Análise de Algoritmos

Conceitos básicos, critérios de complexidade de tempo e espaço

Análise de Algoritmos

- Prever os recursos requisitados pelo algoritmo:
  - Interesse primário: memória, largura de banda de comunicação, hardware do computador.
  - Análise de algoritmos: medir o tempo computacional.
- Objetivo: identificar algoritmos mais eficientes.
  - Pode existir mais de um candidato viável.
  - Descartamos algoritmos inferiores durante o processo de análise.

Algoritmo

- Sequência de passos computacionais que transforma entrada (input) em saída (output).
- Ferramenta capaz de resolver um problema computacional bem definido.

Exemplo de Problema Computacional: Ordenação

- Problema computacional que surge em diversas situações.
- Definição:
  - Input: Uma sequência de n números:$\{a_1\ a_2, a_3, \ldots, a_n\}$
  - Output: Uma reordenação da sequência em $\{a'_1, a'_2, a'_3 \ldots a'_n\}$ tal que $a'_1 \leq a'_2 \leq a'_3 \leq \ldots \leq a'_n$

Exemplo de Problema Computacional: MDC

- Definição:
  - Input: inteiros positivos a e b
  - Output: c= MDC(a,b), Maior divisor comum de a e b.

Instância do problema e corretude do algoritmo

- A **instância de um problema** consiste da entrada necessária para calcular a solução do problema.
- A **corretude de um algoritmo** indica que ele termina sua execução, retornando saídas corretas para todas as instâncias do problema.
- Todavia, algoritmos ” incorretos” podem ser úteis se for possível controlar sua taxa de erro!
- Neste curso, apenas algoritmos corretos serão abordados.

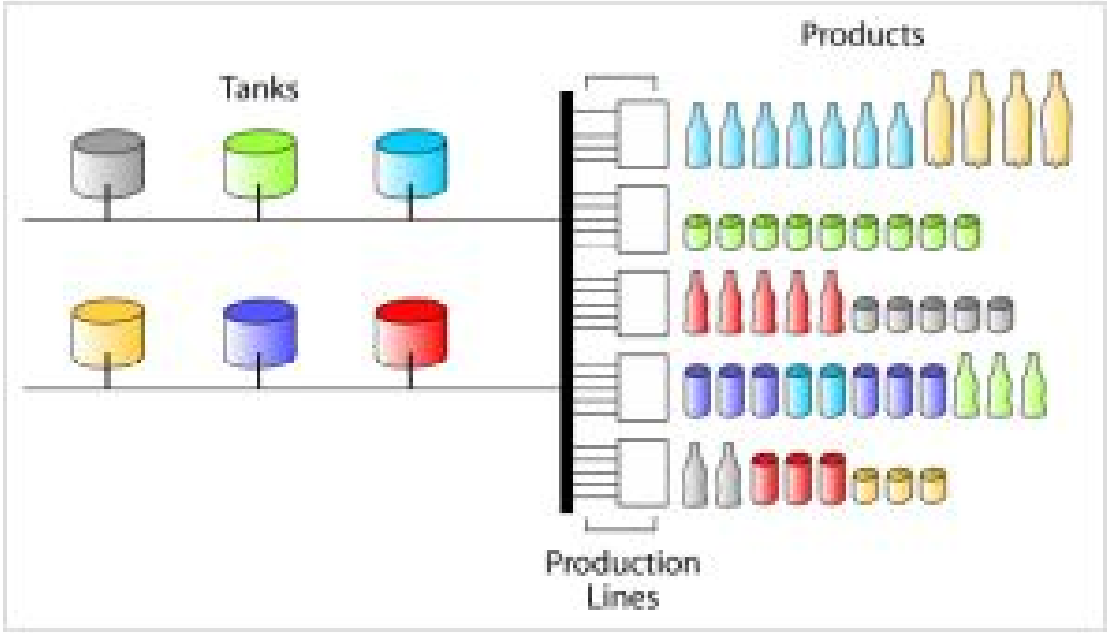
Pseudocódigo

- Podemos especificar um algoritmo como
  - Programa de computador: uso de linguagem de programação como C++, Java, python, etc.
  - Hardware design.
- A especificação deve fornecer uma descrição precisa do procedimento computacional a ser seguido.
- Uma forma mais geral de especificar algoritmos é através de pseudocódigo.

Pseudocódigo

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 . . j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

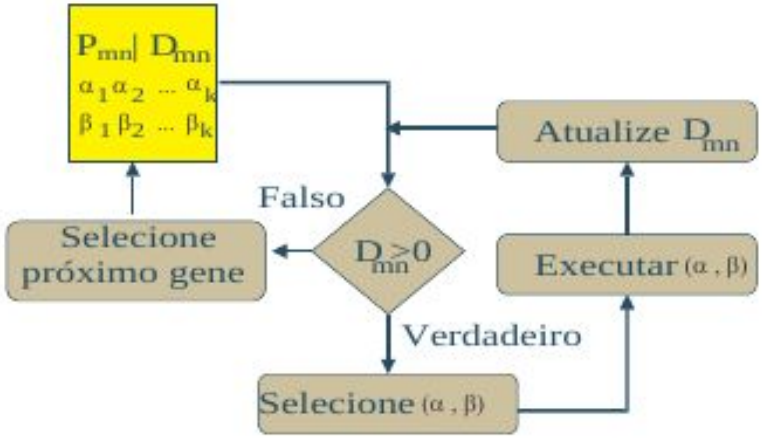
Problemas solucionados por algoritmos



Problemas solucionados por algoritmos

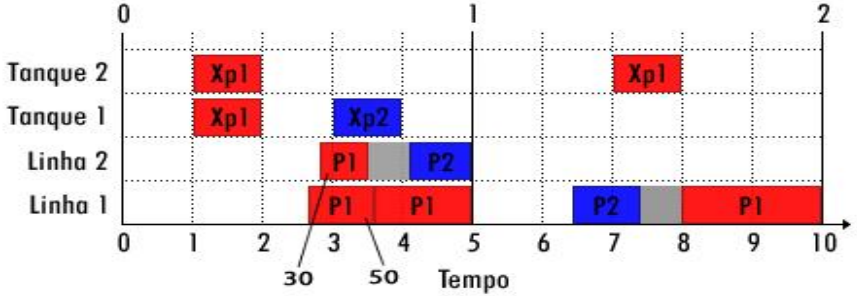
Indivíduo 1

T_1	P1 | 70 1 2 2 1 2 2 3 2	P2 | 110 2 2 2 2 3 3 4 2	P1 | 80 1 2 1 2 4 3 2 2
T_2	P1 | 100 1 1 2 2 4 2 1 3	P2 | 120 1 1 1 2 1 3 3 2	



T_2

P1 | 80 1 2 1 2 4 3 2 2



Algoritmo 2: Gerando níveis aleatórios.

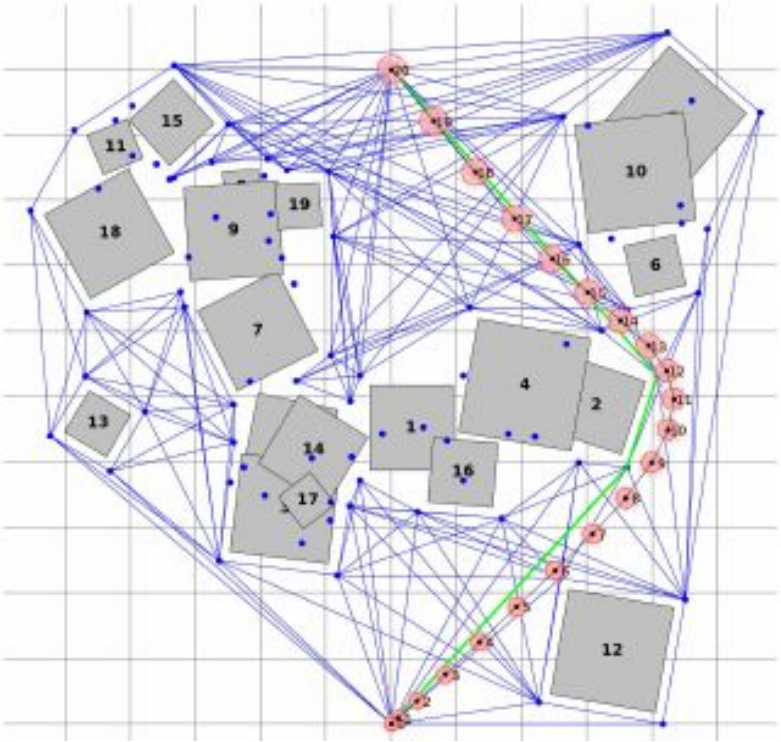
```
1  A ← {};
2  b ← U(1, B);
3  w_preenchida ← 0;
4  w_A ← N(W * 0.5, W * 0.5);
5  enquanto w_preenchida < w_A faça
6      S ← {};
7      V ← todos os blocos;
8      h_preenchida ← 0;
9      h_s ← N(H * 0.5, H * 0.5);
10     enquanto h_preenchida < h_i e V não está vazio faça
11         c_prox ← EscolheClasseSeguraAleatoria(c_ant);
12         l_prox ← EscolheBlocoAleatorioDaClasse(c_prox, V);
13         se c_prox não é uma caixa então
14             se c_prox não é um pódio e U(0,1) < 0.5 então
15                 DuplicaBloco(l_prox);
16             se AreaQuadDelimitador_ant > AreaQuadDelimitador_prox então
17                 adiciona l_prox em S;
18                 h_preenchida ← h_preenchida + h_prox;
19         senão se blocos no topo de S cabem dentro de l_prox então
20             adiciona l_prox em S;
21             h_preenchida ← h_preenchida + h_prox;
22     InserePorcos(S);
23     se h_preenchida > 0 então
24         w_preenchida ← w_preenchida + w_s;
25     senão
26         w_preenchida ← w_preenchida + k;
27     adiciona S em A;
```

		31 | 0				31 | 0		
	31 | 0	13 | 0				13 | 0		
	13 | 0	28 | 0				28 | 0		
	12 | 0	29 | 1				29 | 0		
	13 | 1	10 | 1		31| 0		15 | 0		31| 0
4	11 | 0	24 | 1	X	18| 0	X	25 | 1	X	18| 0
pássar.	pilha 1	pilha 2	pilha 3	pilha 4	pilha 5	pilha 6	pilha 7	pilha 8

Problemas solucionados por algoritmos

Algorithm 2: HISA

```
1  model ← CNPP^LEN;
2  n ← 0;
3  repeat
4      G ← create-graph(Δ/2^n);
5      path ← Dijkstra(G);
6      submodel ← reference-path(model, path, n);
7      Execute Algorithm1(submodel, FRR) with Δ/2^n;
8      n ← n + 1;
9  until isFeasible(submodel.status) or n ≥ maxIter;
10 if isFeasible(submodel.status) then
11     return model;
12 else
13     return HISA fail;
```



Problemas solucionados por algoritmos

- Problemas podem apresentar muitas soluções candidatas.
- A maioria não soluciona ou não está entre as melhores soluções a serem utilizadas em aplicações práticas.
- O objetivo é elaborar algoritmos corretos e eficientes.
- Porém, há problemas para os quais não existem algoritmos eficientes.

Algoritmos e problema polinomiais.

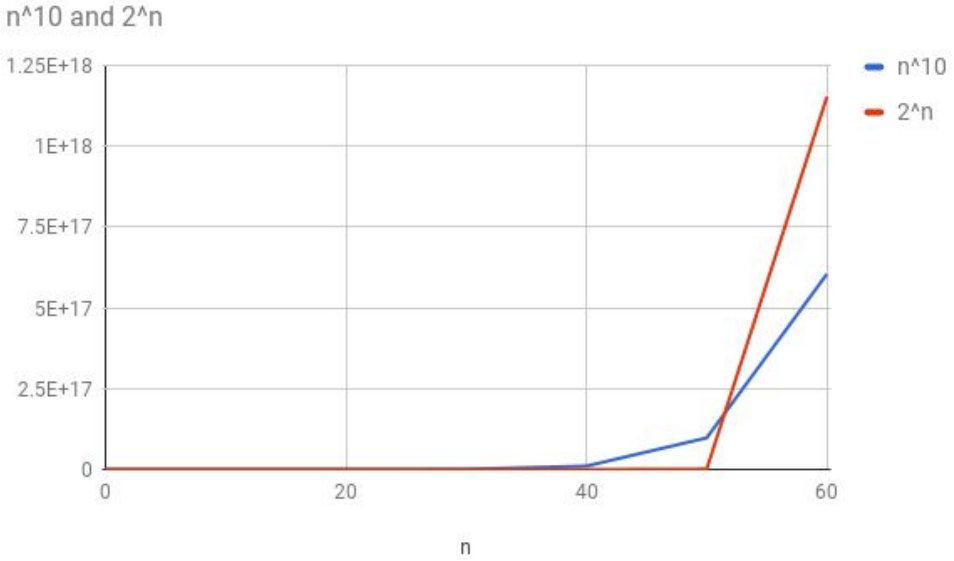
- **Algoritmo polinomial** é aquele que gasta um tempo limitado por um polinômio para solucionar instâncias do problema.
- O polinômio é dado em função do tamanho das instâncias do problema.
- Se existir um algoritmo polinomial para uma problema computacional, o problema é classificado como um **problema polinomial.**

Algoritmos e problema polinomiais.

- **Classe P** de problemas polinomiais.
- Problemas considerados *tratáveis* ou *fáceis*.
- Mesmo para N^{10}, onde N é o tamanho da entrada?

Algoritmos e problema polinomiais.

n	n^{10}	2^n
0	0	1
10	10000000000	1024
20	10240000000000	1048576
30	590490000000000	1073741824
40	1.04858E+16	1099511627776
50	9.76563E+16	1.1259E+15
60	6.04662E+17	1.15292E+18



Algoritmos e problema polinomiais.

- Exemplo: Equação inteira do segundo grau
  - Dados números inteiros a, b e c, encontrar um número inteiro x tal que $ax^2 + bx + c = 0$.
  - Ou verificar que x não existe.

Algoritmos e problema polinomiais.

- Exemplo:Equação inteira do segundo grau
  - ⇨ Temos que calcular (a * x* x) +(b* x) + c e comparar com zero.
  - ⇨ Se x inteiro satisfaz $ax^2 + bx + c = 0$, os cálculos consomem tempo limitado pelo polinômio do segundo grau com coeficientes a,b e c.
  - ⇨ Logo, trata-se de um problema polinomial.

Algoritmos e problema polinomiais.

- Exemplo:Fatoração
  - Seja $n \in \mathbb{N}$, calcular $p \in \mathbb{N}$ maior que 1 e menor que n com p divisor de n.
  - Ou verificar que p não existe.

Algoritmos e problema polinomiais.

- Exemplo:Fatoração
  - ⇨ Se p fatora n, então $n=k.p$ com $k \in \mathbb{N}$ e $k>1$.
  - ⇨ Logo, $p=n/k < n$.
  - ⇨ Os cálculos para p gastam um tempo polinomial em n.
  - ⇨ Logo, trata-se de outro problema polinomial.

Problemas difícies

- Algoritmos não polinomiais são considerados **intratáveis**.
- Eles são lentos com tempo na ordem de 1.5^N , 2^N, 3^N, 10^N.

Problemas difícies

- Exemplo: Subset sum (soma de subconjuntos)
  - Sejam os números naturais $p_1,\ldots, p_n$ e c.
  - Encontrar um subconjunto K de {1,…, n} tal que a soma de cada p_k para $k \in K$ seja igual ao valor de c.
  - Ou verificar que um tal subconjunto não existe.

Problemas difícies

- Exemplo: Subset sum (soma de subconjuntos)
  - ⇒ Um algoritmo ingênuo avaliaria todos os subconjuntos até encontrar aquele que atendesse a restrição do problema
  - ⇒ Ou, após avaliar todos os subconjuntos, retornaria a não existência de um que atenda a restrição do problema.
  - ⇒ Pior caso: ordem de $N.2^N$……Pq???

Problemas difícies

- Um problema é **polinomialmente verificável** se uma solução de uma instância do problema é de fato ou não uma solução.
  - Verifica-se a existência de um algoritmo que responda sim ou não a questão anterior.
  - A resposta sim deve ser fornecida dentro de um tempo limitado por um polinômio em função do tamanho da instância.
  - **A classe de problemas NP é constituída por problemas polinomialmente verificáveis**.

Problemas difícies

- NP **NÃO** significa não polinomial.
- NP significa ***nondeterministic polynomial***.
- P $\subset$ NP ou P=NP?
  - Até o momento, algum problema em NP pode não estar em P.

Problemas difíceis

- Sejam A e B problemas. Dizemos que o problema A não é mais difícil que o problema B, se A for um subproblema de B.
- Exemplos: Quadrado perfeito
  - Dado um número natural n, encontrar um número natural x tal que $x^2 = n$ ou provar que não existe x.
  - O quadrado perfeito não é mais difícil que o problema da equação inteira do segundo grau.
  - Provar isso!!!

Problemas difíceis

- **Classe NP- difícil** é formada por problemas tão difíceis quanto os problemas mais difíceis em NP.
- Um problema é **NP-difícil** quando todos os problemas em NP não são mais difíceis que ele.
- Um problema NP-difícil que esteja em NP é dito NP-completo.

Problemas difíceis

- Steven Cook e Leonid Levin provaram de forma independente a existência de problemas NP-completos.
- **Redução polinomial:** Sejam A e B problemas. Se A é NP-completo e B não é mais difícil que A, então B também é NP-completo.
- Prova-se que P = NP quando for encontrado um algoritmo polinomial para um único problema NP-completo. Isso ainda não foi provado!!

Eficiência

- Diferentes algoritmos que solucionam um mesmo problema podem apresentar uma diferença significativa em termos de eficiência.
- Tais diferenças podem ser mais impactantes que diferenças de hardware e software.

Eficiência

- Exemplo: Ordenar 10^6 números.
  - O algoritmo **insertion sort** leva um tempo na ordem de $c_1 n^2$ para ordenar n itens, onde a constante c_1 não depende de n.
  - O algoritmo **merge sort** leva um tempo na ordem de $c_2 n.\lg n$, onde a constante c_2 não depende de n.

Eficiência

- Exemplo: Ordenar 10^6 números.
  - Suponha que o computador A execute 10 bilhões de tarefas por segundos.
  - Um excelente programador implementa o algoritmo insertion sort em linguagem de máquina no computador A conseguindo performance de $2n^2$.

Eficiência

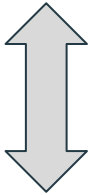
- Exemplo: Ordenar 10^6 números.
  - Suponha que o computador B execute 10 milhões de tarefas por segundos.
  - Um programador com nível mediano implementa o algoritmo merge sort (em linguagem C usando o DEV!!). Ele consegue uma performance de 50nlogn.
  - O computador A é 1000 vezes mais rápido do que o computador B.

Eficiência

- Exemplo: Ordenar 10^6 números.

Ordenar 100 Milhões de números:
A: 23d
B: 4h

$$A: \frac{2(10^7)}{10^{16}} = 20000seg(> 5.5h)$$



17 vezes superior

$$B: \frac{50(10^7)lg(10^7)}{10^7} = 1163seg(< 20min)$$

Eficiência

- A eficiência não recai sobre o tipo de linguagem utilizada para implementar o algoritmo, nem sobre o tipo de máquina na qual o algoritmo é executado.
- A eficiência precisa ser avaliada através de um critério adequado para comparar algoritmos.
- Esse critério deve ser independente da linguagem e tipo de máquina.

Eficiência

- Nesse contexto, a **eficiência** de um algoritmo pode ser definida como sua **complexidade de tempo**.
- A **complexidade de tempo** é dada pelo número de **instruções básicas** que ele executa considerando o tamanho da entrada.
- Observa-se que o **pior caso** possível é considerado na determinação do tempo máximo para solucionar uma instância de grande porte.
- Trata-se de uma **análise assintótica**, ou seja, avalia-se o comportamento do algoritmo para entradas de grande porte.

Eficiência

- Exemplo: Algoritmo que gasta um tempo limitado por um polinômio da forma:

$$p(n)=2n^5+200n^4+2000n^2+$$

$$+20000n+200000$$

n	2n^5	+200n^4	+2000n^2	+20000n	200000	Total
1	2	200	2000	20000	200000	222202
10	200000	2000000	200000	200000	200000	2800000
100	2,00E+10	2,00E+10	2,00E+07	2,00E+06	2,00E+05	4,00E+10
1000	**2,00E+15**	2,00E+14	2,00E+09	2,00E+07	2,00E+05	**2,20E+15**
10000	**2,00E+20**	2,00E+18	2,00E+11	2,00E+08	2,00E+05	**2,02E+20**
100000	**2,00E+25**	2,00E+22	2,00E+13	2,00E+09	2,00E+05	**2,00E+25**
1000000	**2,00E+30**	2,00E+26	2,00E+15	2,00E+10	2,00E+05	**2,00E+30**
10000000	**2,00E+35**	2,00E+30	2,00E+17	2,00E+11	2,00E+05	**2,00E+35**

RAM Model

- Random-access machine (RAM) model
  - Permite uma adequada comparação entre algoritmos diferentes aplicados a um mesmo problema.
  - Facilita a avaliação do comportamento dos algoritmos em instâncias de grande porte.

RAM Model

- Random-access machine (RAM) model
  - Sequencial:
    - A instruções são executadas uma após a outra , ou seja, sem operações simultâneas.
  - Há um único processador.

RAM Model

- Random-access machine (RAM) model
  - Tipos de instruções consideradas:
    - aritméticas (adicionar, subtrair, multiplicar, dividir, resto, piso, teto)
    - transferências de dados (carregar, armazenar, copiar)
    - controles (estrutura condiciona, chamada de sub-rotinas e retorno).
  - Cada instrução consome uma quantidade de tempo constante.

RAM Model

- Random-access machine (RAM) model
  - Tipos básicos: inteiros e reais.
  - Um limite no tamanho de cada palavra de dados é assumido.
    - Por exemplo, assumimos que inteiros são representados por clgn bits para alguma constante c≥1 e entradas de tamanho n.
    - A constante c≥ 1 permite cada elemento da entrada individualmente.
    - O valor constante para c impede que o tamanho da palavra cresça arbitrariamente.

RAM Model

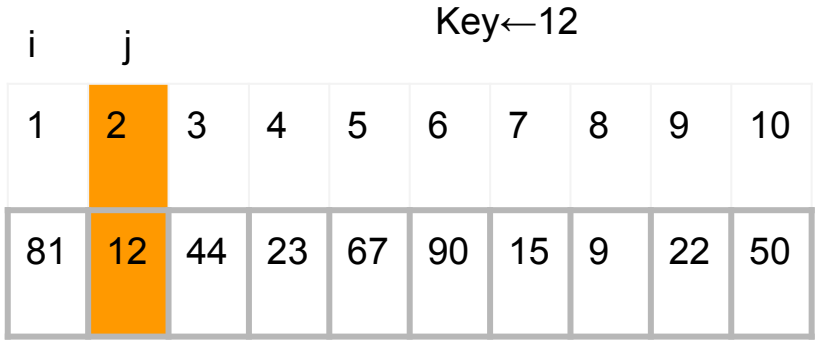
- Desvantagens
  - Análise focada no pior caso que pode não ocorrer frequentemente.
  - Não permite análise do comportamento no caso médio

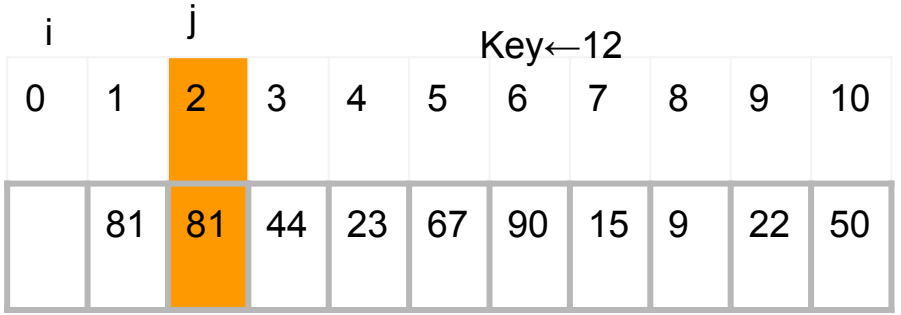
Analisando Algoritmos

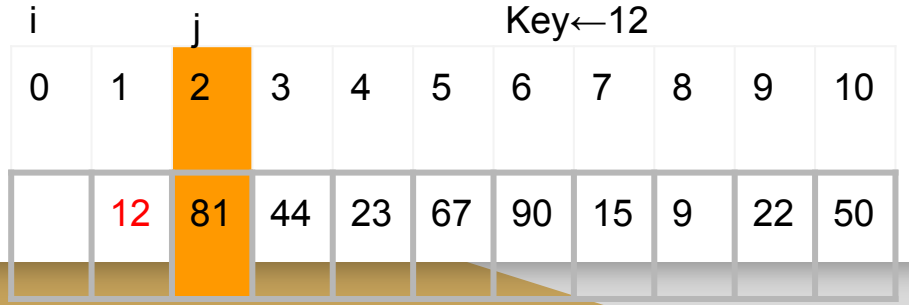
- Exemplo: Problema de Ordenação
  - Input: Uma sequência de n números:$\{a_1\ a_2, a_3, \ldots., a_n\}$
  - Output: Uma reordenação da sequência em $\{a'_1, a'_2, a'_3 \ldots a'_n\}$ tal que $a'_1 \leq a'_2 \leq a'_3 \leq \ldots \leq a'_n$

Analisando Algoritmos

```
INSERTION-SORT (A, n)
for j ←2 to n
        do key ← A[ j]
                i ← j –1
                while i > 0 and A[i] > key
                        do A[i+1] ← A[i]
                        i ← i –1
                A[i+1] = key
```

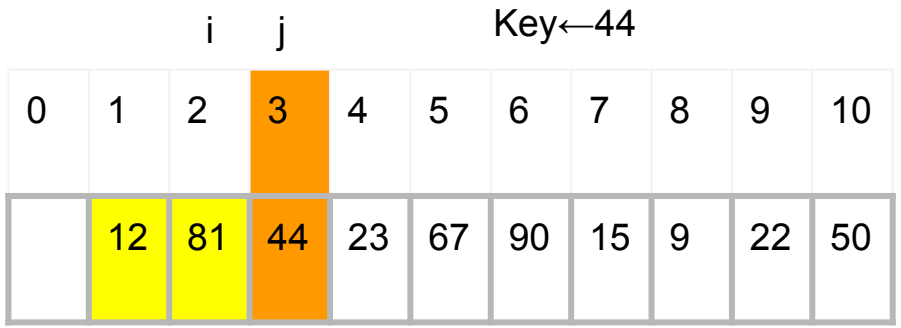


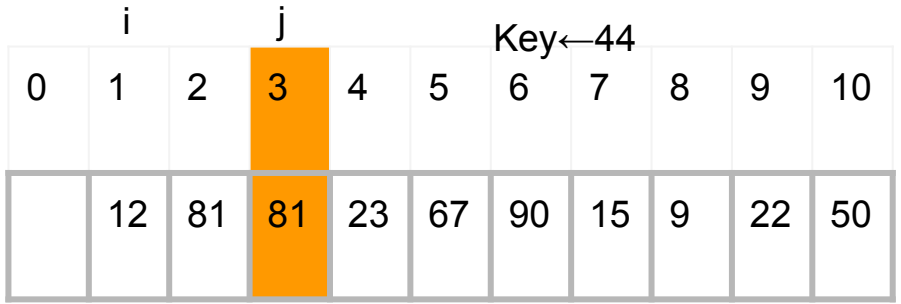


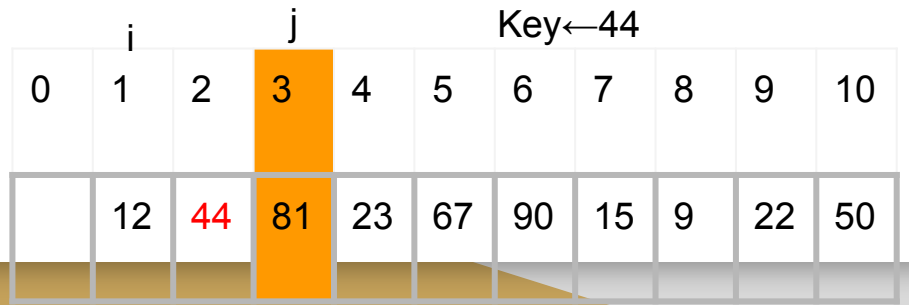


Analisando Algoritmos

```
INSERTION-SORT (A, n)
for j ←2 to n
        do key ← A[ j]
                i ← j –1
                while i > 0 and A[i] > key
                        do A[i+1] ← A[i]
                        i ← i –1
                A[i+1] = key
```

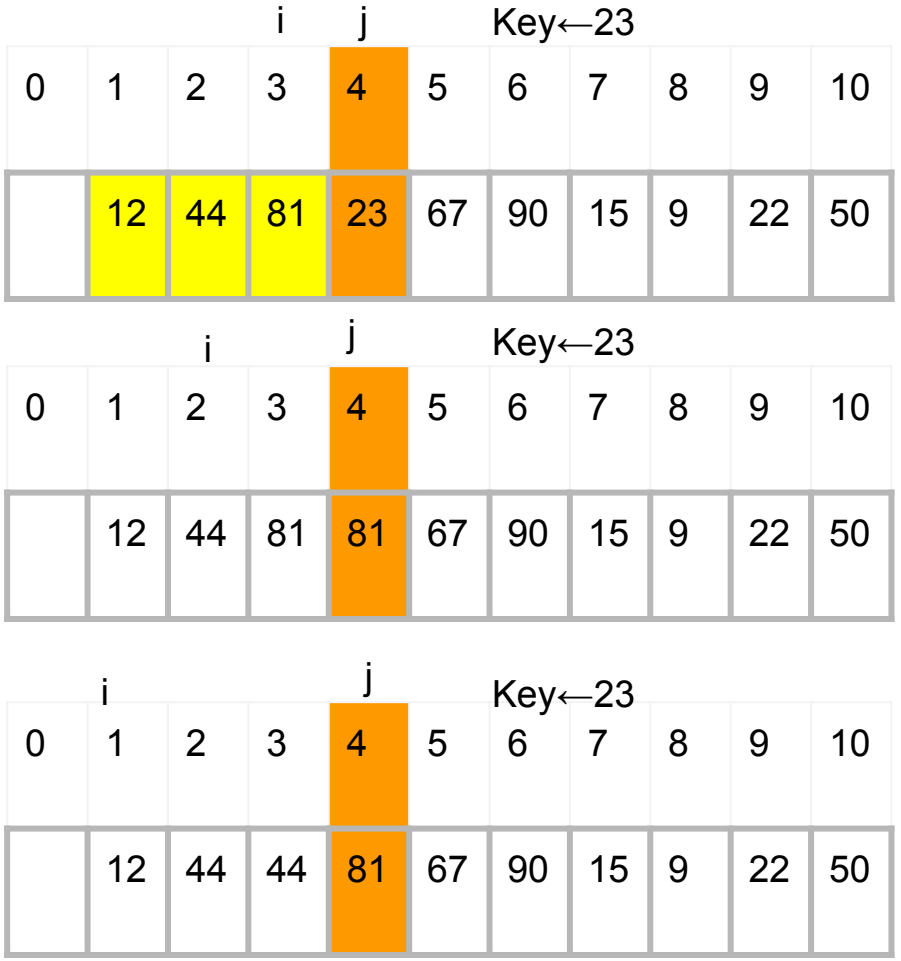






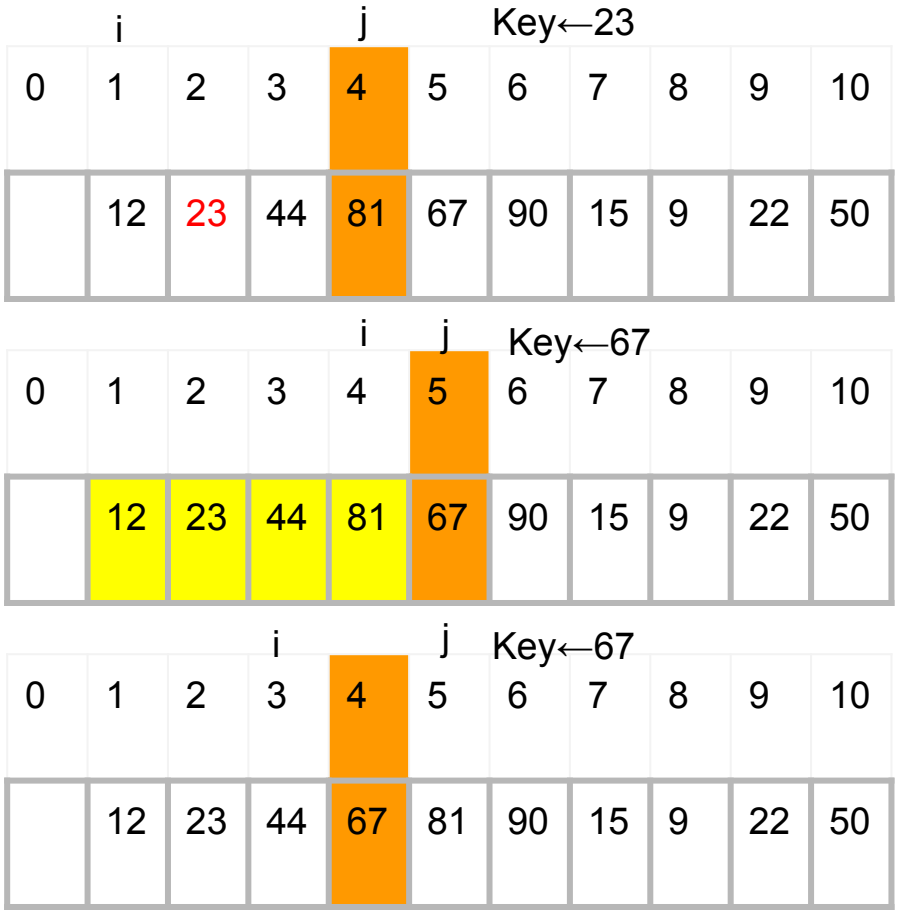
Analisando Algoritmos

```
INSERTION-SORT (A, n)
for j ←2 to n
        do key ← A[ j]
                i ← j –1
                while i > 0 and A[i] > key
                        do A[i+1] ← A[i]
                        i ← i –1
                A[i+1] = key
```



Analisando Algoritmos

```
INSERTION-SORT (A, n)
for j ←2 to n
        do key ← A[ j]
                i ← j –1
                while i > 0 and A[i] > key
                        do A[i+1] ← A[i]
                        i ← i –1
                A[i+1] = key
```



Analisando Algoritmos

```
INSERTION-SORT (A, n)
1. for j ←2 to n
2.      do key ← A[ j]
3.              i ← j –1
4.              while i > 0 and A[i] > key
5.                      do A[i+1] ← A[i]
6.                              i ← i –1
7.              A[i+1] = key
```

- Complexidade de tempo avaliada pela contagem do número de instruções básicas executadas para uma entrada de tamanho n.
- Seja c_m um valor constante que indica o custo em termos de tempo para cada execução das instruções na linha m .
- Seja t_j a quantidade de vezes que as condições dentro do comando de repetição while foram verificadas na linha 4 para cada valor de j atribuído na linha 1.

Analisando Algoritmos

INSERTION-SORT (A, n)	Custo	Número de execuções
1.for j ←2 to n	c1	n
2. do key ← A[j]	c2	n-1
3. i ← j –1	c3	n-1
4. while i > 0 and A[i] > key	c4	$\sum_{j=2}^{n} t_j$
5. do A[i+1] ← A[i]	c5	$\sum_{j=2}^{n} (t_j - 1)$
6. i ← i –1	c6	$\sum_{j=2}^{n} (t_j - 1)$
7. A[i+1] = key	c7	n-1

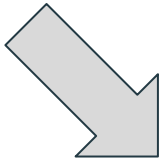
Tempo total de execução para entrada de tamanho n:

$$T(n) = c_1 n + c2(n-1) + c3(n-1) + \\ + c4 \sum_{j=2}^{n} t_j + c5 \sum_{j=2}^{n} (t_j - 1) + c6 \sum_{j=2}^{n} (t_j - 1) + \\ c7(n-1)$$

Analisando Algoritmos

- **Tempo total de execução para o melhor caso - Vetor já ordenado**
  - Tj = 1 pois a condição na linha 4 nunca será satisfeita.

$$T(n) = c_1 n + c2(n-1) + c3(n-1) + \\ + c4\sum_{j=2}^{n} t_j + c5\sum_{j=2}^{n}(t_j - 1) + c6\sum_{j=2}^{n}(t_j - 1) + \\ c7(n-1)$$



$$t_j = 1$$

$$\sum_{j=2}^{n} t_j = n - 1$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$
$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c + 7)$$
$$T(n) = an + b$$

Analisando Algoritmos

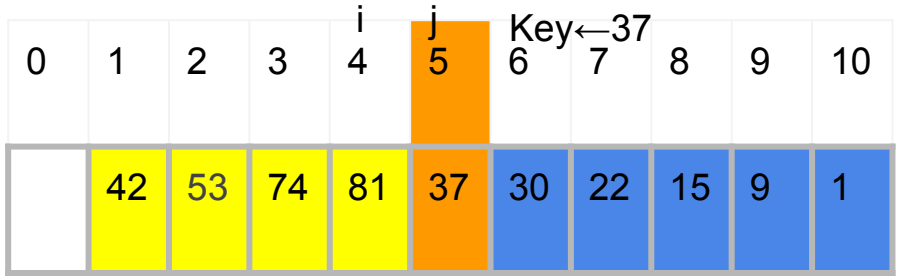
- **Tempo total de execução para o pior caso - Vetor ordenado em ordem oposta ao ordenamento proposto.**

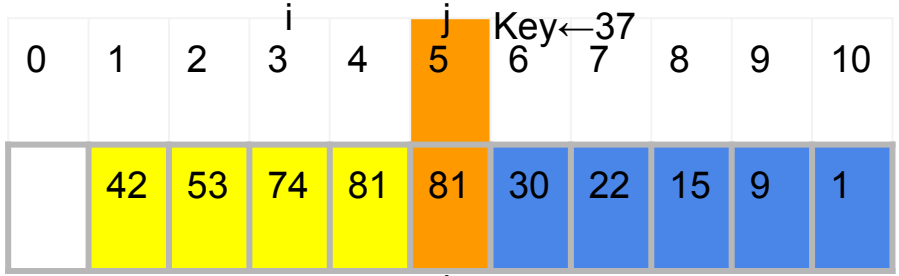
INSERTION-SORT (A, n)

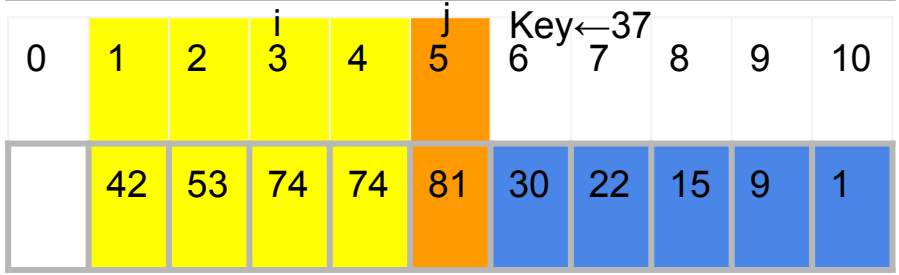
...

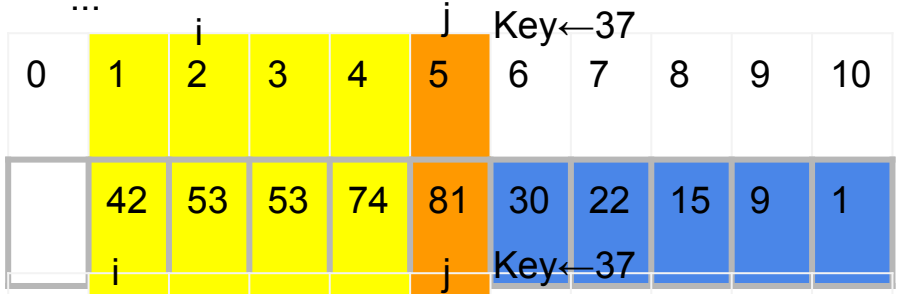
do A[i+1] ← A[i]
i ← i –1

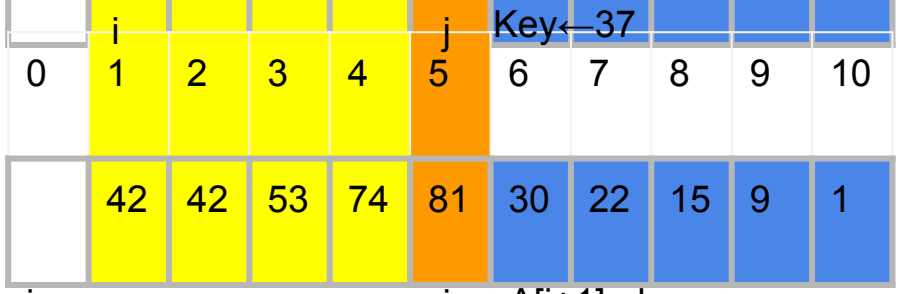
...











Analisando Algoritmos

- **Tempo total de execução para o pior caso - Vetor ordenado em ordem oposta ao ordenamento proposto.**

$$T(n) = c_1 n + c2(n-1) + c3(n-1) + \\ + c4\sum_{j=2}^{n} t_j + c5\sum_{j=2}^{n}(t_j - 1) + c6\sum_{j=2}^{n}(t_j - 1) + \\ c7(n-1)$$

$$t_j = j$$

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = 2 + 3 + 4 + 5 + 6 ... n = \frac{n(n+1)}{2}$$ -1

$$\sum_{j=2}^{n}(t_j - 1) = \sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) +$$

$$+ c_4\frac{n(n+1))}{2} + +c_5\frac{n(n-1)}{2} + c_6\frac{n(n-1)}{2} + c_7(n-1)$$

$$T(n) = (\frac{c_4 + c_5 + c_6}{2})n^2 + [c_1 + c_2 + c_3 + c_7 + \frac{(c_4 - c_5 - c_6)}{2}]n$$

$$- (c_2 + c_3 + c_7)$$