

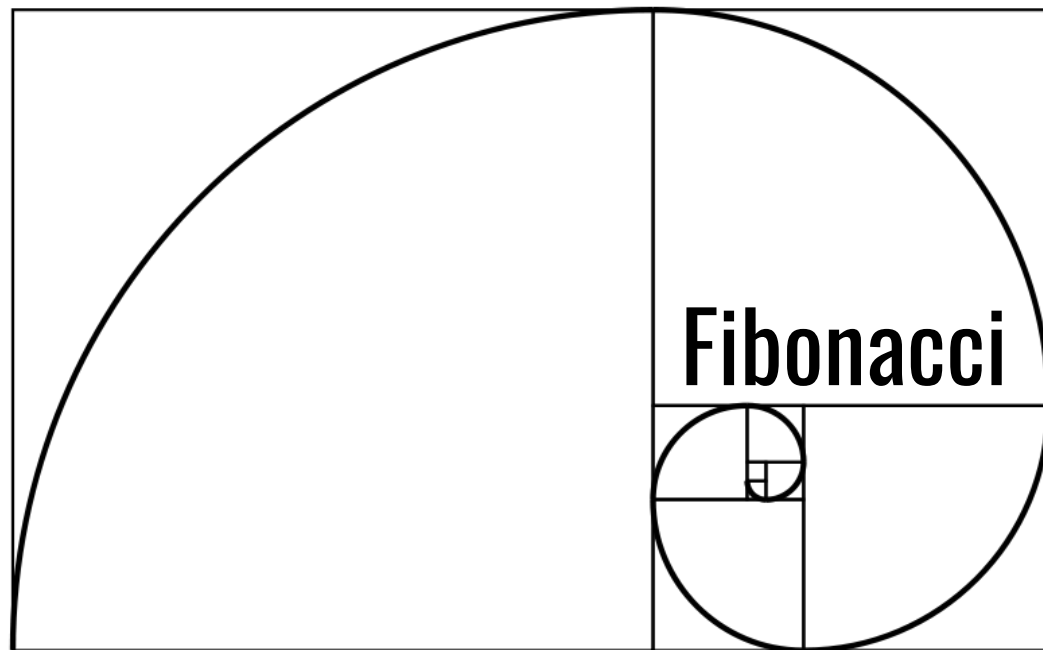


Maratona **Cln**

**Seletiva 2020**

# Programação Dinâmica (aula 1)

Aula 5



# Fibonacci

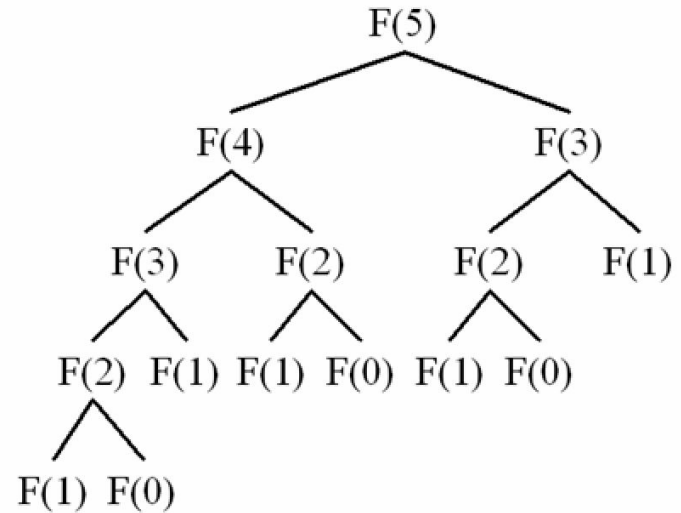
```
int fibo(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return fibo(n - 1) + fibo(n - 2);  
}
```

$$F_n = F_{n-1} + F_{n-2}$$

# Fibonacci

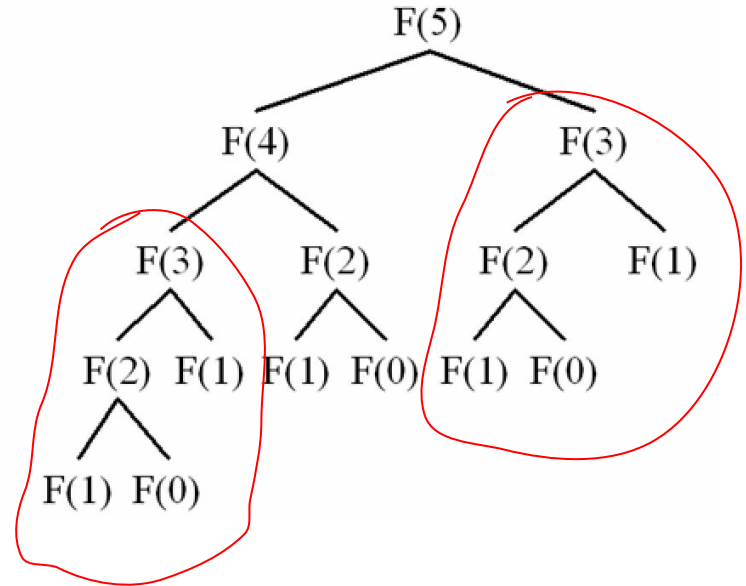


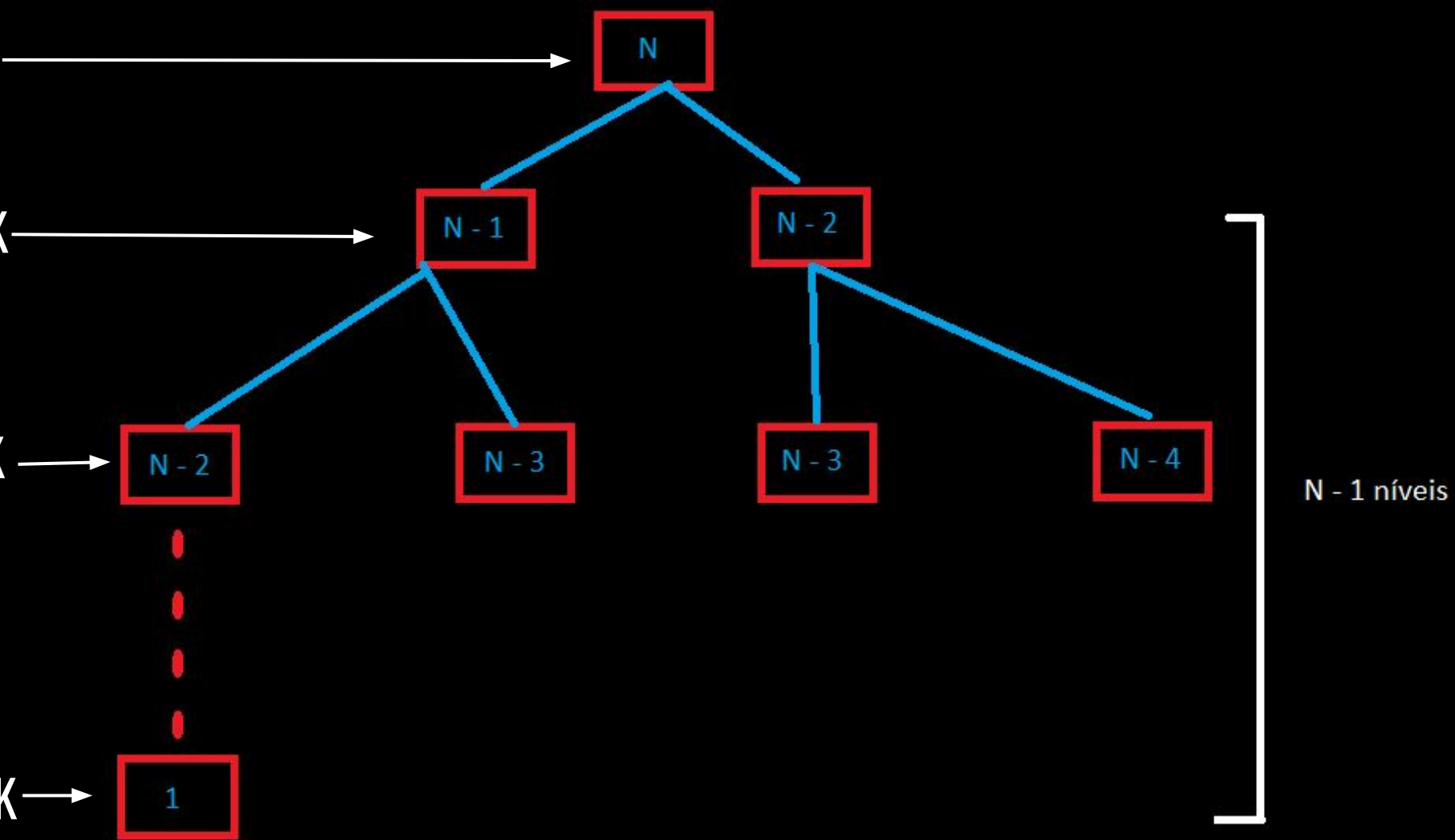
```
int fibo(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return fibo(n - 1) + fibo(n - 2);  
}
```

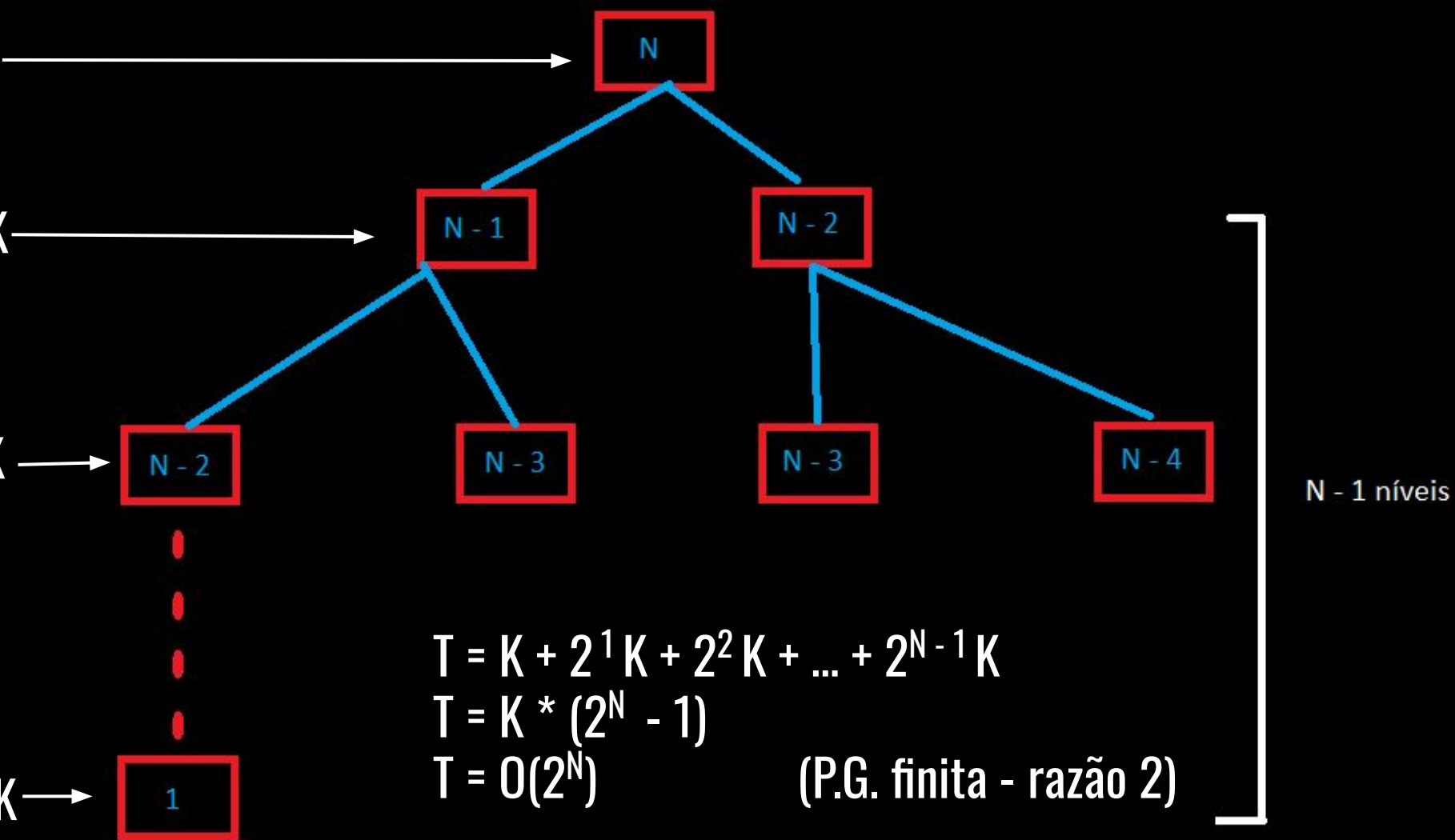


# Fibonacci

```
int fibo(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return fibo(n - 1) + fibo(n - 2);  
}
```









# Fibonacci

---



Gargalo identificado: recalcular vários estados inúmeras vezes.

Ideia: se eu consigo ver que o resultado já foi calculado e tenho como acessá-lo, não preciso recalcular!

-> Memorização!

# Fibonacci (versão recursiva)



```
const int MAX_FIB_N = 20;
```

```
ll memo[MAX_FIB_N]; // inicializado com -1
```

```
ll fib(int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    if (memo[n] != -1) return memo[n];  
    return memo[n] = fib(n - 1) + fib(n - 2);  
}
```

**Complexidade:  $O(N)$**

**Ideia: cada número de Fibonacci de 0 a N só precisa ser calculado uma vez**

# Fibonacci (versão iterativa)



MaratonaCIn

```
const int MAX_FIB_N = 20;
```

```
ll fibo[MAX_FIB];
```

```
int main() {
```

```
    fibo[0] = 0;
```

```
    fibo[1] = 1;
```

```
    for (int i = 2; i < MAX_FIB_N; ++i) {
```

```
        fibo[i] = fibo[i - 1] + fibo[i - 2];
```

```
    }
```

```
}
```

# Programação Dinâmica

---



Maratona CIn

- Armazenar resultados obtidos, evitando recalculá-los
- Estados
  - que tipo de informação eu preciso ter para identificar unicamente em que ponto da solução eu estou?
- Transições
  - dado um estado, que escolhas eu tenho a fazer, isto é, de que modo eu posso usar os recursos que definem aquele estado para continuar a solução?

# Problema da Mochila

- Dado um conjunto de itens, onde o  $i$ -ésimo item possui peso  $W_i$  e preço  $P_i$ , e uma mochila de capacidade total  $S$ , escolha um subconjunto de itens, tal que a soma dos seus pesos não excede  $S$ , e a soma dos seus preços é máxima.
  - Brute-force
    - gerar todos subconjuntos e, dentre os válidos, escolher o que maximiza o preço total;
      - para cada item, eu tenho uma opção: ou eu coloco ele na mochila, ou não
        - ao chegar no item, eu preciso saber se ele cabe na mochila. portanto, eu preciso, a todo momento, saber quanto espaço livre há na mochila

# Problema da Mochila



```
int brute(int item_atual, int espaco_livre) {
    if (item_atual == num_itens) return 0;
    int melhor_opcao = -1;
    int solucao_com_item = -1;
    if (espaco_livre >= peso[item_atual]) {
        int novo_espaco_livre = espaco_livre - peso[item_atual];
        solucao_com_item = brute(item_atual + 1, novo_espaco_livre) +
preco[item_atual];
    }
    int solucao_sem_item = brute(item_atual + 1, espaco_livre);
    melhor_opcao = max(solucao_com_item, solucao_sem_item);
    return melhor_opcao;
```

# Problema da Mochila



- Problema: gerar todos os subconjuntos, dado um conjunto de  $N$  elementos  
-> complexidade  $O(2^N)$ .
- Solução: memoizar!
- Estados
  - como visto previamente, as informações relevantes que definem bem em que ponto do problema eu estou: qual item eu estou olhando, e qual o espaço livre que eu tenho.
  - `dp[MAX_ITENS][MAX_PESO]`
- Transições
  - opções: escolher um item, caso possível, ou não escolhê-lo
  - pos -> em qual item estou (posição no vetor)
    - escolher item: `solve(pos + 1, espaco_livre - peso[pos]) + preco[pos]`
    - não escolher item: `solve(pos + 1, espaco_livre)`

# Problema da Mochila (versão recursiva)



```
int solve(int pos, int espaco_livre) {
    if (pos == num_itens) return 0;
    if (dp[pos][espaco_livre] != -1) return dp[pos][espaco_livre];
    int melhor_opcao = -1;
    int solucao_com_item = -1;
    if (espaco_livre >= peso[pos]) {
        int novo_espaco_livre = espaco_livre - peso[pos];
        solucao_com_item = solve(pos + 1, novo_espaco_livre) + preco[pos];
    }
    int solucao_sem_item = solve(pos + 1, espaco_livre);
    melhor_opcao = max(solucao_com_item, solucao_sem_item);
    return dp[pos][espaco_livre] = melhor_opcao;
}
```



# Problema da Mochila (versão recursiva)

---



MaratonaCIn

- Análise de complexidade:
  - cada estado só precisa ser calculado uma única vez
  - temos  $\text{MAX\_ITENS} * \text{MAX\_CAP}$  estados
  - cada estado possui uma transição  $O(1)$  - escolher ou não um item
  - complexidade total = complexidade\_transicao \* qtd\_estados
- Finalmente, complexidade (tempo e memória) da DP:
  - $O(\text{MAX\_ITENS} * \text{MAX\_CAP})$

# Problema da Mochila (versão iterativa)

```
// lembrar de inicializar o vetor dp
// com as bases da solução
for (int item = 1; item <= QTD_ITENS; ++item) {
    for (int espaco_livre = 0; espaco_livre <= CAP_MOCHILA; ++espaco_livre) {
        int pegar_item = -1;
        int nao_pegar_item = dp[item - 1][espaco_livre];
        if (espaco_livre >= peso[item]) {
            pegar_item = dp[item - 1][espaco_livre - peso[item]] + preco[item];
        }
        dp[item][espaco_livre] = max(pegar_item, nao_pegar_item);
    }
}
```

# Subarray de a soma máxima

---



- Dado um array  $A$ , encontre o subarray com a soma máxima.
  - Bruteforce:
    - Para todo subarray de  $A$ , calcule a soma e atualize a resposta.
    - $O(n^2)$  de complexidade
    - Muito lento!
    - Como melhorar? :P



# Subarray de soma máxima (brute force)

```
long long solve(vector<int> A) {  
    int n = (int) A.size();  
    long long ans = -1e18;  
    for (int i = 0; i < n; i++) {  
        long long cur = 0;  
        for (int j = i; j < n; j++) {  
            cur += A[j];  
            ans = max(ans, cur);  
        }  
    }  
    return ans;  
}
```

# Subarray com a soma máxima

---



Maratona CIn

- Solução: programação dinâmica.
- Estado:
  - Apenas um inteiro id.
  - $dp[id]$  = melhor resposta olhando as posições do array até o índice id.
- Transições:
  - $dp[id] = \max(\text{preff}[id] - mn[id - 1], dp[id - 1])$
  - $\text{preff}[id] = A[1] + A[2] + \dots + A[id]$
  - $mn[id] = \min(\text{preff}[1], \text{preff}[2], \dots, \text{preff}[id])$

# Subarray de soma máxima (PD iterativa)



Maratona CIn

```
long long solve(vector<int> A) {  
    int n = (int) A.size();  
    vector<long long> dp(n + 1), preff(n + 1), mn(n + 1);  
    dp[0] = -1e18;  
    for (int i = 0; i < n; i++) {  
        preff[i + 1] = preff[i] + A[i];  
        mn[i + 1] = min(preff[i + 1], mn[i]);  
        dp[i + 1] = max(dp[i], preff[i + 1] - mn[i]);  
    }  
    return dp[n];  
}
```

# Longest Increasing Subsequence (LIS)

---



- Problema: dado um array  $A$ , encontre a maior subsequência crescente.
  - Bruteforce:
    - Gerar todas subsequências do array e testar se ela é crescente e maior que a resposta atual
    - $O(2^N)$  de complexidade
    - Muito ruim

# Longest Increasing Subsequence (LIS)

---



- Solução com programação dinâmica:
- Estado:
  - $dp[i]$  = maior subsequência que termina no  $i$ -ésimo elemento, considerando apenas subsequências que contém elementos até o índice  $i$  do vetor
  - vetor  $dp$  iniciado com valor 1 em todos os índices (sequência contendo apenas o próprio elemento)
- Transições:
  - Para todo  $j < i$ , se  $A[j] < A[i]$ ,  $dp[i] = \max(dp[i], dp[j] + 1)$
- Complexidade:
  - tempo:  $O(n^2)$
  - memória:  $O(n)$



# Longest Increasing Subsequence (LIS)



```
// lembrar de inicializar o vetor dp
// com as bases da solução
for(int i = 0; i < n; ++i){
    for(int j = 0; j < i; ++j){
        if(A[j] < A[i]){ // < ou <= dependendo do problema
            dp[i] = max(dp[i], dp[j] + 1);
        }
    }
} // O(N2)
```

# Longest Increasing Subsequence (LIS)



- Existe uma solução melhor, com complexidade de tempo  $O(n \log n)$ 
  - Crie um vetor auxiliar  $S$ , cujo tamanho, no final, será o tamanho da LIS
  - Para cada elemento  $A[i]$  do vetor, faça uma busca binária em  $S$  para achar o primeiro elemento maior ou igual que ele (lower\_bound)
    - Se não existir valor maior ou igual em  $S$ , colocar  $A[i]$  no final de  $S$
    - Se existir valor maior ou igual, substituir o valor na menor posição possível em  $S$  por  $A[i]$
  - A resposta será o tamanho do vetor auxiliar  $S$

Obs: Só funciona para LIS estritamente crescente.

Obs2: o vetor auxiliar  $S$  apenas nos dá a resposta (tamanho da LIS), e não a sequência em si.

# Longest Increasing Subsequence (LIS)



MaratonaCIn

```
int lis(int A[], int n){  
    vector<int> S;  
    for(int i = 0; i < n; i++){  
        auto it = lower_bound(S.begin(), S.end(), a[i]);  
        if(it == S.end())  
            S.push_back(A[i]);  
        else  
            *it = A[i];  
    }  
    return S.size();  
} // O(nlogn)
```