alpha < 1$)

iv) $T(1) = 1$

$T(n) = T(n-1) + \lg n$

v) $T(1) = 1$

$T(n) = 2T(n/2) + n\lg n$

Máximo e Mínimo

Problema: *dada uma lista A com n elementos, determinar o maior e o menor elemento da lista.*

Um algoritmo incremental, projetado por indução, para esse problema faz $2n-3$ comparações: fazemos uma comparação no caso base e duas no passo... (confira!)

Usando a estratégia de divisão e conquista podemos melhorar um pouco o desempenho:

- Divida a lista em dois subconjuntos de mesmo tamanho, respectivamente A_1 com $\lfloor n/2 \rfloor$ elementos e A_2 com $\lceil n/2 \rceil$ elementos e, solucione os subproblemas;
- O máximo da lista A é o máximo dos máximos de A_1 e A_2 e o mínimo de A é o mínimo de A_1 e A_2.

Máximo e Mínimo

Algoritmo:

```
MaxMin(A, e, d)
  se d - e <= 1
      então se A(e) > A(d)
                  então max ← A(e)
                        min  ← A(d)
                  senão max ← A(d)
                        min ← A(e)
      senão m ← ⌊(d+e)/2⌋
            (max1, min1) ← MaxMin(A, e, m)
            (max2, min2) ← MaxMin(A, m+1, d)
            se max1 > max2
               então max ← max1
               senão max ← max2
            se min1 < min2
               então min ← min1
               senão min ← min2
  devolve (max, min)
```

Máximo e Mínimo

Na análise desse algoritmo vamos considerar o número de comparações efetuados para determinar *max* e *min*, resultando a recorrência:

$$T(n) = 1 \qquad \text{se} \quad n \leq 2$$

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2 \qquad \text{se} \quad n > 2$$

Note que não podemos empregar o teorema Mestre.

Vamos supor que o número de elementos n é uma potência de 2, isto é $n = 2^k$ para k inteiro, e trabalhar a relação:

$$T(n) = 2T(n/2) + 2 = 2[2T(n/4) + 2] + 2 =$$
$$= 2[2[2T(n/8) + 2] + 2] + 2 = \ldots = 2^k + [2^{k-1} - 2]$$

Obtendo a solução: $T(n) = 3n/2 - 2$

Assintóticamente, os dois algoritmos para esse problema são equivalentes, ambos apresentam solução: $T(n) = \Theta(n)$.

Entretanto, a divisão e conquista permite que menos comparações sejam feitas.

Par de pontos com a menor distância (closest pair)

- *Closest pair*: dados n pontos no plano, encontrar um par com a menor distância Euclidiana entre os dois.

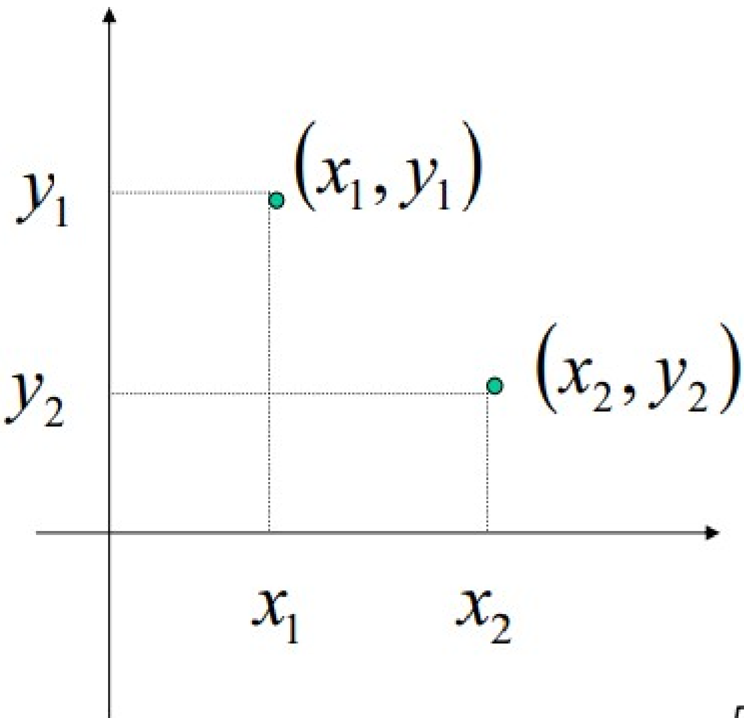
Closest Pair

Entrada: Um conjunto de pontos n $\mathcal{P} = \langle p_1, p_2, \ldots, p_n \rangle$, em duas dimensões.

Saída: O par de pontos p_1 *e* p_2 que apresenta a menor distância euclideana.

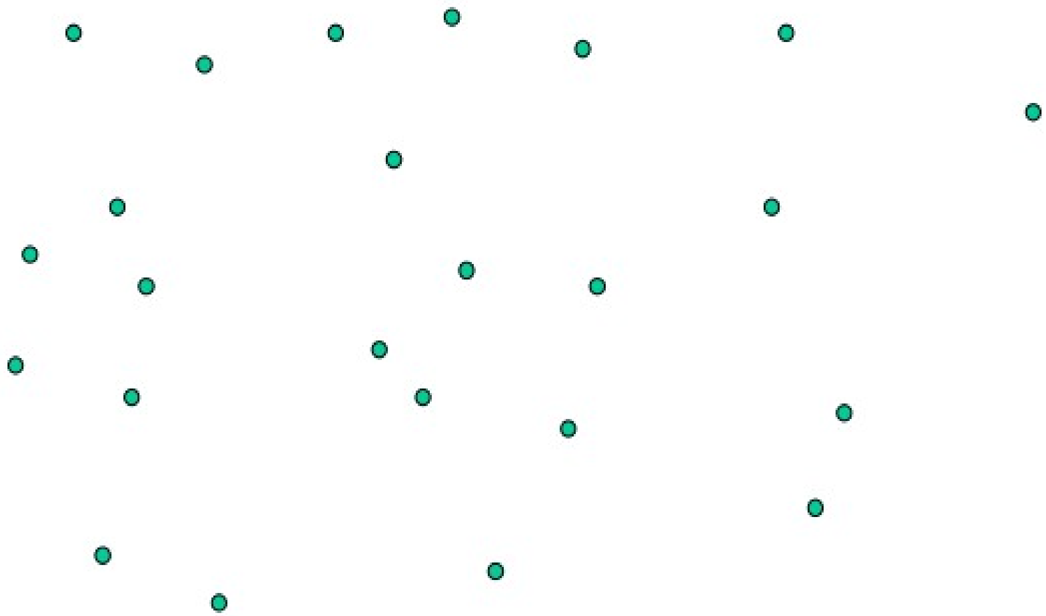
Menor Distância Entre Pontos

- Distância Euclideana

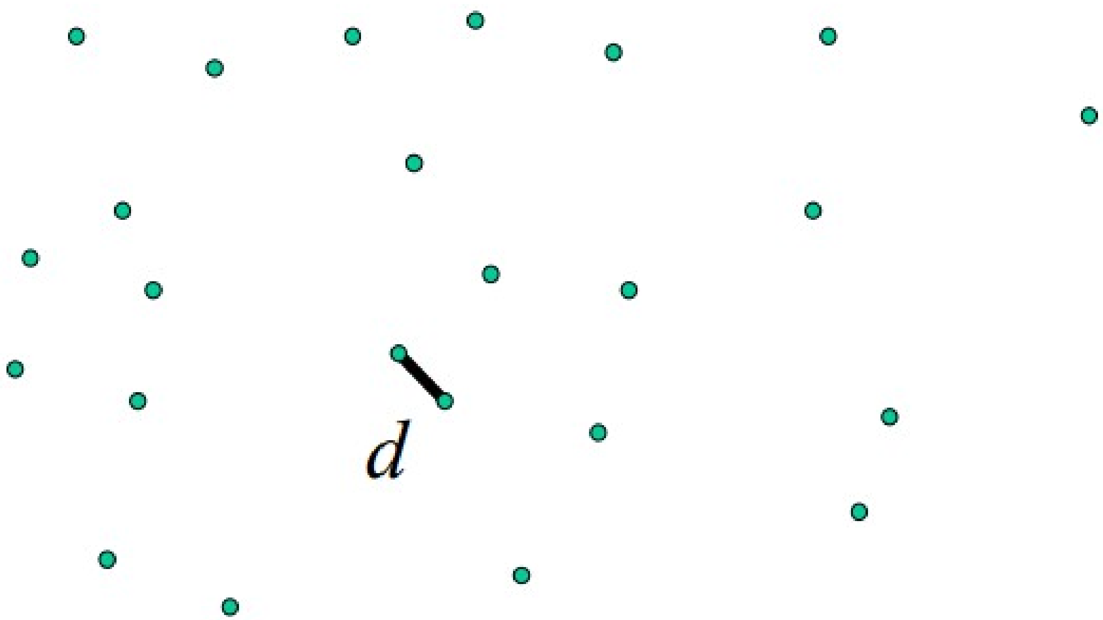


$$\left\|(x_1, y_1) - (x_2, y_2)\right\| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Menor Distância Entre Pontos



Menor Distância Entre Pontos

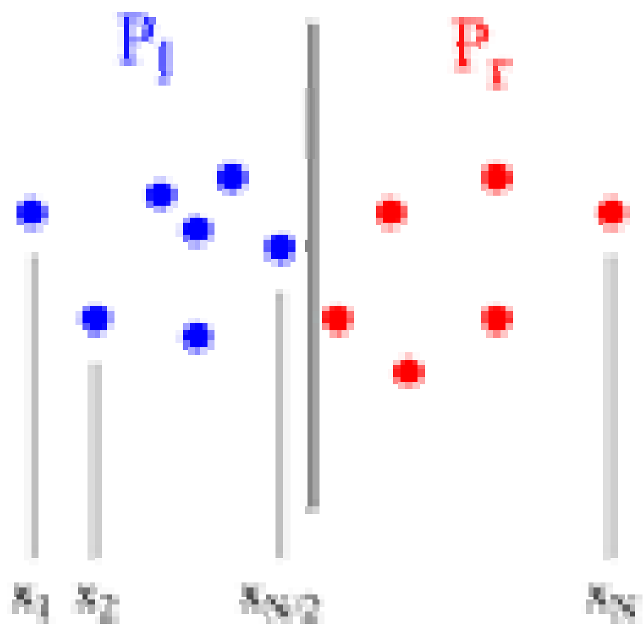


Menor Distância Entre Pontos

- Solução Força Bruta é $O(n^2)$.
- Vamos assumir:
  - Não existem pontos com a mesma coordenada x.
  - Não existem pontos com a mesma coordenada y.
- Como resolver este problema considerando 1D?
- É possível aplicar Divisão e Conquista?

Menor Distância Entre Pontos – 2D

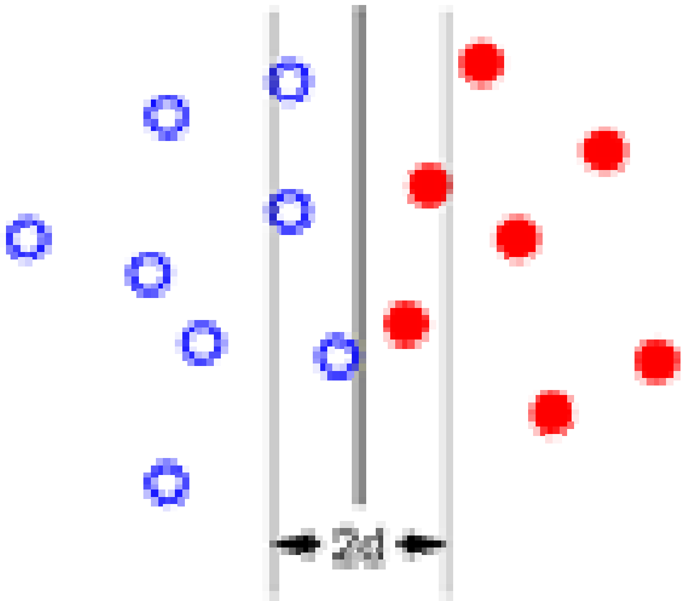
- Como dividir em sub-problemas?
  - Ordenar de acordo com a coordenada x e dividir em duas partes: esquerda e direita.



Menor Distância Entre Pontos – 2D

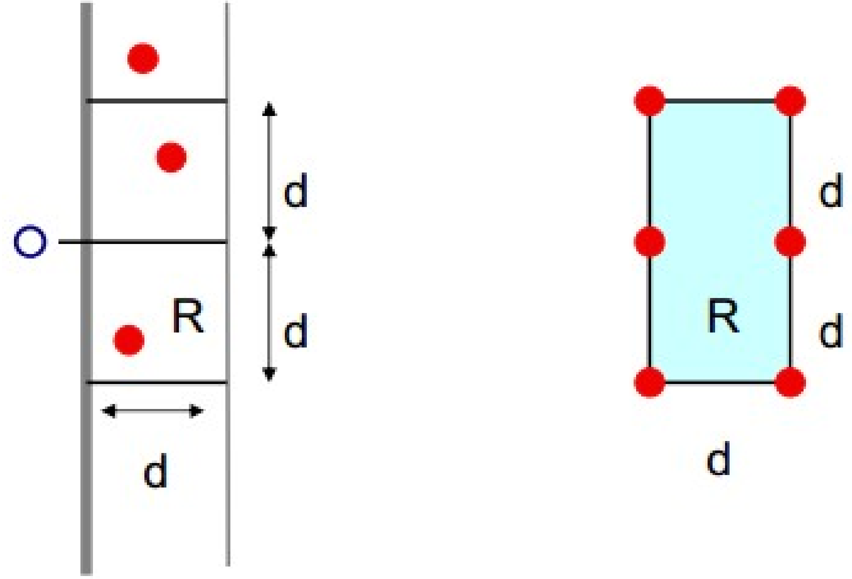
- Resolver recursivamente cada sub-problema, obtendo d_l e d_r.
- O que podemos observar?
  - Já temos a menor distância em cada uma das partes.
  - Fazer d= min{d_l , d_r}.
  - Falta analisar distância entre pontos de sub-problemas distintos.
  - Devemos analisar todos os casos?
    - Somente pontos que se encontram em uma faixa de tamanho 2d em torno da linha divisória.

Menor Distância Entre Pontos – 2D



Menor Distância Entre Pontos – 2D

- Qual a quantidade de pontos que se encontram dentro da faixa de tamanho 2d?
  - Se considerarmos um **p** $\in$ P_l, todos os pontos de P_r que devem ser considerados devem estar em um retângulo **R** de dimensões **d** × **2d**.



Menor Distância Entre Pontos – 2D

- Como determinar os seis pontos?
  - Projeção de pontos nos eixos ***x*** e ***y***.
  - Pode-se fazer isso para todo **p** $\in P_l$ **e** P_r, em O(n) (pontos ordenados).
- Relação de recorrência é **T(n)= 2.T(n/2) + O(n)**
  - Sabemos que isso é **O(n.log n)**

ClosestPair(P)

Pré-processamento
Construir P_x e P_y como listas ordenadas pelas coordenadas **x** e **y**

Divisão
Quebrar P em P_l e P_r

Conquista
d_l = ClosestPair(P_l)
d_r = ClossetPair(P_r)

Combinação
d= min{d_l, d_r}
Determinar faixa divisória e pontos
Verificar se tem algum par com distância < d

Programação Dinâmica

Programação Dinâmica

- O algoritmo PD resolve cada subproblema uma vez só e então grava sua resposta em uma tabela, evitando assim o trabalho de recalcular a resposta toda vez que o subproblema é encontrado.
- Nesse contexto: “programação” refere-se a uma tabulação e não à codificação de um algoritmo.
- Aplicada a problemas de otimização com muitas soluções possíveis: cada solução tem um valor, e desejamos encontrar uma solução com um valor ótimo (mínimo ou máximo).

Números de Fibonacci

$$F_0 = 0 \qquad F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

n	0	1	2	3	4	5	6	7	8	9
F_n	0	1	1	2	3	5	8	13	21	34

Algoritmo recursivo para F_n:

FIBO-REC (n)
    se $n \leq 1$
        então devolva n
        senão $a \leftarrow$ FIBO-REC $(n-1)$
                $b \leftarrow$ FIBO-REC $(n-2)$
                devolva $a + b$

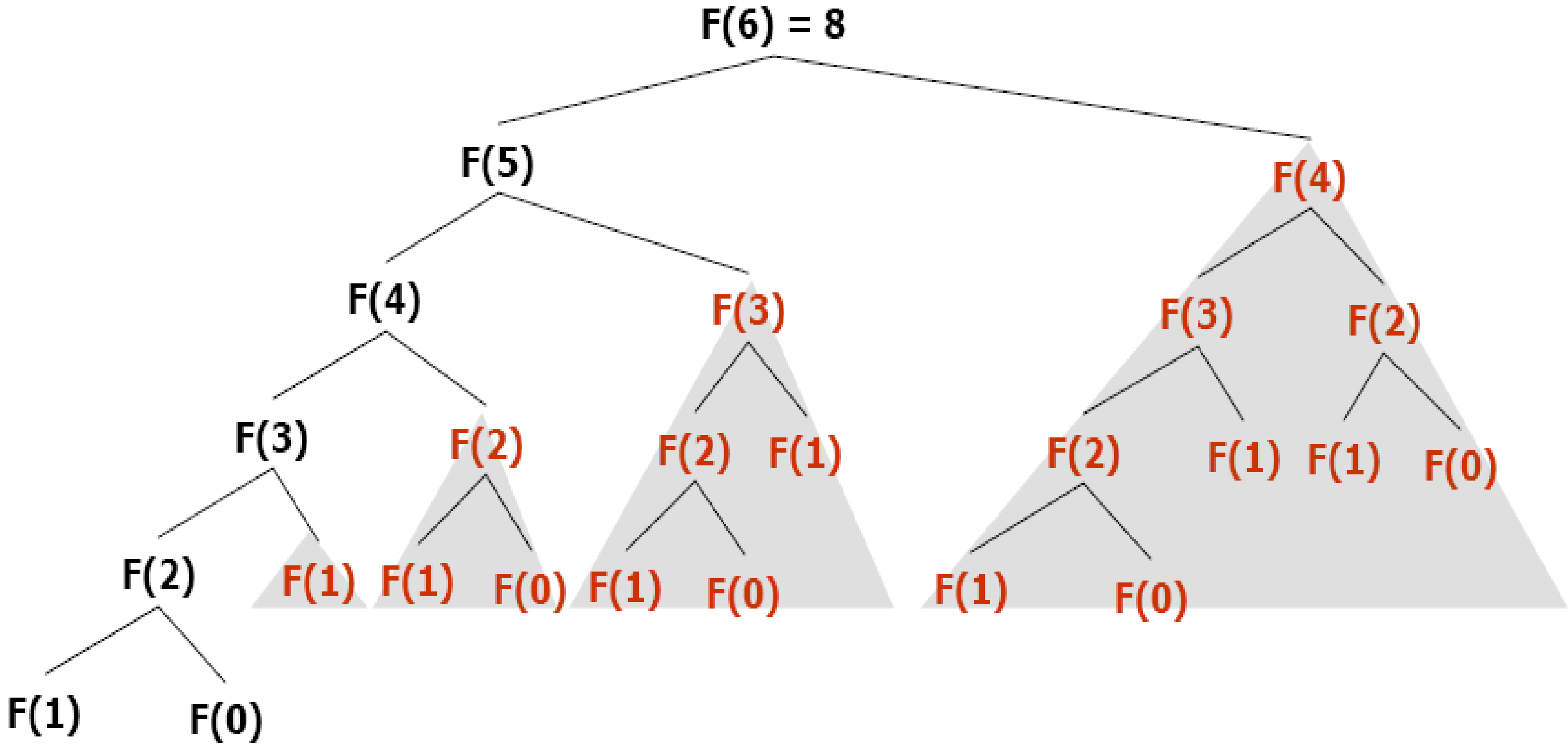
Consumo de tempo

$T(n) :=$ número de somas feitas por FIBO-REC (n)

linha	número de somas
1-2	$= 0$
3	$= T(n-1)$
4	$= T(n-2)$
5	$= 1$

$$T(n) \quad = T(n-1) + T(n-2) + 1$$

Números de Fibonacci



Recorrência

$$T(0) = 0$$

$$T(1) = 0$$

$$T(n) = T(n-1) + T(n-2) + 1 \quad \text{para } n = 2, 3, \ldots$$

A que classe Ω pertence $T(n)$?
A que classe O pertence $T(n)$?

Recorrência

$$T(0) = 0$$

$$T(1) = 0$$

$$T(n) = T(n-1) + T(n-2) + 1 \quad \text{para } n = 2, 3, \ldots$$

A que classe Ω pertence $T(n)$?
A que classe O pertence $T(n)$?

Solução: $T(n) > (3/2)^n$ para $n \geq 6$.

n	0	1	2	3	4	5	6	7	8
T_n	0	0	1	2	4	7	12	20	33
$(3/2)^n$	1	1.5	2.25	3.38	5.06	7.59	11.39	17.09	25.63

Recorrência

Prova: $T(6) = 12 > 11.40 > (3/2)^6$ e $T(7) = 20 > 18 > (3/2)^7$.
Se $n \geq 8$, então

$$\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 \\
&\overset{\text{hi}}{>} (3/2)^{n-1} + (3/2)^{n-2} + 1 \\
&= (3/2+1)\,(3/2)^{n-2} + 1 \\
&> (5/2)\,(3/2)^{n-2} \\
&> (9/4)\,(3/2)^{n-2} \\
&= (3/2)^2 (3/2)^{n-2} \\
&= (3/2)^n\,.
\end{aligned}$$

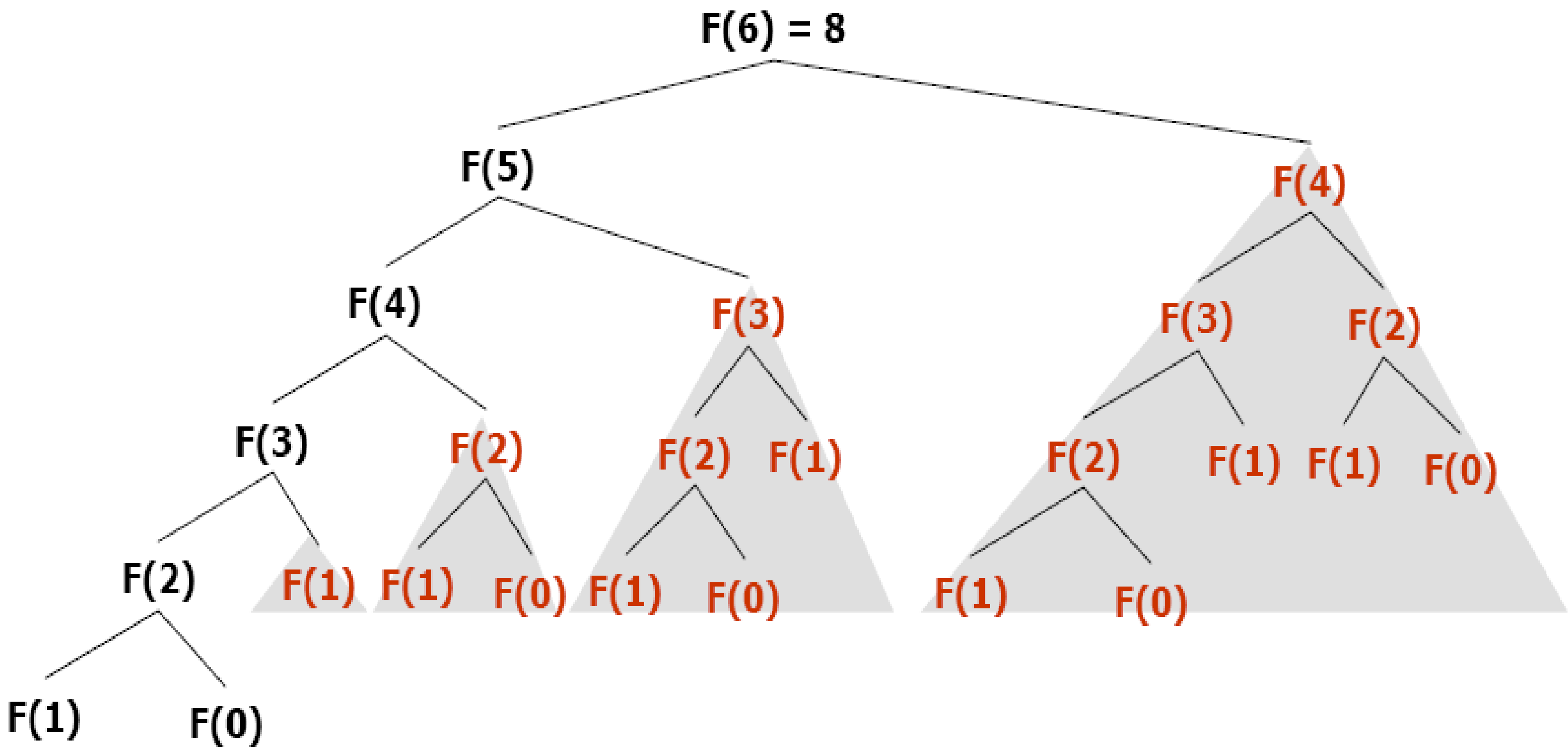
Logo, $T(n)$ é $\Omega((3/2)^n)$.

Verifique que $T(n)$ é $\mathrm{O}(2^n)$.

Consumo de tempo

Consumo de tempo é exponencial.

Algoritmo resolve subproblemas muitas vezes.



Resolve subproblemas muitas vezes

```
FIBO-REC(5)
  FIBO-REC(4)
    FIBO-REC(3)
      FIBO-REC(2)
        FIBO-REC(1)
        FIBO-REC(0)
      FIBO-REC(1)
    FIBO-REC(2)
      FIBO-REC(1)
      FIBO-REC(0)
  FIBO-REC(3)
    FIBO-REC(2)
      FIBO-REC(1)
      FIBO-REC(0)
    FIBO-REC(1)
```

FIBO-REC(5) = 5

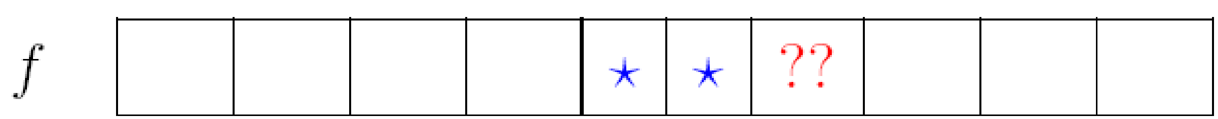
Resolve subproblemas muitas vezes

```
FIBO-REC(8)
  FIBO-REC(7)
    FIBO-REC(6)
      FIBO-REC(5)
        FIBO-REC(4)
          FIBO-REC(3)
            FIBO-REC(2)
              FIBO-REC(1)
              FIBO-REC(0)
            FIBO-REC(1)
          FIBO-REC(2)
            FIBO-REC(1)
            FIBO-REC(0)
        FIBO-REC(3)
          FIBO-REC(2)
            FIBO-REC(1)
            FIBO-REC(0)
          FIBO-REC(1)
      FIBO-REC(4)
        FIBO-REC(3)
          FIBO-REC(2)
            FIBO-REC(1)
            FIBO-REC(0)
          FIBO-REC(1)
        FIBO-REC(2)
          FIBO-REC(1)
          FIBO-REC(0)
    FIBO-REC(5)
      FIBO-REC(4)
        FIBO-REC(3)
          FIBO-REC(2)
            FIBO-REC(1)
            FIBO-REC(0)
          FIBO-REC(1)
        FIBO-REC(2)
          FIBO-REC(1)
          FIBO-REC(0)
      FIBO-REC(3)
        FIBO-REC(2)
          FIBO-REC(1)
          FIBO-REC(0)
        FIBO-REC(1)
  FIBO-REC(6)
    FIBO-REC(5)
      FIBO-REC(4)
        FIBO-REC(3)
          FIBO-REC(2)
            FIBO-REC(1)
            FIBO-REC(0)
          FIBO-REC(1)
        FIBO-REC(2)
          FIBO-REC(1)
          FIBO-REC(0)
      FIBO-REC(3)
        FIBO-REC(2)
          FIBO-REC(1)
          FIBO-REC(0)
        FIBO-REC(1)
    FIBO-REC(4)
      FIBO-REC(3)
        FIBO-REC(2)
          FIBO-REC(1)
          FIBO-REC(0)
        FIBO-REC(1)
      FIBO-REC(2)
        FIBO-REC(1)
        FIBO-REC(0)
```

Algoritmo de programação dinâmica

```
FIBO (n)
    f[0] ← 0
    f[1] ← 1
    para i ← 2 até n faça
        f[i] ← f[i − 1] + f[i − 2]
    devolva f[n]
```

Note a tabela $f[0\,..\,n{-}1]$.



Consumo de tempo é $\Theta(n)$.

Algoritmo de programação dinâmica

Versão com economia de espaço.

```
FIBO (n)
    se n = 0 então devolva 0
    f_ant ← 0
    f_atual ← 1
    para i ← 2 até n faça
        f_prox ← f_atual + f_ant
        f_ant ← f_atual
        f_atual ← f_prox
    devolva f_atual
```

Versão recursiva eficiente

MEMOIZED-FIBO (f, n)

para $i \leftarrow 0$ **até** n **faça**
    $f[i] \leftarrow -1$
devolva LOOKUP-FIBO (f, n)

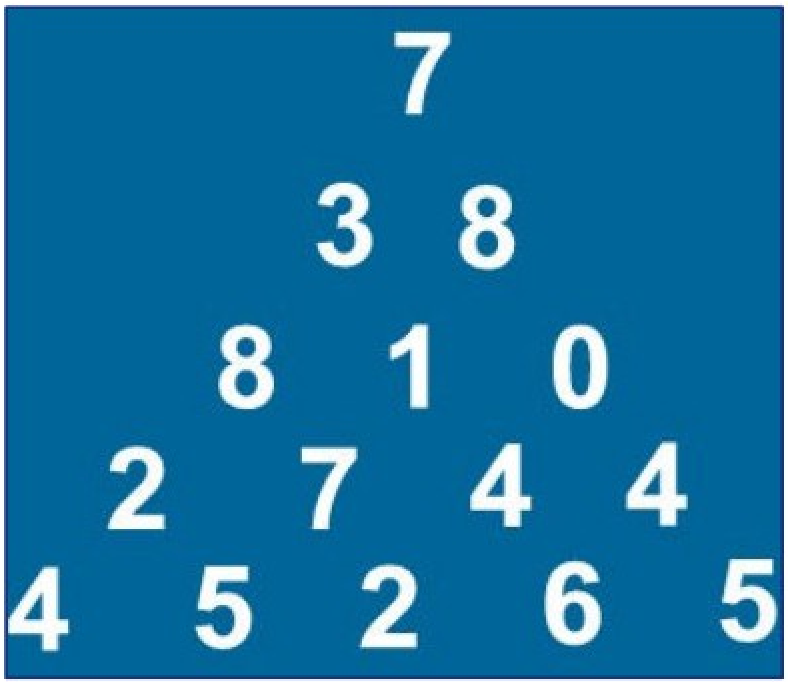
LOOKUP-FIBO (f, n)

se $f[n] \geq 0$
    então devolva $f[n]$
se $n \leq 1$
    então $f[n] \leftarrow n$
    senão $f[n] \leftarrow$ LOOKUP-FIBO$(f, n-1)$
        $+$ LOOKUP-FIBO$(f, n-2)$
devolva $f[n]$

Não recalcula valores de f.

Pirâmide de números

- Problema clássico das **Olimpíadas Internacionais de Informática de 1994**

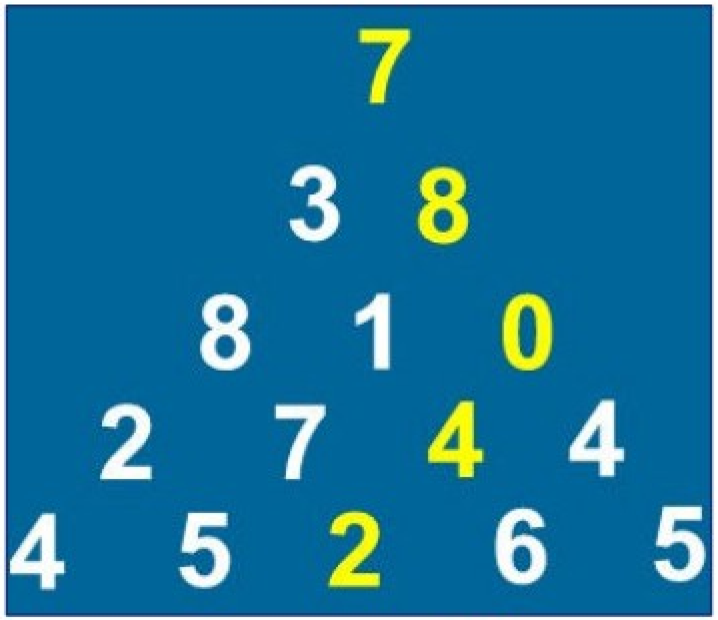


Problema

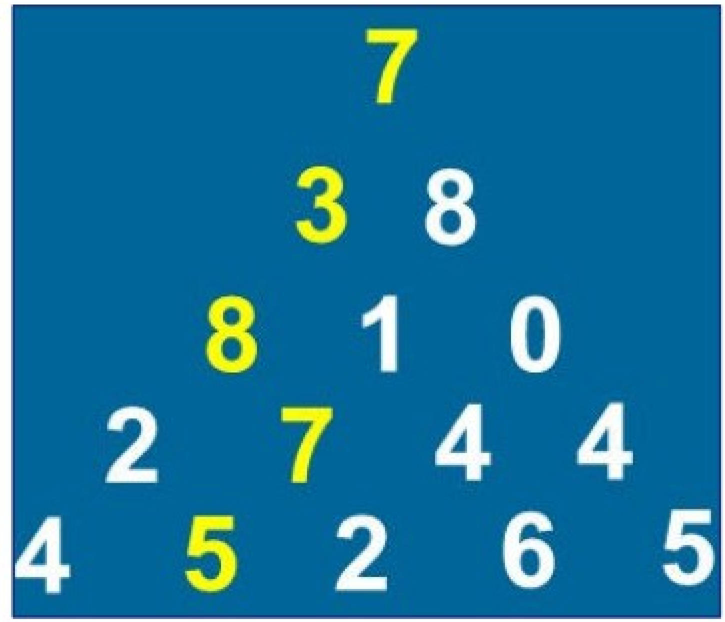
Calcular a rota, que começa no topo da pirâmide e acaba na base, com maior soma . Em cada passo podemos ir diagonalmente para baixo e para a esquerda ou para baixo e para a direita.

Pirâmide de números

- Duas possíveis rotas



Soma = 21



Soma = 30

- **Limites:** todos os números da pirâmide são inteiros entre 0 e 99 e o número de linhas do triângulo é no máximo 100.

Pirâmide de números

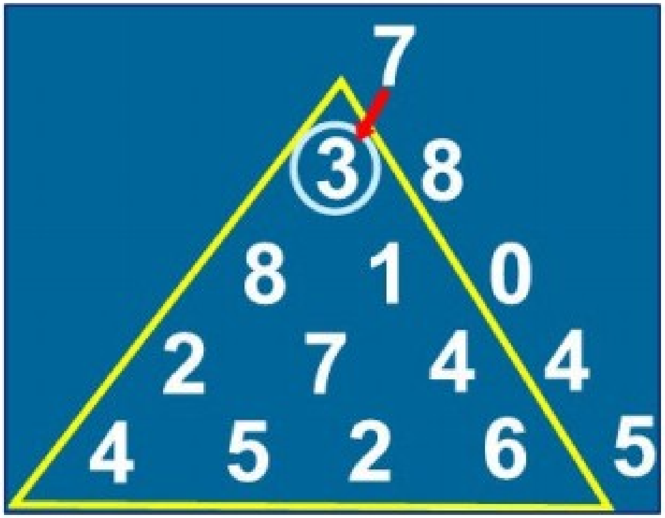
- Como resolver o problema?
- Ideia: **Força Bruta!**
  - avaliar todos os caminhos possíveis e ver qual o melhor.
- Mas quanto tempo demora isto?
  - Quantos caminhos existem?

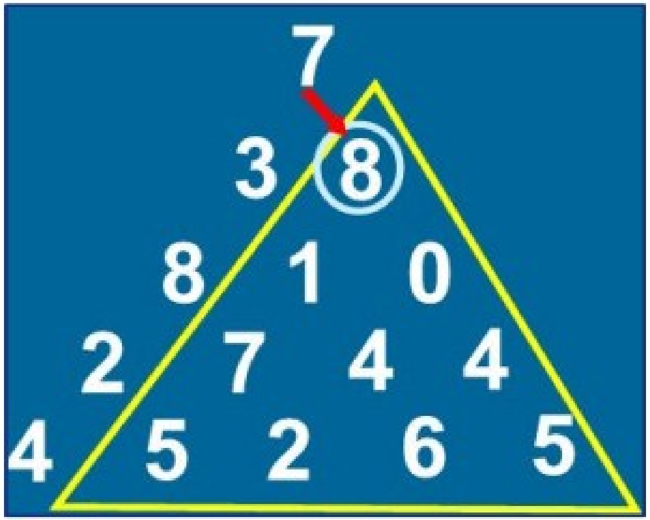
Pirâmide de números

- Análise da complexidade
  - Em cada linha podemos tomar **duas** decisões diferentes: esquerda ou direita
  - Seja n a altura da pirâmide. Uma rota é constituída por **n–1** decisões diferentes.
  - Existem 2^{n-1} caminhos diferentes. Então, um programa que calculasse todas rotas teria complexidade temporal $O(2^n)$: crescimento exponencial!
  - Note-se que $2^{99} \approx 6{,}34 \times 10^{29}$, que é um número demasiado grande!

Pirâmide de números

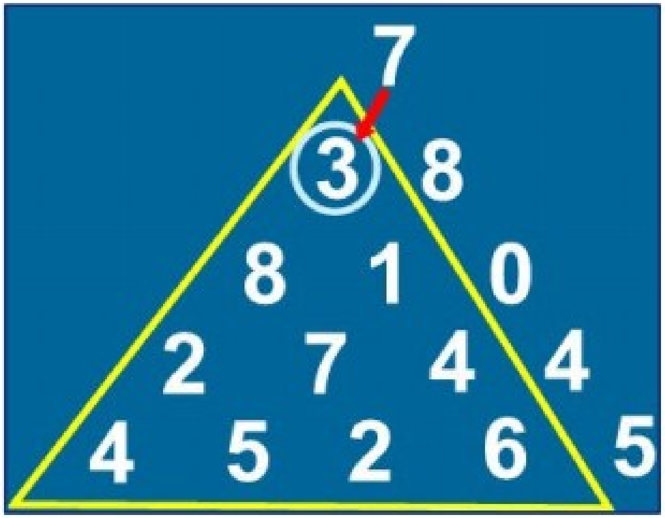
- Quando estamos no topo da pirâmide, temos duas decisões possíveis (esquerda ou direita):

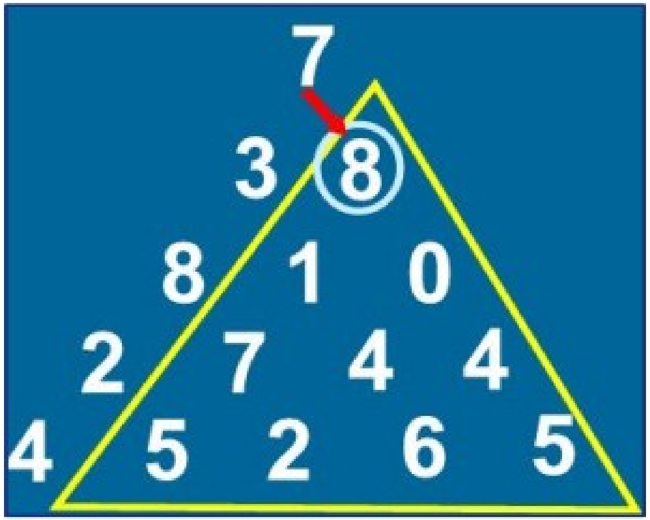




- Em cada um dos casos, temos de ter em conta todas as rotas das respectivas subpirâmides assinaladas a amarelo.

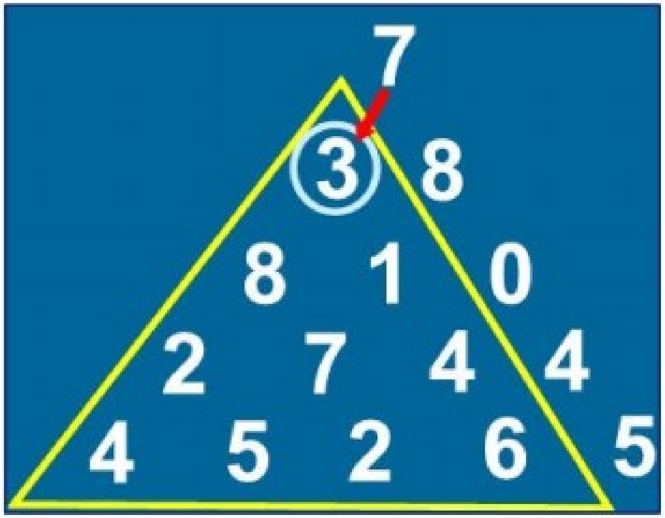
Pirâmide de números

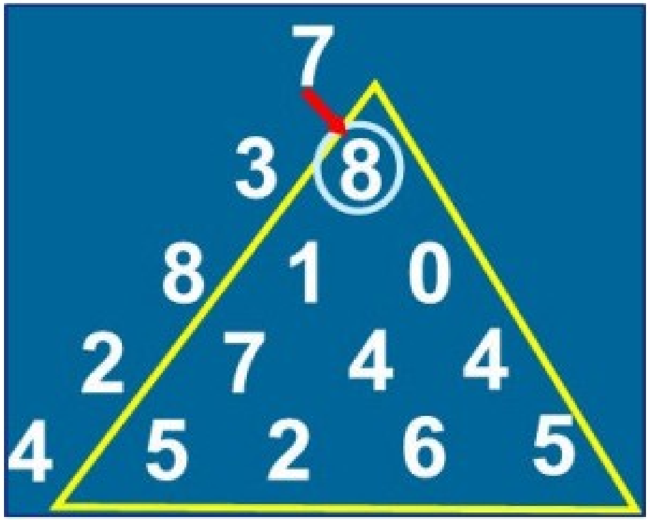




- Mas o que nos interessa saber sobre estas subpirâmides?

Pirâmide de números

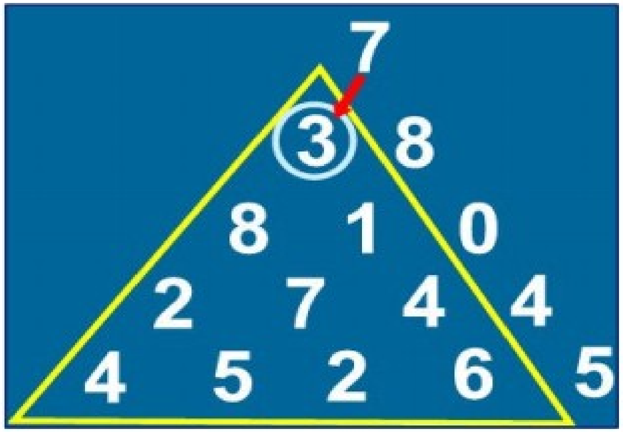


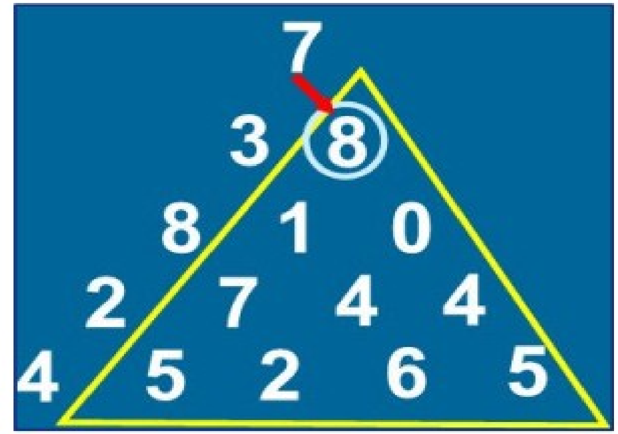


- Mas o que nos interessa saber sobre estas subpirâmides?

Apenas interessa o valor de sua melhor rota interna (que é uma instância menor do mesmo problema)!

Pirâmide de números





- Mas o que nos interessa saber sobre estas subpirâmides?

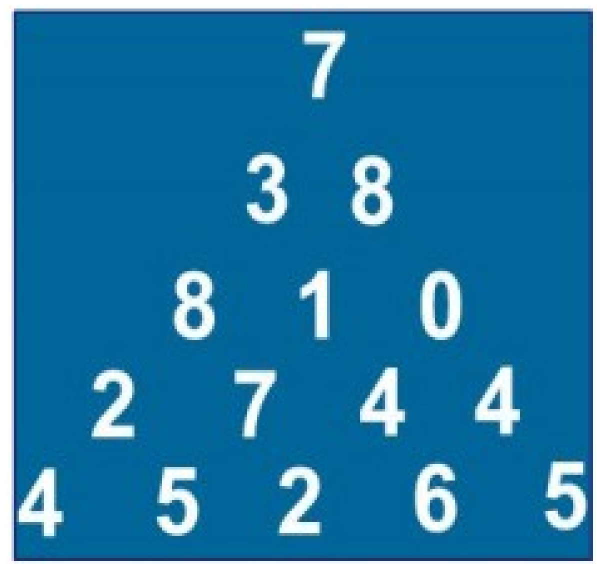
Apenas interessa o valor de sua melhor rota interna (que é uma instância menor do mesmo

- Para o exemplo, a solução é 7 mais o máximo entre o valor da melhor rota de cada uma das subpirâmides

Pirâmide de números

- Então este problema pode ser resolvido recursivamente.
  - Seja **P[i][j]** o j-ésimo número da i-ésima linha
  - Seja **Max(i,j)** o melhor que conseguimos a partir da posição i,j

	1	2	3	4	5
1	7				
2	3	8			
3	8	1	0		
4	2	7	4	4	
5	4	5	2	6	5



Pirâmide de números

- Então:

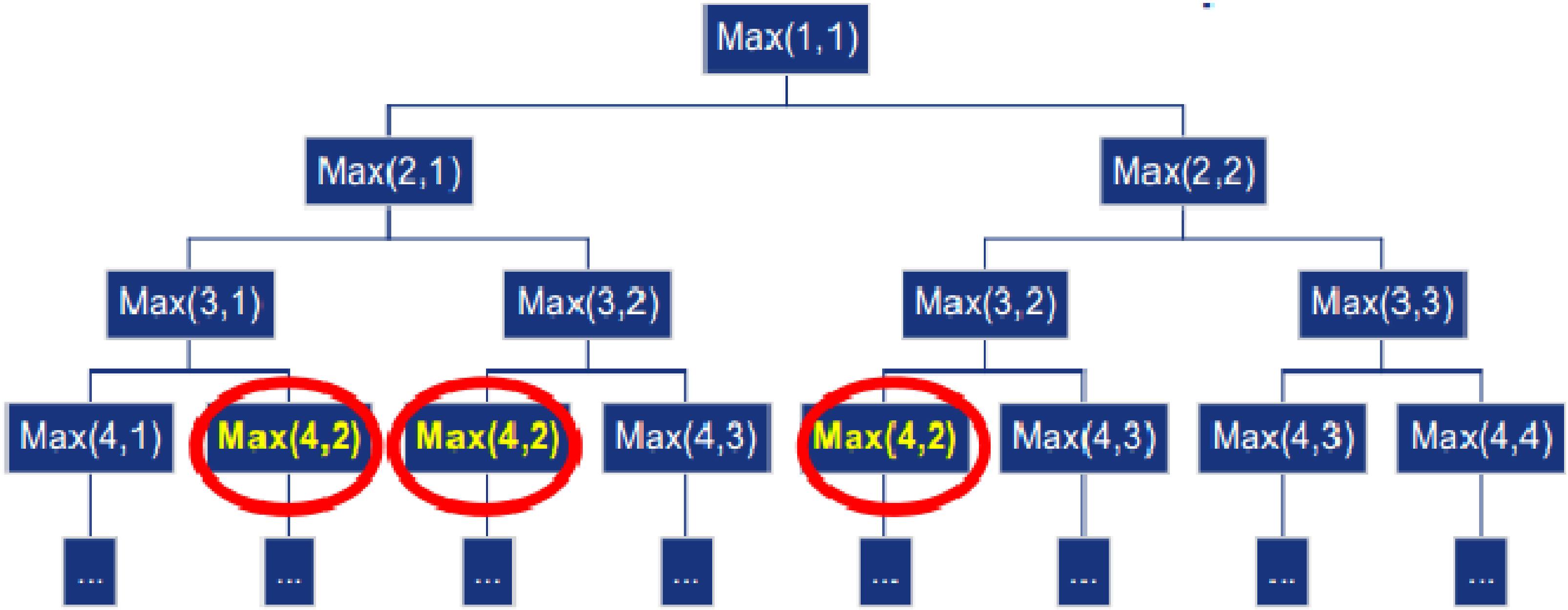
```
Max(i,j):
    Se  i = n  então
        Max(i,j) = P[i][j]
    Senão
        Max(i,j) = P[i][j] + máximo(Max(i+1,j), Max(i+1,j+1))
```

	1	2	3	4	5
1	7				
2	3	8			
3	8	1	0		
4	2	7	4	4	
5	4	5	2	6	5

Para resolver o problema basta chamar Max(1,1)

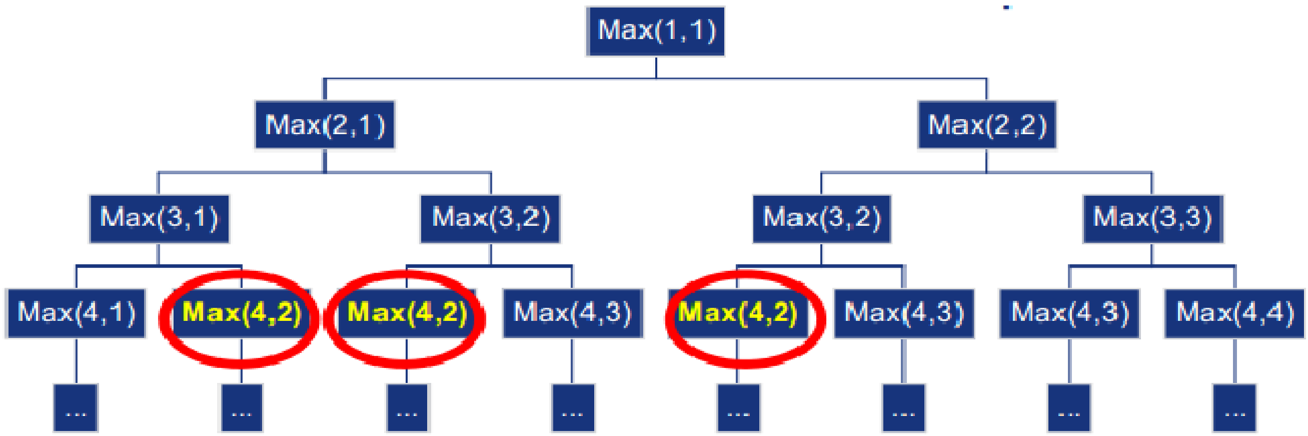
Pirâmide de números

- Continuamos com crescimento exponencial!

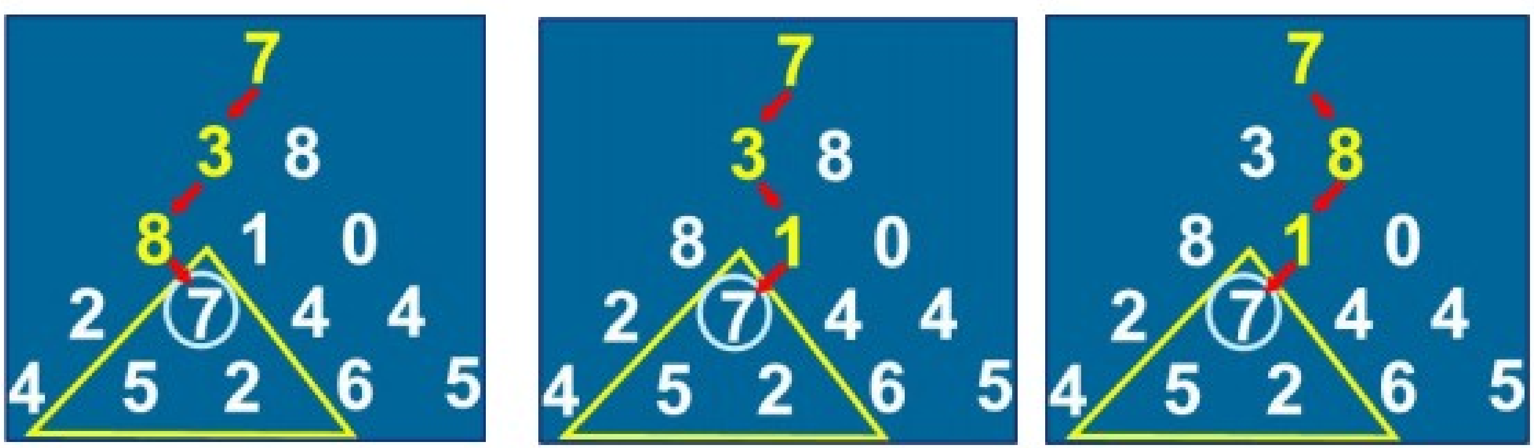


Pirâmide de números

- Continuamos com crescimento exponencial!



- Estamos avaliando o mesmo subproblema várias vezes!



Pirâmide de números

- Temos de **reaproveitar** o que já calculamos
  - Só calcular uma vez o mesmo subproblema!

- Ideia: criar uma **tabela** com o valor obtido para cada subproblema!
  - Matriz M[i][j]

- Será que existe uma **ordem para preencher a tabela** de modo a que quando precisamos de um valor já o temos?

Pirâmide de números

- Começar a partir do fim!

7

3 8

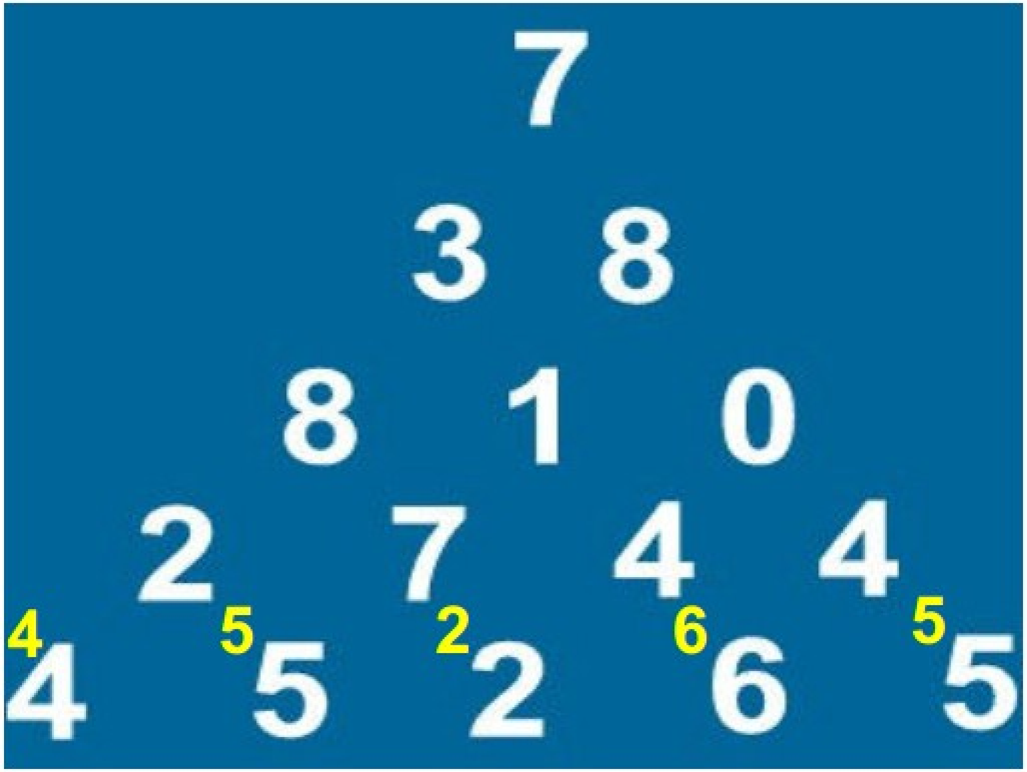
8 1 0

2 7 4 4

4 5 2 6 5

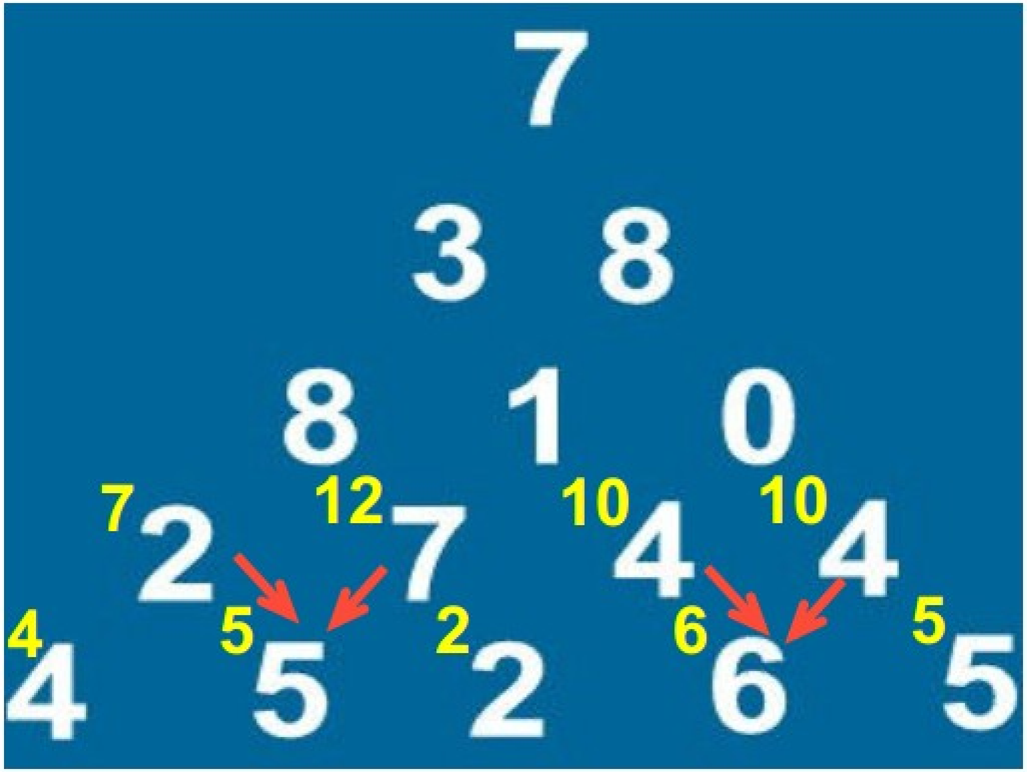
Pirâmide de números

- Começar a partir do fim!



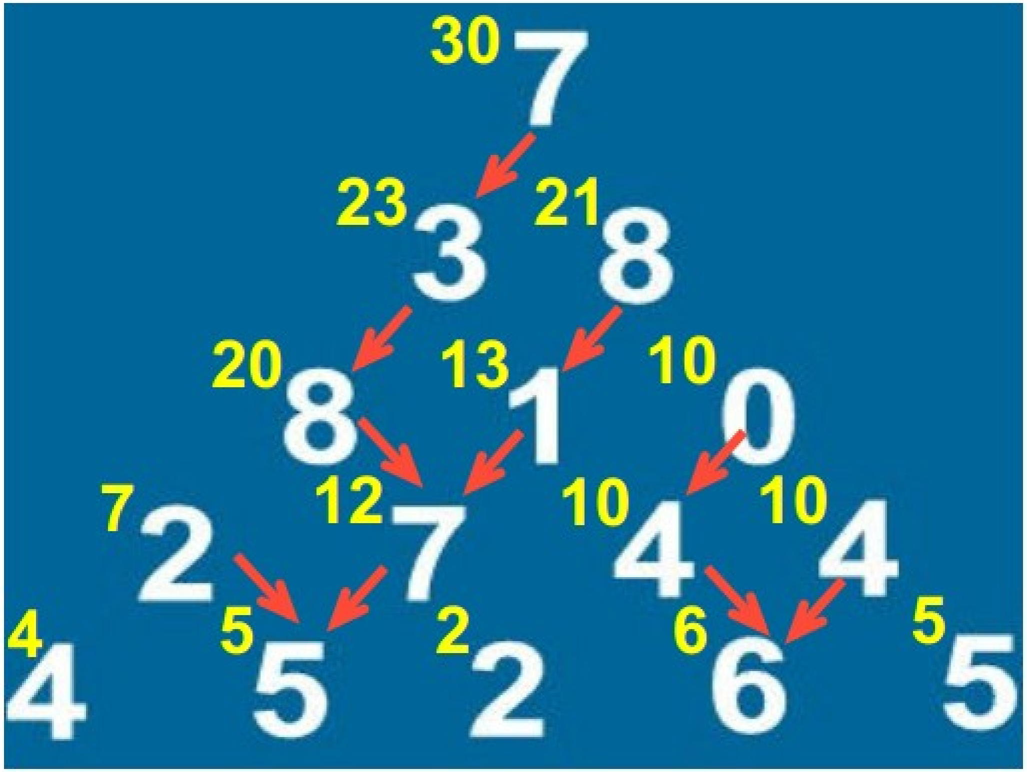
Pirâmide de números

- Começar a partir do fim!



Pirâmide de números

- Começar a partir do fim!



Pirâmide de números

- Tendo em conta a maneira como preenchemos a tabela, até podemos aproveitar **P[][]**!

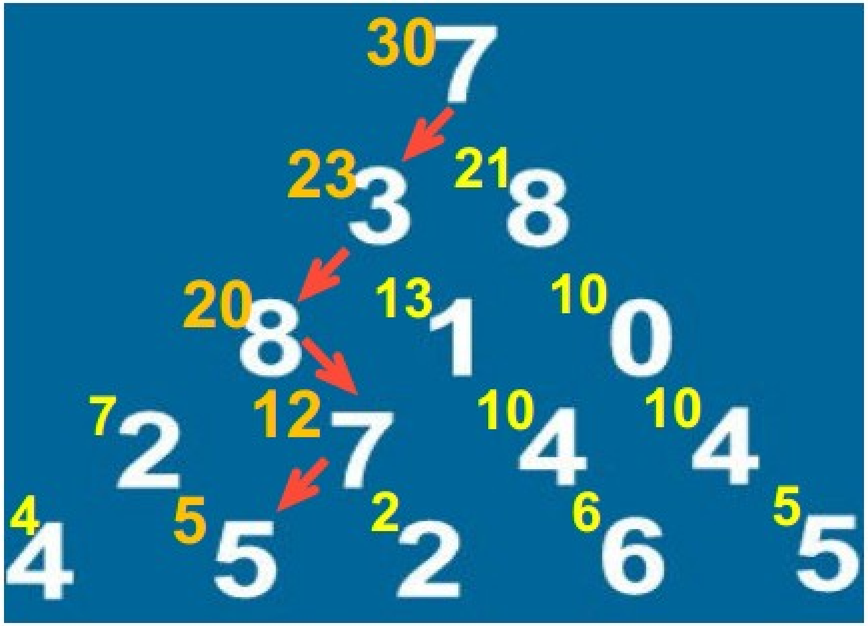
```
Calcular ():
    Para  i: n-1 até 1 fazer
        Para  j: 1 até i fazer
            P[i][j] = P[i][j] + máximo(P[i+1][j], P[i+1][j+1])
```

- Com isto solução fica em **P[1][1]**!

- Agora, o tempo necessário para resolver o problema já só cresce **polinomialmente** (**$O(n^2)$**) e já temos uma solução admissível para o problema (**99^2=9801**)

Pirâmide de números

- Se fosse necessário saber constituição da melhor solução?
  - Basta usar a tabela já calculada!



Pirâmide de números

- Para resolver o problema da pirâmide de números usamos...

Programação Dinâmica (PD)

- Quais são então as características que um problema deve apresentar para poder ser resolvido com PD?
  - Subestrutura óptima
  - Subproblemas coincidentes

Programação Dinâmica - características

Subestrutura óptima ("optimal substructure"):

- Quando a solução óptima de um problema contém nela própria soluções óptimas para subproblemas do mesmo tipo.

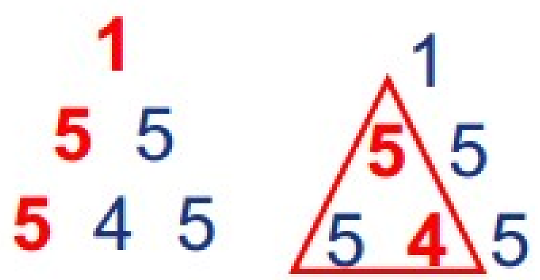
> **Exemplo:** *No problema das pirâmides de números, a solução óptima contém nela própria os melhores percursos de subpirâmides, ou seja, soluções óptimas de subproblemas.*

- Quando um problema apresenta esta característica diz-se que ele respeita o **princípio de optimalidade** ("optimality principle").

Programação Dinâmica - características

- **Subestrutura ótima** **("optimal substructure"):**
  - É preciso ter cuidado porque isto nem sempre acontece!

Exemplo: *se, no problema das pirâmides, o objetivo fosse encontrar a rota que maximizasse o resto da divisão inteira entre 10 e o valor dessa rota.*



A solução óptima (1→5→5) não contém a solução óptima para a subpirâmide assinalada a amarelo (5→4)

Programação Dinâmica - características

Subproblemas coincidentes:

- Quando um espaço de subproblemas é pequeno, isto é, não são muitos os subproblemas a resolver pois muitos deles são exatamente iguais uns aos outros.

Exemplo: *no problema das pirâmides, para um determinada instância do problema, existem apenas* $n+(n-1)+\ldots+1<n^2$ *subproblemas (crescem polinomialmente) pois, como já vimos, muitos subproblemas que aparecem são coincidentes.*

- Também esta característica nem sempre acontece, quer porque mesmo com subproblemas coincidentes são muitos subproblemas a resolver, quer porque não existem subproblemas coincidentes.

Exemplo: *no quicksort, cada chamada recursiva é feita a um subproblema novo, diferente de todos os outros.*

Programação Dinâmica - metodologia

- Se um problema apresenta estas **duas características**, temos uma boa **pista** de que se pode aplicar a PD. No entanto, nem sempre isso acontece.

- Que **passos** devemos então tomar se quisermos resolver um problema usando PD?

Programação Dinâmica - metodologia

- Que **passos** devemos então tomar se quisermos resolver um problema usando PD?

1) Caracterizar a solução óptima do problema

2) Definir recursivamente a solução óptima, em função de soluções óptimas de subproblemas

3) Calcular as soluções de todos os subproblemas:

"de trás para a frente" ou com "memoization"

4) Reconstruir a solução óptima, baseada nos cálculos efetuados (opcional)

Programação Dinâmica - metodologia

- Se um problema apresenta estas **duas características**, temos uma boa **pista** de que se pode aplicar a PD. No entanto, nem sempre isso acontece.

- Que **passos** devemos então tomar se quisermos resolver um problema usando PD?

Programação Dinâmica - metodologia

- Que **passos** devemos então tomar se quisermos resolver um problema usando PD?

1) Caracterizar a solução óptima do problema

2) Definir recursivamente a solução óptima, em função de soluções óptimas de subproblemas

3) Calcular as soluções de todos os subproblemas:

"de trás para a frente" ou com "memoization"

4) Reconstruir a solução óptima, baseada nos cálculos efetuados (opcional)

Programação Dinâmica - metodologia

1) Caracterizar a solução óptima de um problema

- Compreender bem o problema
- Verificar se um algoritmo que verifique todas as soluções à força bruta não é suficiente
- Tentar generalizar o problema (é preciso prática para perceber como generalizar da maneira correcta)
- Procurar dividir o problema em subproblemas do mesmo tipo
- Verificar se o problema obedece ao princípio de optimalidae
- Verificar se existem subproblemas coincidentes

Programação Dinâmica - metodologia

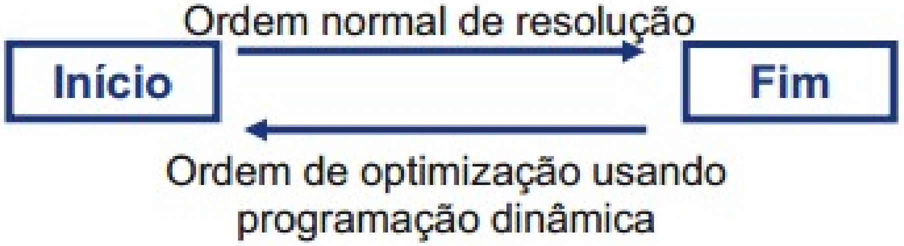
2) Definir recursivamente a solução óptima, em função de soluções óptimas de subproblemas.

- Definir recursivamente o valor da solução óptima, com rigor e exactidão, a partir de subproblemas mais pequenos do mesmo tipo
- Imaginar sempre que os valores das soluções óptimas já estão disponíveis quando precisamos deles
- Não é necessário codificar. Basta definir matematicamente a recursão.

Programação Dinâmica - metodologia

3) Calcular as soluções de todos os subproblemas: "de trás para a frente"

- Descobrir a ordem em que os subproblemas são precisos, a partir dos subproblemas mais pequenos até chegar ao problema global (**"bottom-up"**) e codificar, usando uma tabela.
- Normalmente, esta ordem é inversa à ordem normal da função recursiva que resolve o problema



Programação Dinâmica - metodologia

3) Calcular as soluções de todos os subproblemas: "memoization"

- Existe uma técnica, chamada "**memoization**", que permite resolver o problema pela ordem normal ("**top-down**")
- Usar a função recursiva obtida directamente a partir definição da solução e ir mantendo uma tabela com os resultados dos subproblemas.
- Quando queremos aceder a um valor pela primeira vez temos de calculá-lo e a partir daí basta ver qual é.

Programação Dinâmica - metodologia

4) Reconstruir a solução óptima, baseada nos cálculos efetuados

- Pode ou não ser requisito do problema
- Duas alternativas:
  - **Diretamente** a partir da tabela dos sub-problemas
  - **Nova tabela** que guarda as decisões em cada etapa
- Não necessitando de saber qual a melhor solução, podemos por vezes poupar espaço

Programação de linha de montagem

Linha de montagem automotiva

- Temos uma fábrica com duas linhas de montagens.

- Um chassi de automóvel entra em cada linha de montagem, tem as peças adicionadas a ele em uma série de estações, e um automóvel pronto sai no final da linha.

- Cada linha de montagem possui n estações, numeradas com $j = 1, 2, ..., n$. Cada linha de montagem é numerada com $i = 1$ ou 2.

Linha de montagem automotiva

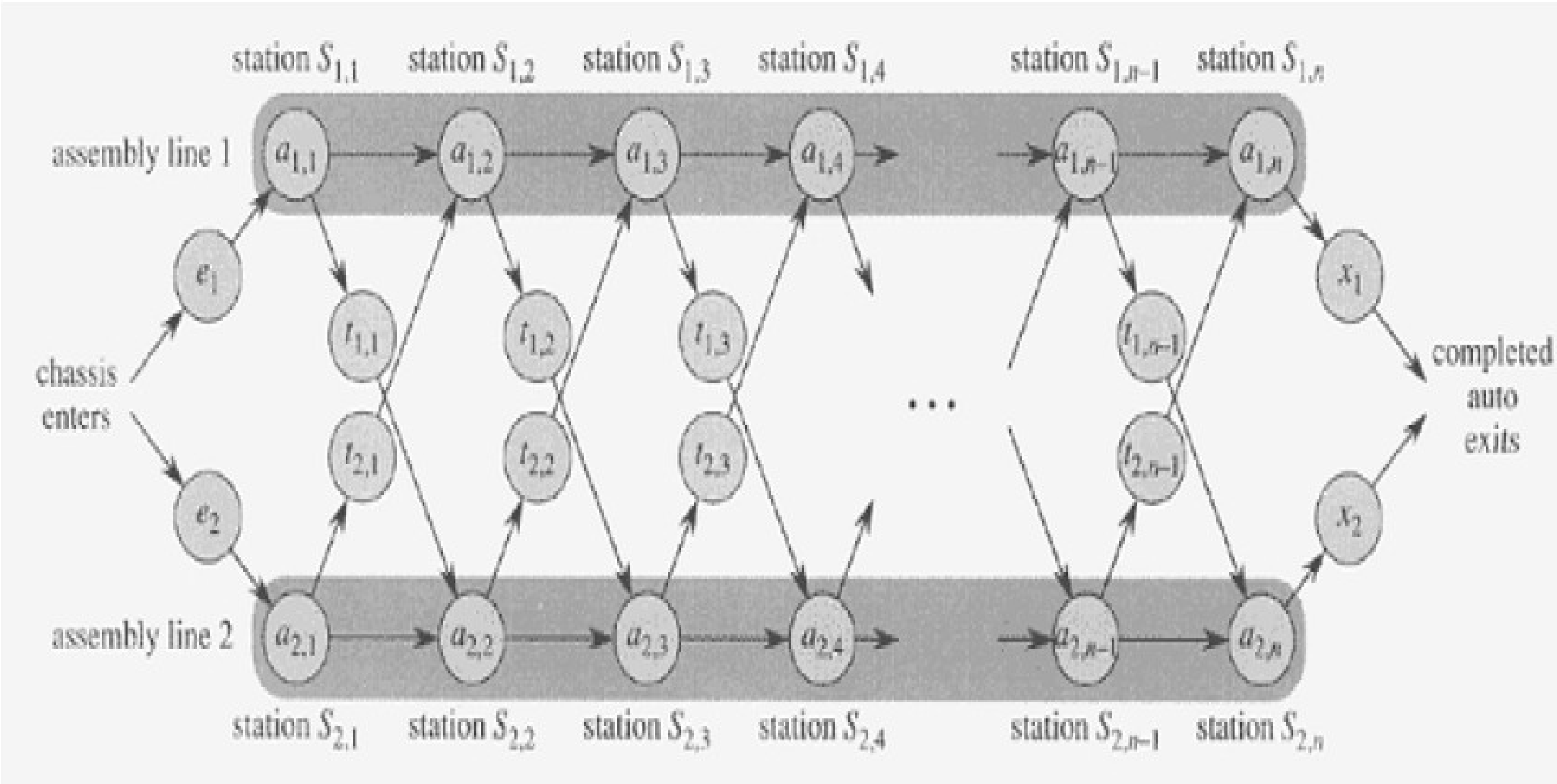
- Temos $S_{i,j}$ como sendo a *j*-ésima estação na linha *i*.

- A *j*-ésima estação da linha 1 ($S_{1,j}$) executa a mesma função que a *j*-ésima estação na linha 2 ($S_{2,j}$).

- Porém, as estações foram construídas em épocas diferentes e com tecnologias diferentes, de forma que o tempo exigido em cada estação varia, até mesmo as estações na mesma posição nas duas linhas.

Linha de montagem

- Denotamos o tempo de montagem exigido na estação $S_{i,j}$ por $a_{i,j}$.
- Além disso, temos um tempo de entrada e_i, um tempo de saída x_i e um tempo para transferir um chassi de uma linha de montagem para outra depois da passagem pela estação $S_{i,j}$, que é $t_{i,j}$.
- É importante ressaltar que uma transferência pode ocorrer até quando a estação for n-1, pois após a *n*-ésima estação, a montagem se completa.



station $S_{1,1}$
station $S_{1,2}$
station $S_{1,3}$
station $S_{1,4}$
station $S_{1,n-1}$
station $S_{1,n}$
assembly line 1
$a_{1,1}$
$a_{1,2}$
$a_{1,3}$
$a_{1,4}$
$a_{1,n-1}$
$a_{1,n}$
e_1
$t_{1,1}$
$t_{1,2}$
$t_{1,3}$
$t_{1,n-1}$
x_1
chassis enters
completed auto exits
$t_{2,1}$
$t_{2,2}$
$t_{2,3}$
$t_{2,n-1}$
e_2
x_2
assembly line 2
$a_{2,1}$
$a_{2,2}$
$a_{2,3}$
$a_{2,4}$
$a_{2,n-1}$
$a_{2,n}$
station $S_{2,1}$
station $S_{2,2}$
station $S_{2,3}$
station $S_{2,4}$
station $S_{2,n-1}$
station $S_{2,n}$

Linha de montagem

- Podemos definir o problema que será resolvido pela programação dinâmica, que é determinar que estações escolher na linha 1 e quais escolher na linha 2 para minimizar o tempo total de passagem de um único automóvel pela fábrica.

- **Dicas:**

- custo é apenas o tempo.

- observar as entradas: "e's", "a's", "t's" e os "x's", pois é o que o programa receberá.

1 - A estrutura de uma solução ótima

- 1ª etapa do paradigma da PD
- Na programação da linha de montagem, uma solução ótima para um problema: encontrar o caminho mais rápido passando pela estação $S_{i,j}$

1 - A estrutura de uma solução ótima

- A solução contém em seu interior uma solução ótima para subproblemas :
- encontrar a passagem mais rápida por $S_{1,j-1}$ ou $S_{2,j-1}$.
- Essa é a propriedade da subestrutura ótima.

2 - Uma solução recursiva

- De acordo com o problema, temos que o caminho mais rápido até completar a estação $S_{1,j}$ é o caminho mais rápido até completar a estação $S_{1,j-1}$, e depois passar diretamente pela a estação $S_{1,j}$, ou o caminho mais rápido até completar a estação $S_{2,j-1}$, uma transferência da linha 2 para a linha 1, e depois passar pela estação $S_{1,j}$ (toma-se o que for menor dos dois).
- Um raciocínio análogo vale para completar $S_{2,j}$.

2 - Uma solução recursiva

Recorrências :

- $f_{1j} = e_1 + a_{1,1}$ se $j = 1$.
- $f_{1j} = \min(f_{1(j-1)} + a_{1,j}\ ,\ f_{2[j-1]} + t_{2,j-1} + a_{1,j})$ se $j >= 2$.

- $f_{2j} = e_2 + a_{2,1}$ se $j = 1$.
- $f_{2j} = \min(f_{2(j-1)} + a_{2,j}\ ,\ f_{1[j-1]} + t_{1,j-1} + a_{2,j})$ se $j >= 2$.

2 - Uma solução recursiva

- Quantas chamadas recursivas um algoritmo precisa realizar para resolver esse problema da linha de montagem?
- No máximo duas, mas há um caso em que nenhuma chamada recursiva é feita: o caso da primeira estação.

2 - Uma solução recursiva

Algoritmo

```
TEMPO(n)
  Se(f1(n) + x1) < (f2(n) + x2)
     Então tempominimo = f1(n) + x1
           saidaporlinha = 1
     Senão tempominimo = f2(n) + x2
           saidaporlinha = 2
```

2 - Uma solução recursiva

$f_{i(n)}$

  Se $(n = 1)$

    então devolver $a_{i(1)} + e_i$

  Se $f_{i(n-1)}+a_{i(n)} < f_{\hat{\imath}(n-1)}+t_{\hat{\imath}(n-1)}+a_{i(n)}$

    então devolver $f_{i(n-1)}+a_{i(n)}$

devolver $f_{\hat{\imath}(n-1)} + t_{\hat{\imath}(n-1)} + a_{i(n)}$

î vale 1 se i for igual a 2 ou 2 se i for igual a 1

solução gera muitas chamadas recursivas, inviabilizando seu uso.

3 - Cálculo do valor de uma solução ótima

um algoritmo utilizando a técnica de programação dinâmica, consegue calcular o caminho mais rápido pela fábrica e o tempo que ele demora, no tempo $\Theta(n)$.

Atenção para o uso de 4 vetores no lugar da recursão: f_1, f_2, l_1 e l_2.

```
FASTEST-WAY(a, t, e, x, n)
f1[1] ← e1+ a1,1
f2[1] ← e2 + a2,2
for j ← 2 to n
      do if f1[j-1] + a1,j <= f2[j-1] + t2,j-1 + a1,j
          then  f1[j] ← f1[j-1] + a1,j
                      l1[j] ← 1
          else  f1[j] ← f2[j-1] + t2,j-1 + a1,j
                      l2[j] ← 2
        if f2[j-1] + a2,j <= f1[j-1] + t1,j-1 + a2,j
           then f2[j] ← f2[j-1] + a2,j
                      l2[j] ← 2
           else f2[j] ← f1[j-1] + t1,j-1 + a2,j
                      l2[j] ← 1
  if f1[n] + x1 <= f2[n] + x2
     then  f* = f1[n] + x1
               l* = 1
     else  f* = f2[n] + x2
               l* = 2
```

4. Construção de uma solução ótima

procedimento PRINT-STATIONS, imprime as estações usadas, em ordem decrescente de número de estações.

E imprime a ordem inversa, da última estação para a primeira.

4. Construção de uma solução ótima

```
PRINT-STATIONS(l,n)
i ← l*
imprimir "linha" i ",estação" n
for j ← n downto 2
      do i ← l_i[j]
      imprimir "linha" i ",estação" j-1
```

! ...