Algoritmos

Programação Avançada
Universidade Federal de Pernambuco
Adrien Durand-Petiteville
adrien.durandpetiteville@ufpe.br

Análise de Algoritmos

Por que análise de desempenho?

- Há muitas coisas importantes que devem ser cuidadas, como facilidade de uso, modularidade, segurança, capacidade de manutenção, etc.
- Só podemos ter todas as coisas acima se tivermos desempenho.
- Portanto, o desempenho é como uma moeda através da qual podemos comprar todas as coisas acima.
- Para resumir, desempenho $==$ escala.
- Imagine um editor de texto que possa carregar 1000 páginas, mas que consiga soletrar verificação 1 página por minuto.

Considerando dois algoritmos para uma tarefa, como descobrirmos qual é melhor?

- Uma maneira ingênua de fazer isso é:
  Implementar os dois algoritmos e executar os dois programas em seu computador para diferentes entradas e ver qual deles leva menos tempo.
- Há muitos problemas com esta abordagem para a análise dos algoritmos.
  - 1) É possível que para algumas entradas, o primeiro algoritmo tenha um desempenho melhor do que o segundo. E para alguns inputs, o segundo tem melhor desempenho.
  - 2) Também pode ser possível que, para algumas entradas, o primeiro algoritmo tenha um melhor desempenho em uma máquina e o segundo funciona melhor em outra máquina para algumas outras entradas.

A Análise Assimptótica

- A Análise Assimptótica é a grande ideia que trata das questões acima na análise de algoritmos.
- Na Análise Assimptótica, avaliamos o desempenho de um algoritmo **em termos de tamanho de entrada** (não medimos o tempo real de execução).
- Calculamos, como o tempo (ou espaço) tomado por um algoritmo aumenta com o tamanho da entrada.

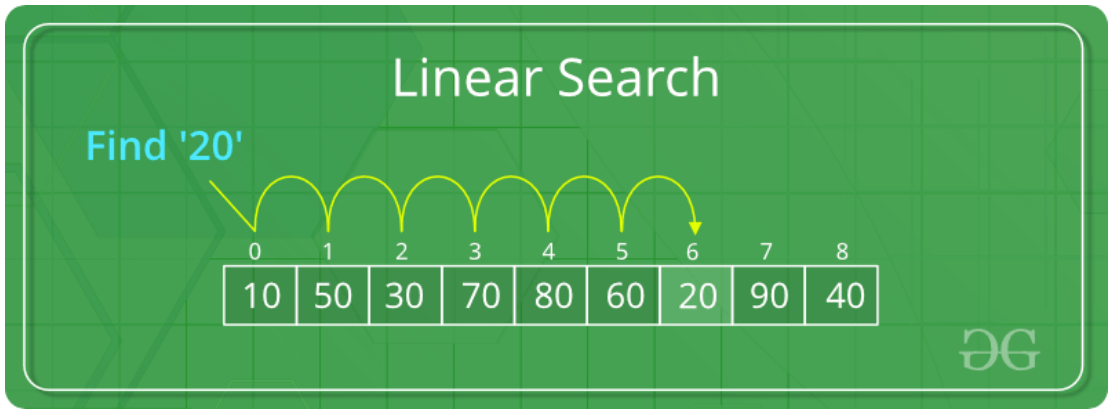
O Problema da Busca

- Dada uma chave de busca e uma coleção de elementos, onde cada elemento possui um identificador único, desejamos encontrar o elemento da coleção que possui o identificador igual ao da chave de busca ou verificar que não existe nenhum elemento na coleção com a chave fornecida.

```
Input : arr[] = {10, 20, 80, 30, 60, 50,
                 110, 100, 130, 170}
          x = 110;
Output : 6
Element x is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50,
                 110, 100, 130, 170}
           x = 175;
Output : -1
Element x is not present in arr[].
```

Busca sequencial

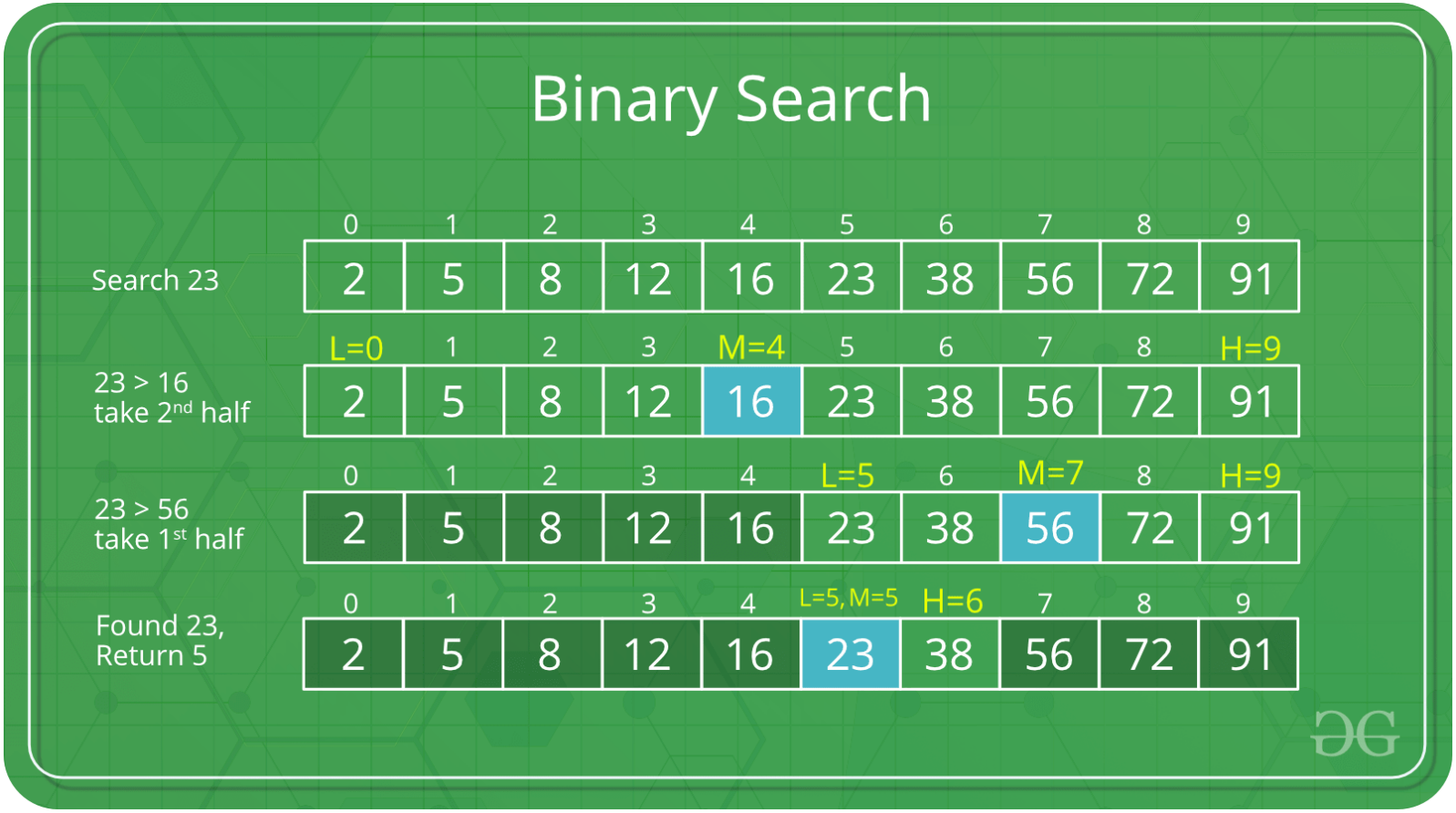
- É o método de pesquisa mais simples que existe.
- Funcionamento:
  - a partir do primeiro registro, pesquise sequencialmente até encontrar o valor procurado ou até chegar ao fim do vetor;
  - então pare.



Busca binária

- Método de busca eficiente para um **conjunto ordenado**.
- Compare o elemento procurado com o elemento que está na posição do meio do vetor:
  - Se o elemento foi encontrado, termine a busca.
  - Se o elemento procurado é menor do que o elemento do meio, repita a busca para a primeira metade do vetor.
  - Se o elemento procurado é maior do que o elemento do meio, repita a busca para a segunda metade do vetor.
- Esse processo se repete até que o elemento seja encontrado ou até que todo vetor tenha sido consultado sem sucesso

Busca binária - 2



A Análise Assimptótica - Exemplo

- Por exemplo, consideremos o problema de busca em uma matriz ordenada.
- Uma maneira de pesquisar é Linear Search e a outra maneira é Binary Search.
- Para entender como a Análise Assimptótica resolve os problemas acima mencionados na análise de algoritmos, digamos que executamos a Busca Linear em um computador rápido `A` e a Busca Binária em um computador lento `B` e escolhemos os valores constantes para os dois computadores de modo que nos diga exatamente quanto tempo leva para que a máquina em questão realize a busca em segundos.
- Digamos que a constante para `A` é 0,2 e a constante para `B` é 1000, o que significa que `A` é 5000 vezes mais potente que `B`.

A Análise Assimptótica - Exemplo - 2

- Para valores pequenos de matriz de entrada tamanho `n`, o computador rápido pode levar menos tempo.
- Mas, após um certo valor do tamanho da matriz de entrada, a Busca Binária definitivamente começará a demorar menos tempo em comparação com a Busca Linear, embora a Busca Binária esteja sendo executada em uma máquina lenta.
- O motivo é a ordem de crescimento da Busca Binária em relação ao tamanho de entrada é **logarítmica** enquanto a ordem de crescimento da Busca Linear é **linear**.
- Assim, as constantes dependentes da máquina podem sempre ser ignoradas após um certo valor de tamanho de entrada.

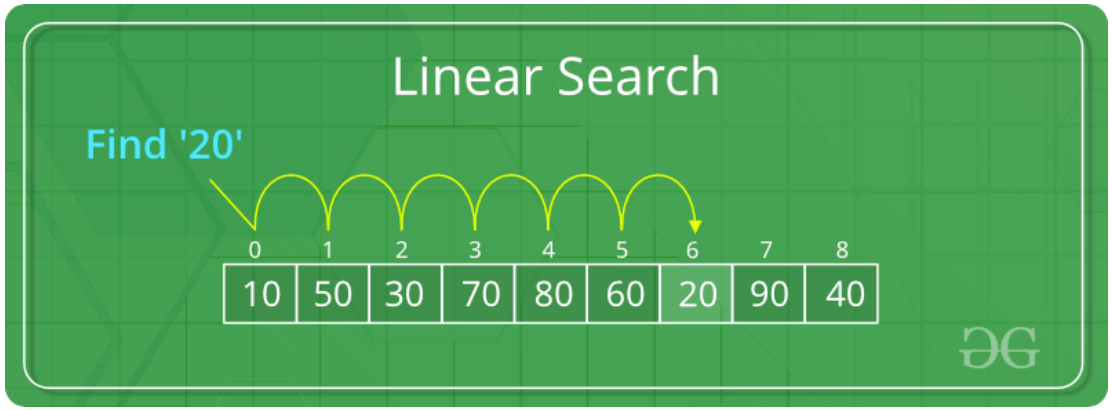
A Análise Assimptótica - Exemplo - 3

- Tempo de execução de Busca Sequencial em segundos em A: $0.2 * n$
- Tempo de execução de Busca Binária em segundos em B: $1000 * \log n$

n	Running time on A	Running time on B
10	2 sec	~ 1 h
100	20 sec	~ 1.8 h
10^6	~ 55.5 h	~ 5.5 h
10^9	~ 6.3 years	~ 8.3 h

Casos da Análise Assimptótica

- Há casos para analisar um algoritmo:
  - O pior caso
  - Caso médio
  - Melhor caso
- Usamos um exemplo de busca sequencial e analisá-lo usando a análise assimptótica.



Análise de Melhor Caso (Bogus)

- Na melhor análise de casos, calculamos o menor limite no tempo de execução de um algoritmo.
- Devemos conhecer o caso que causa o número mínimo de operações a serem executadas.
- No problema da busca linear, o melhor caso ocorre quando o elemento está presente no primeiro local.
- O número de operações no melhor caso é constante (não depende de n).
- Portanto, a complexidade de tempo no melhor caso seria $\Theta(1)$

Análise média de casos (às vezes feita)

- Na análise de casos médios, tomamos todas as entradas possíveis e calculamos o tempo de computação para todas as entradas.
- Somar todos os valores calculados e dividir a soma pelo número total de entradas.
- Devemos conhecer (ou prever) a distribuição dos casos.
- Para o problema da busca sequencial, vamos supor que todos os casos estão uniformemente distribuídos (incluindo o caso do elemento não estar presente na matriz).
- Portanto, somamos todos os casos e dividimos a soma por $(n + 1)$. A seguir está o valor da complexidade média dos casos.

Análise dos piores casos (Normalmente Feito)

- Na análise dos piores casos, calculamos o limite superior no tempo de execução de um algoritmo.
- Devemos conhecer o caso que causa o número máximo de operações a serem executadas.
- Para a busca sequencial, o pior caso acontece quando o elemento a ser pesquisado não está presente na matriz.
- Quando o elemento não está presente, as funções de busca o comparam com todos os elementos do conjunto um por um.
- Portanto, o pior caso de complexidade temporal da busca linear seria $\theta(n)$.

Análise Assimptótica de Casos

- Na maioria das vezes, fazemos a análise dos piores caso para analisar algoritmos.
- Na análise dos piores casos, garantimos um limite superior no tempo de execução de um algoritmo que é uma boa informação.
- A análise média de casos não é fácil de fazer na maioria dos casos práticos e raramente é feita.
- Na análise de casos médios, devemos conhecer (ou prever) a distribuição matemática de todos os inputs possíveis.
- A análise dos melhores casos é falsa.
- Garantir um limite inferior em um algoritmo não fornece nenhuma informação, pois no pior caso, um algoritmo pode levar anos para ser executado.

Notações assimptóticas

- A ideia principal da análise assimptótica é ter uma medida da eficiência dos algoritmos que não dependem de constantes específicas da máquina e não requerem que os algoritmos sejam implementados e o tempo gasto pelos programas para serem comparados.
- Notações assimptóticas são ferramentas matemáticas para representar a complexidade do tempo dos algoritmos para análise assimptótica.
- As 3 notações assimptóticas a seguir são usadas principalmente para representar a complexidade de tempo dos algoritmos.

A notação Theta

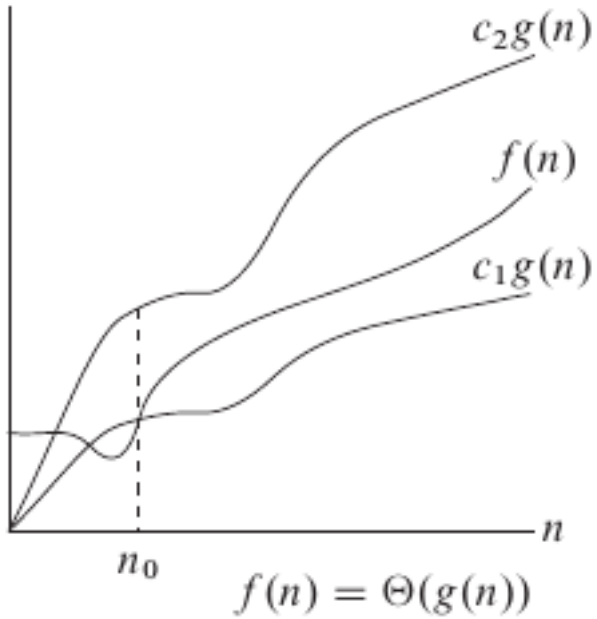
- A notação theta limita uma função por cima e por baixo, de modo que define um comportamento assimptótico exato.
- Uma maneira simples de obter a notação Theta de uma expressão é deixar cair os termos de ordem baixa e ignorar as constantes principais.
- Por exemplo, considere a seguinte expressão.

$$3n^3 + 6n^2 + 6000 = \Theta(n^3)$$

- Abandonar termos de ordem inferior é sempre bom porque sempre haverá um número (n) após o qual $\Theta(n^3)$ tem valores mais altos que $\Theta(n^2)$, independentemente das constantes envolvidas.
- Se $f(n)$ é Theta de $g(n)$, então o valor $f(n)$ está sempre entre $c1 * g(n)$ e $c2 * g(n)$ para grandes valores de n ($n \geq n_0$).

A notação Theta - 2

- Se $f(n)$ é Theta de $g(n)$, então o valor $f(n)$ está sempre entre $c1 * g(n)$ e $c2 * g(n)$ para grandes valores de n ($n \geq n_0$).

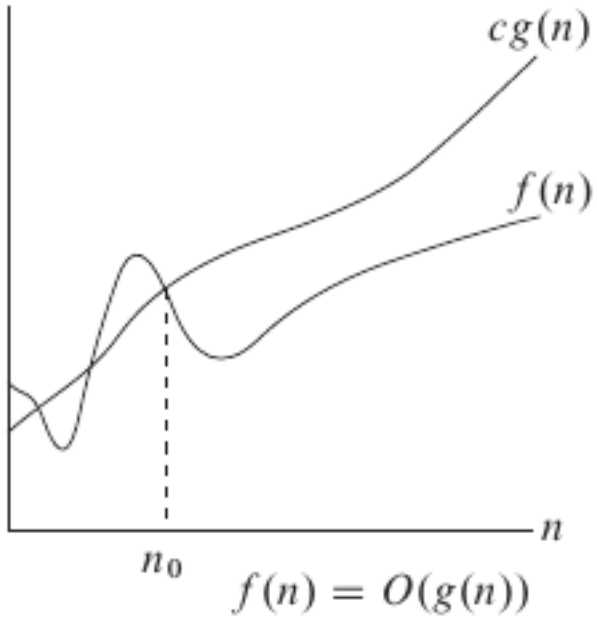


A notação Grande O

- A notação Grande O (Big O) define um limite superior de um algoritmo, ela limita uma função apenas de cima.
- Por exemplo, considere o caso de `Insertion Sort`: leva tempo linear na melhor das hipóteses e tempo quadrático na pior das hipóteses.
- Podemos dizer com segurança que a complexidade de tempo do tipo Inserção é $O(n^2)$.
- Note que $O(n^2)$ também cobre o tempo linear.
- Se usarmos a notação Theta para representar a complexidade de tempo de `Insertion Sort`, temos que usar duas afirmações para os melhores e os piores casos:
  - O pior caso de complexidade de tempo é $\Theta(n^2)$.
  - A melhor complexidade de tempo é $\Theta(n)$.
- A notação Grande O é útil quando temos apenas o limite superior de complexidade de tempo de um algoritmo.

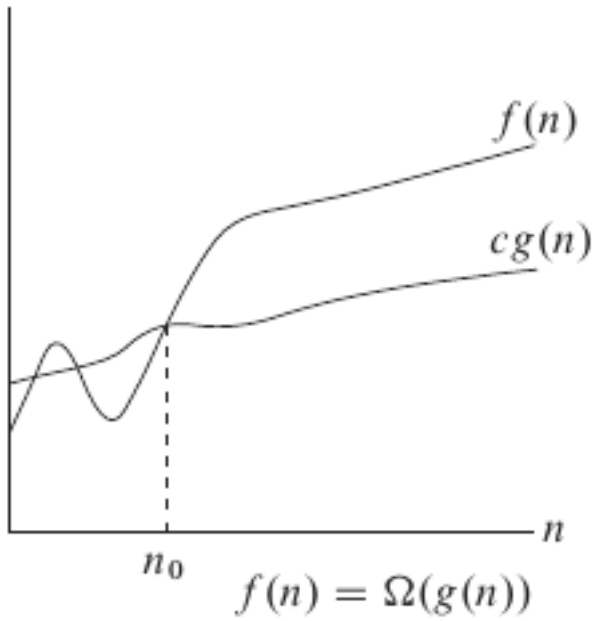
A notação Grande O - 2

- A notação Grande O (Big O) define um limite superior de um algoritmo, ela limita uma função apenas de cima.



A notação Ω

- Assim como a notação Grande O fornece um limite superior assimptótico sobre uma função, a notação Ω fornece um limite inferior assimptótico.
- A notação pode ser útil quando temos um limite inferior de complexidade temporal de um algoritmo.
- A notação Ω é a notação menos utilizada entre todas as três.



Análise de laços: $\mathbf{O}(1)$

- A complexidade temporal de uma função (ou conjunto de declarações) é considerada como $\mathbf{O}(1)$ se ela não contiver laço, recursividade e chamada para qualquer outra função temporal não constante.
- Um laço ou recursividade com um número constante de iterações também é considerado como $\mathbf{O}(1)$.
- Por exemplo, o seguinte laço é $\mathbf{O}(1)$.

```
// Here c is a constant
for (int i = 1; i <= c; i++) {
     // some O(1) expressions
}
```

Análise de laços: $\mathbf{O}(n)$

- A complexidade temporal de um laço é considerada como $\mathbf{O}(n)$ se as variáveis do laço forem incrementadas / decrescidas por uma quantidade constante.
- Por exemplo, as seguintes funções têm $\mathbf{O}(n)$ complexidade de tempo.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
     // some O(1) expressions
}

for (int i = n; i > 0; i -= c) {
     // some O(1) expressions
}
```

Análise de laços: $\mathbf{O}(n^c)$

- A complexidade temporal dos laços aninhados é igual ao número de vezes que a declaração mais interna é executada.
- Por exemplo, as seguintes funções de laços têm $\mathbf{O}(n^2)$ complexidade de tempo.

```
for (int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
       // some O(1) expressions
    }
}

for (int i = n; i > 0; i -= c) {
    for (int j = i+1; j <=n; j += c) {
       // some O(1) expressions
}
```

Análise de laços: $\mathbf{O}(\log n)$

- A complexidade temporal de um laço é considerada como $\mathbf{O}(\log n)$ se as variáveis do laço forem divididas / multiplicadas por uma quantidade constante.

```
for (int i = 1; i <=n; i *= c) {
    // some O(1) expressions
}
for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
```

Análise de laços: $\mathbf{O}(\log \log n)$

- A complexidade temporal de um laço é considerada como $\mathbf{O}(\log \log n)$ se as variáveis do laço forem reduzidas / aumentadas exponencialmente em uma quantidade constante.

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {
    // some O(1) expressions
}
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 1; i = fun(i)) {
    // some O(1) expressions
}
```

Combinar as complexidades temporais de laços consecutivos

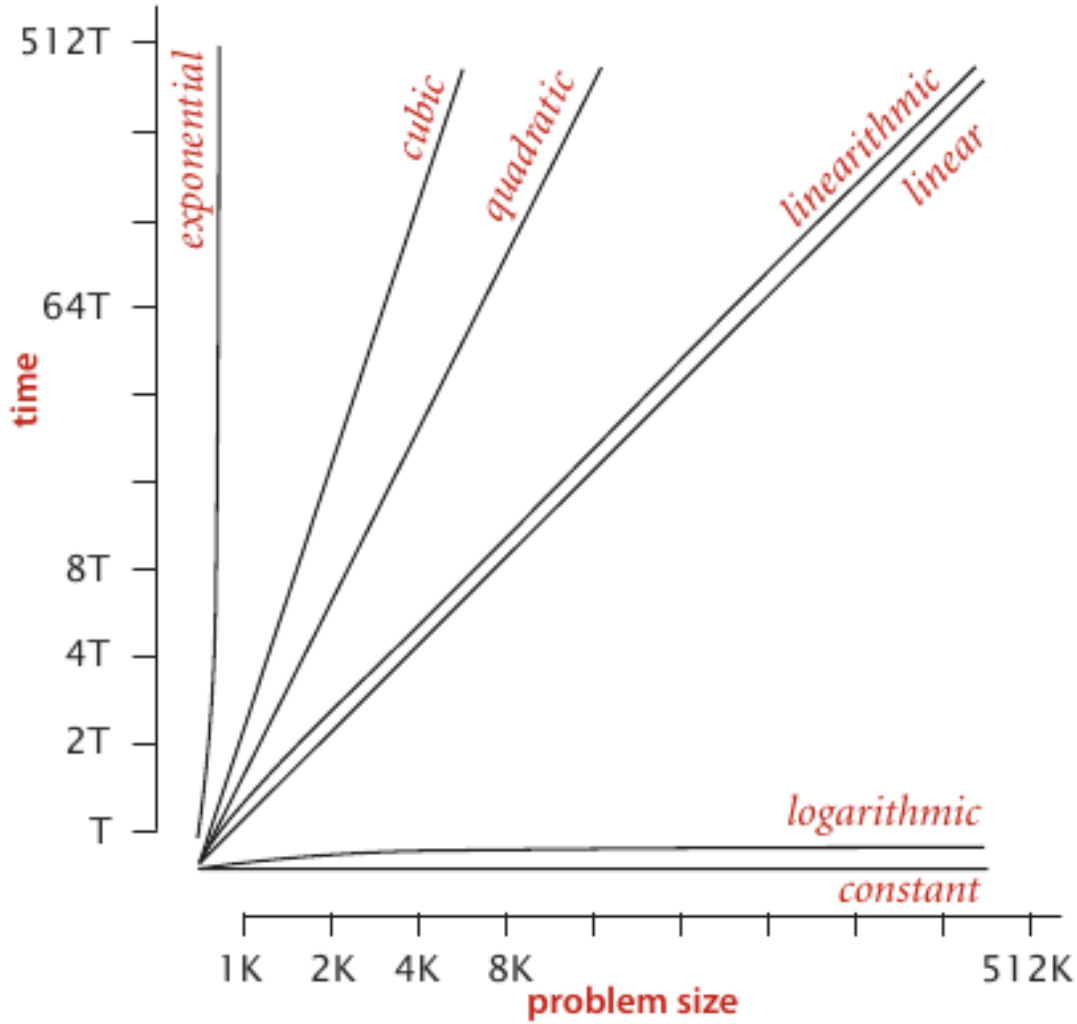
- Quando existem laços consecutivos, calculamos a complexidade do tempo como soma das complexidades de tempo de laços individuais.

```
for (int i = 1; i <=m; i += c) {
     // some O(1) expressions
}
for (int i = 1; i <=n; i += c) {
     // some O(1) expressions
}
Time complexity of above code is O(m) + O(n) which is O(m+n)
If m == n, the time complexity becomes O(2n) which is O(n).
```

Calcular a complexidade temporal quando há muitos if-else declarações dentro do laço

- A complexidade do pior caso é a mais útil entre as melhores, médias e piores.
- Portanto, temos que considerar o pior caso.
- Avaliamos a situação quando valores em condições de `if-else` causam o máximo número de declarações a serem executadas.
- Por exemplo, considere a função de busca sequencial onde consideramos o caso quando o elemento está presente no final ou não está presente de todo.
- Quando o código é complexo demais para considerar todos os casos de `if-else`, podemos obter um limite superior ignorando o `if-else` e outras afirmações de controle complexas.

Ordem de crescimento



Estrutura de dados: média de casos

Data structure	Access	Search	Insertion	Deletion
Array	O(1)	O(N)	O(N)	O(N)
Stack	O(N)	O(N)	O(1)	O(1)
Queue	O(N)	O(N)	O(1)	O(1)
Singly Linked list	O(N)	O(N)	O(1)	O(1)
Doubly Linked List	O(N)	O(N)	O(1)	O(1)
Hash Table	O(1)	O(1)	O(1)	O(1)
Binary Search Tree	O(log N)	O(log N)	O(log N)	O(log N)

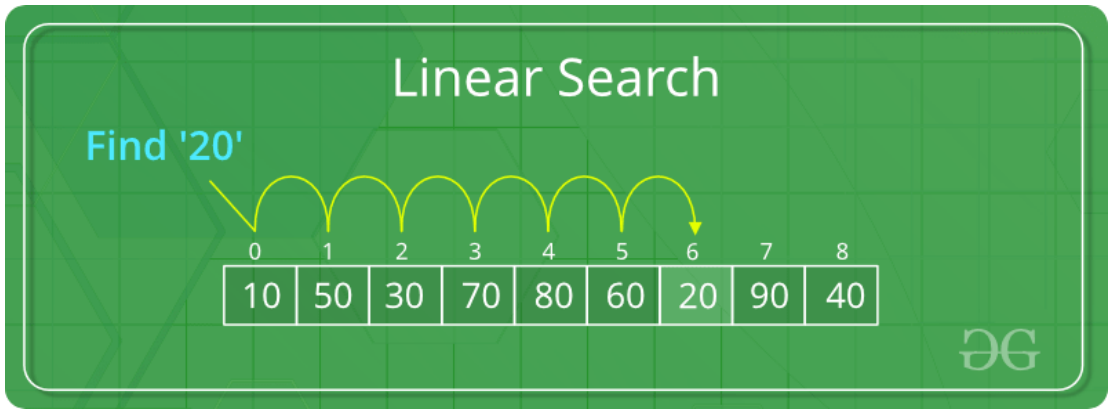
Estrutura de dados: piores casos

Data structure	Access	Search	Insertion	Deletion
Array	O(1)	O(N)	O(N)	O(N)
Stack	O(N)	O(N)	O(1)	O(1)
Queue	O(N)	O(N)	O(1)	O(1)
Singly Linked list	O(N)	O(N)	O(1)	O(1)
Doubly Linked List	O(N)	O(N)	O(1)	O(1)
Hash Table	O(N)	O(N)	O(N)	O(N)
Binary Search Tree	O(N)	O(N)	O(N)	O(N)

Algoritmos de busca

Busca sequencial

- É o método de pesquisa mais simples que existe.
- Funcionamento:
  - a partir do primeiro registro, pesquise sequencialmente até encontrar o valor procurado ou até chegar ao fim do vetor;
  - então pare.
- A complexidade temporal do algoritmo é $\mathbf{O}(n)$.



Implementação

```
include <iostream>
using namespace std;

int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

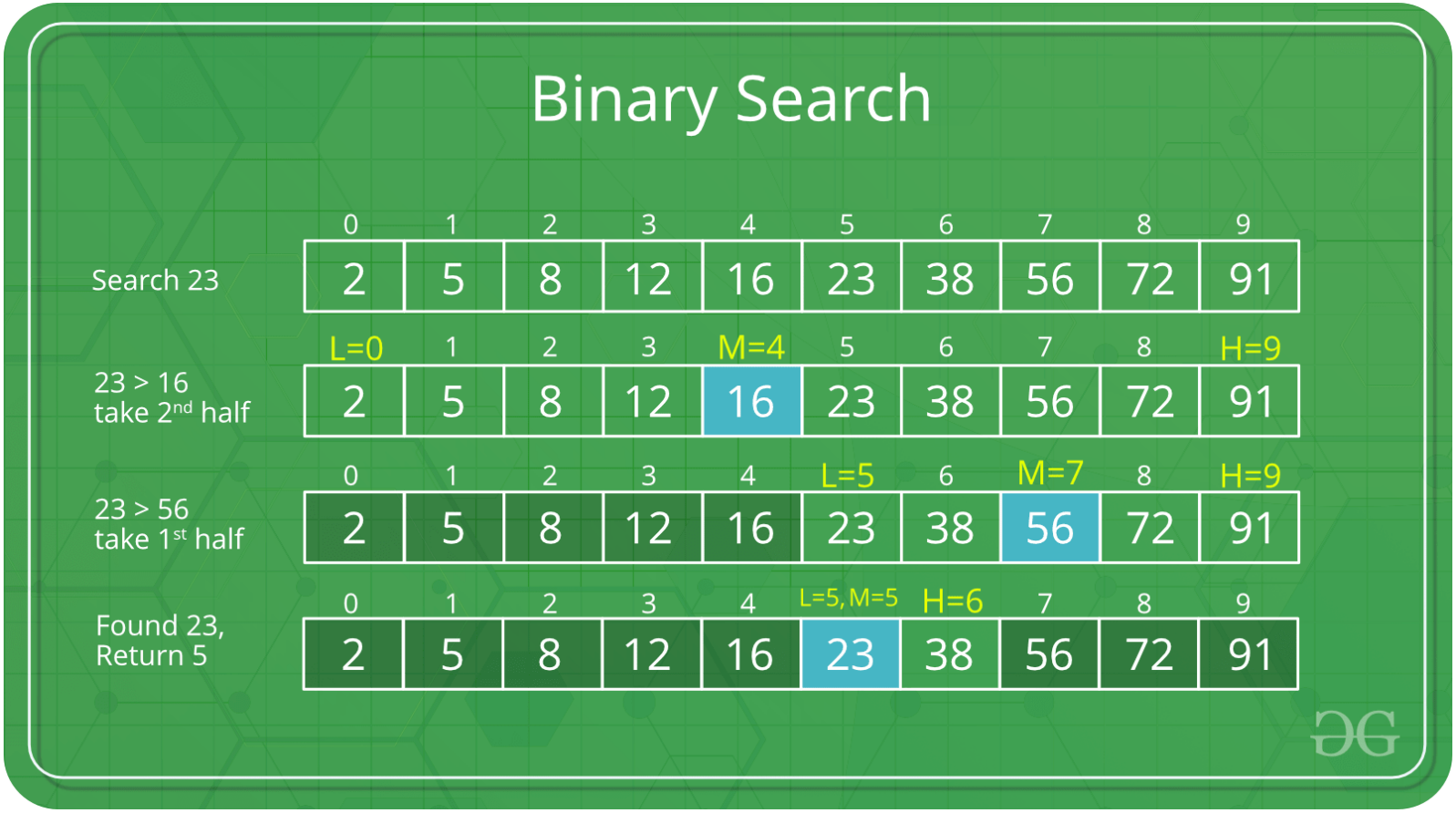
Implementação - 2

```
// Driver code
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function call
    int result = search(arr, n, x);
    (result == -1)
        ? cout << "Element is not present in array"
        : cout << "Element is present at index " << result;
    return 0;
}
```

Busca binária

- Método de busca eficiente para um **conjunto ordenado**.
- Compare o elemento procurado com o elemento que está na posição do meio do vetor:
  - Se o elemento foi encontrado, termine a busca.
  - Se o elemento procurado é menor do que o elemento do meio, repita a busca para a primeira metade do vetor.
  - Se o elemento procurado é maior do que o elemento do meio, repita a busca para a segunda metade do vetor.
- Esse processo se repete até que o elemento seja encontrado ou até que todo vetor tenha sido consultado sem sucesso
- A complexidade temporal do algoritmo é $\mathbf{O}(\log n)$.

Busca binária - 2



Implementação

```
#include <bits/stdc++.h>
using namespace std;

int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then it can only be present
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }
    // We reach here when element is not present in array
    return -1;
}
```

Implementação - 2

```
  int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? cout << "Element is not present in array"
                   : cout << "Element is present at index " << result
    return 0;
}
```

Busca por Salto

- Como a Busca Binária, a Busca por Salto é um algoritmo de busca por conjuntos ordenados.
- A ideia básica é verificar menos elementos (do que a busca sequencial) saltando à frente por passos fixos.
- Por exemplo, suponha que tenhamos um arranjo `arr[]` de tamanho `n` e bloco (a ser pulado) de tamanho `m`.
- Depois pesquisamos nos índices `arr[0]`, `arr[m]`, `arr[2m]`, `...`, `arr[km]` e assim por diante.
- Uma vez encontrado o intervalo (`arr[km] < x < arr[(k+1)m]`), realizamos uma operação de busca sequencial a partir do `km` índice para encontrar o elemento `x`.

Exemplo

- Consideramos o seguinte conjunto:
  $(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610)$
- O comprimento do arranjo é 16.
- A busca por salto encontrará o valor de 55 com os seguintes passos, assumindo que o tamanho do bloco a ser pulado é 4.
  - PASSO 1: Saltar do índice 0 para o índice 4;
  - PASSO 2: Saltar do índice 4 para o índice 8;
  - PASSO 3: Saltar do índice 8 para o índice 12;
  - PASSO 4: Como o elemento no índice 12 é maior que 55, daremos um salto para trás para chegar ao índice 8.
  - PASSO 5: Faça uma busca linear do índice 8 para obter o elemento 55.

Qual é o tamanho ideal do bloco a ser pulado?

- No pior caso, temos que fazer n/m saltos e se o último valor verificado for maior do que o elemento a ser procurado, fazemos comparações $m-1$ mais para a busca sequencial.
- Portanto, o número total de comparações no pior caso será $((n/m)+m-1)$.
- O valor da função $((n/m)+m-1)$ será mínimo quando $m=\sqrt{n}$.
- Portanto, o melhor tamanho do passo é $m=\sqrt{n}$.
- A complexidade temporal do algoritmo é $\mathbf{O}(\sqrt{n})$.

Implementação

```
#include <bits/stdc++.h>
using namespace std;

int jumpSearch(int arr[], int x, int n)
{
    // Finding block size to be jumped
    int step = sqrt(n);

    // Finding the block where element is present (if it is present)
    int prev = 0;
    while (arr[min(step, n)-1] < x)
    {
        prev = step;
        step += sqrt(n);
        if (prev >= n)
            return -1;
    }
```

Implementação - 2

```
    // Doing a linear search for x in block beginning with prev.
    while (arr[prev] < x)
    {
        prev++;

        // If we reached next block or end of array, element is not p
        if (prev == min(step, n))
            return -1;
    }
    // If element is found
    if (arr[prev] == x)
        return prev;

    return -1;
}
```

Implementação - 3

```
// Driver program to test function
int main()
{
    int arr[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21,
                  34, 55, 89, 144, 233, 377, 610 };
    int x = 55;
    int n = sizeof(arr) / sizeof(arr[0]);

    // Find the index of 'x' using Jump Search
    int index = jumpSearch(arr, x, n);

    // Print the index where 'x' is located
    cout << "\nNumber " << x << " is at index " << index;
    return 0;
}
```

Busca Interpolada

- Dado um arranjo ordenado `arr[]` de n valores distribuídos uniformemente.
- Busca sequencial encontra o elemento no tempo $\mathbf{O}(n)$, Busca por salto leva o tempo $\mathbf{O}(\sqrt{n})$ e Busca binária leva o tempo $\mathbf{O}(\log n)$.
- A Busca Interpolada é uma melhoria em relação à Busca Binária para instâncias, onde os valores em um arranjo ordenado são uniformemente distribuídos.
- A Busca Binária sempre vai para o elemento central para verificar.
- Por outro lado, a Busca Interpolada pode ir para locais diferentes de acordo com o valor da chave que está sendo pesquisada.
- Por exemplo, se o valor da chave estiver mais próximo do último elemento, é provável que a Busca Interpolada inicie a busca no lado do fim.

Busca Interpolada - 2

- Para encontrar a posição a ser pesquisada, ela usa a seguinte fórmula:

```
pos = lo + [ (x-arr[lo])*(hi-lo) / (arr[hi]-arr[Lo]) ]

arr[] ==> Array where elements need to be searched
x     ==> Element to be searched
lo    ==> Starting index in arr[]
hi    ==> Ending index in arr[]
```

Algoritmo

- Passo1: Em um laço, calcular o valor de "pos" usando a fórmula da posição.
- Passo2: Se for igual, retornar o índice do item, e sair.
- Passo3: Se o item for menor que `arr[pos]`, calcular a posição da sonda da sub-arranjo esquerda.
  Caso contrário, calcule o mesmo na subarranjo da direita.
- Passo4: Repita até encontrar uma correspondência ou até que a subarranjo se reduza a zero.

Implementação

```
#include<bits/stdc++.h>
using namespace std;

// If x is present in arr[0..n-1], then returns index of it,
// else returns -1.
int interpolationSearch(int arr[], int n, int x)
{
    // Find indexes of two corners
    int lo = 0, hi = (n - 1);

    // Since array is sorted, an element present in array
    // must be in range defined by corner
    while (lo <= hi && x >= arr[lo] && x <= arr[hi])
    {
        if (lo == hi)
        {
            if (arr[lo] == x) return lo;
            return -1;
        }
```

Implementação - 2

```
        // Probing the position with keeping
        // uniform distribution in mind.
        int pos = lo + (((double)(hi - lo) /
                  (arr[hi] - arr[lo])) * (x - arr[lo]));

        // Condition of target found
        if (arr[pos] == x)
            return pos;

        // If x is larger, x is in upper part
        if (arr[pos] < x)
            lo = pos + 1;

        // If x is smaller, x is in the lower part
        else
            hi = pos - 1;
    }
    return -1;
}
```

Implementação - 3

```
// Driver Code
int main()
{
    // Array of items on which search will
    // be conducted.
    int arr[] = {10, 12, 13, 16, 18, 19, 20, 21,
                 22, 23, 24, 33, 35, 42, 47};
    int n = sizeof(arr)/sizeof(arr[0]);

    int x = 18; // Element to be searched
    int index = interpolationSearch(arr, n, x);

    // If element was found
    if (index != -1)
        cout << "Element found at index " << index;
    else
        cout << "Element not found.";
    return 0;
}
```

Busca Binária vs Busca Interpolada

- A Busca Binária vai até o elemento central para verificar independentemente da chave de busca.
- A Busca Interpolada pode ir para locais diferentes de acordo com a chave de pesquisa.
- Se o valor da chave de busca estiver próximo ao último elemento, a Busca Interpolada provavelmente iniciará a busca no lado final.
- Em média, a Busca Interpolada faz comparações de $\Theta(\log(\log(n)))$ (se os elementos estiverem uniformemente distribuídos).
- No pior caso (por exemplo, onde os valores numéricos das chaves aumentam exponencialmente), pode fazer até comparações $\mathbf{O}(n)$.

Busca exponencial

- A busca exponencial envolve duas etapas:
  - Encontrar a faixa onde o elemento está presente.
  - Faça a Busca Binária na faixa encontrada acima.
- Como encontrar a faixa onde o elemento pode estar presente:
  - A ideia é começar com o tamanho de subarranjo 1, comparar seu último elemento com `x`, depois tentar tamanho 2, depois 4 e assim por diante até que o último elemento de um subarranjo não seja maior.
  - Uma vez encontrado um índice i (depois de repetidas dobras de i), sabemos que o elemento deve estar presente entre $i/2$ e i
  - $i/2$: porque não conseguimos encontrar um valor maior na iteração anterior.
-

Implementação

```
#include <bits/stdc++.h>
using namespace std;

int binarySearch(int arr[], int, int, int);

// Returns position of first occurrence of x in array
int exponentialSearch(int arr[], int n, int x)
{
    // If x is present at firt location itself
    if (arr[0] == x)
        return 0;

    // Find range for binary search by repeated doubling
    int i = 1;
    while (i < n && arr[i] <= x)
        i = i*2;

    //  Call binary search for the found range.
    return binarySearch(arr, i/2,
                        min(i, n-1), x);
}
```

Implementação - 2

```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l)/2;

        // If the element is present at the middle itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then it
        // can only be present n left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid-1, x);

        // Else the element can only be present in right subarray
        return binarySearch(arr, mid+1, r, x);
    }

    // We reach here when element is not presentin array
    return -1;
}
```

Implementação - 3

```
// Driver code
int main(void)
{
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr)/ sizeof(arr[0]);
    int x = 10;
    int result = exponentialSearch(arr, n, x);
    (result == -1)? printf("Element is not present in array")
                  : printf("Element is present at index %d", result);
    return 0;
}
```

Busca ternária

```
int ternarySearch(int arr[], int l, int r, int x)
{
    if (r >= l)
    {
          int mid1 = l + (r - l)/3;
          int mid2 = mid1 + (r - l)/3;

          // If x is present at the mid1
          if (arr[mid1] == x)  return mid1;
          // If x is present at the mid2
          if (arr[mid2] == x)  return mid2;
          // If x is present in left one-third
          if (arr[mid1] > x) return ternarySearch(arr, l, mid1-1, x);
          // If x is present in right one-third
          if (arr[mid2] < x) return ternarySearch(arr, mid2+1, r, x);
          // If x is present in middle one-third
          return ternarySearch(arr, mid1+1, mid2-1, x);
    }
    // We reach here when element is not present in array
    return -1;
}
```

Algoritmos de ordenação

Bubble Sort

- Bubble Sort é o algoritmo de ordenação mais simples que funciona trocando repetidamente os elementos adjacentes se eles estiverem em ordem errada.
- A complexidade temporal do algoritmo é $\mathbf{O}(n^2)$.
- Exemplo: (5 1 4 2 8)

```
Primeiro passo

( 5 1 4 2 8 ) -> ( 1 5 4 2 8 ), troca porque 5 > 1

( 1 5 4 2 8 ) -> ( 1 4 5 2 8 ), troca porque 5 > 4

( 1 4 5 2 8 ) -> ( 1 4 2 5 8 ), troca porque 5 > 2

( 1 4 2 5 8 ) -> ( 1 4 2 5 8 ), não troca porque 5 < 8
```

Bubble Sort - 2

Segundo passo

(1 4 2 5 8) -> (1 4 2 5 8)

(1 4 2 5 8) -> (1 2 4 5 8), troca porque 4 > 2

(1 2 4 5 8) -> (1 2 4 5 8)

(1 2 4 5 8) -> (1 2 4 5 8)

O arranjo já está ordenado, mas nosso algoritmo não sabe se está completo.

O algoritmo precisa de um passe inteiro sem nenhuma troca para saber se está ordenado.

Terceiro passo

(1 2 4 5 8) -> (1 2 4 5 8)

(1 2 4 5 8) -> (1 2 4 5 8)

(1 2 4 5 8) -> (1 2 4 5 8)

(1 2 4 5 8) -> (1 2 4 5 8)

Bubble Sort - 3

		0	1	2	3	4	5	6	7
i = 0	j								
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i =1	0	3	1	5	8	2	4	7	**9**
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	**8**	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	**7**		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	**5**			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	**4**				
	1	1	2	3					
i = 6	0	1	2	**3**					
		1	**2**						

Selection Sort

- Selection Sort ordena um arranjo ao encontrar repetidamente o elemento mínimo (considerando ordem ascendente) da faixa não ordenada e colocá-lo no início.
- O algoritmo mantém dois sub-arranjos em um determinado arranjo.
  - 1/ O sub-arranjo que já está ordenado.
  - 2/ O sub-arranjo restante que não está ordenado.
- Em cada iteração, o elemento mínimo (considerando ordem ascendente) do sub-arranjo não ordenado é escolhido e movido para o sub-arranjo ordenado.
- A complexidade temporal do algoritmo é $\mathbf{O}(n^2)$.

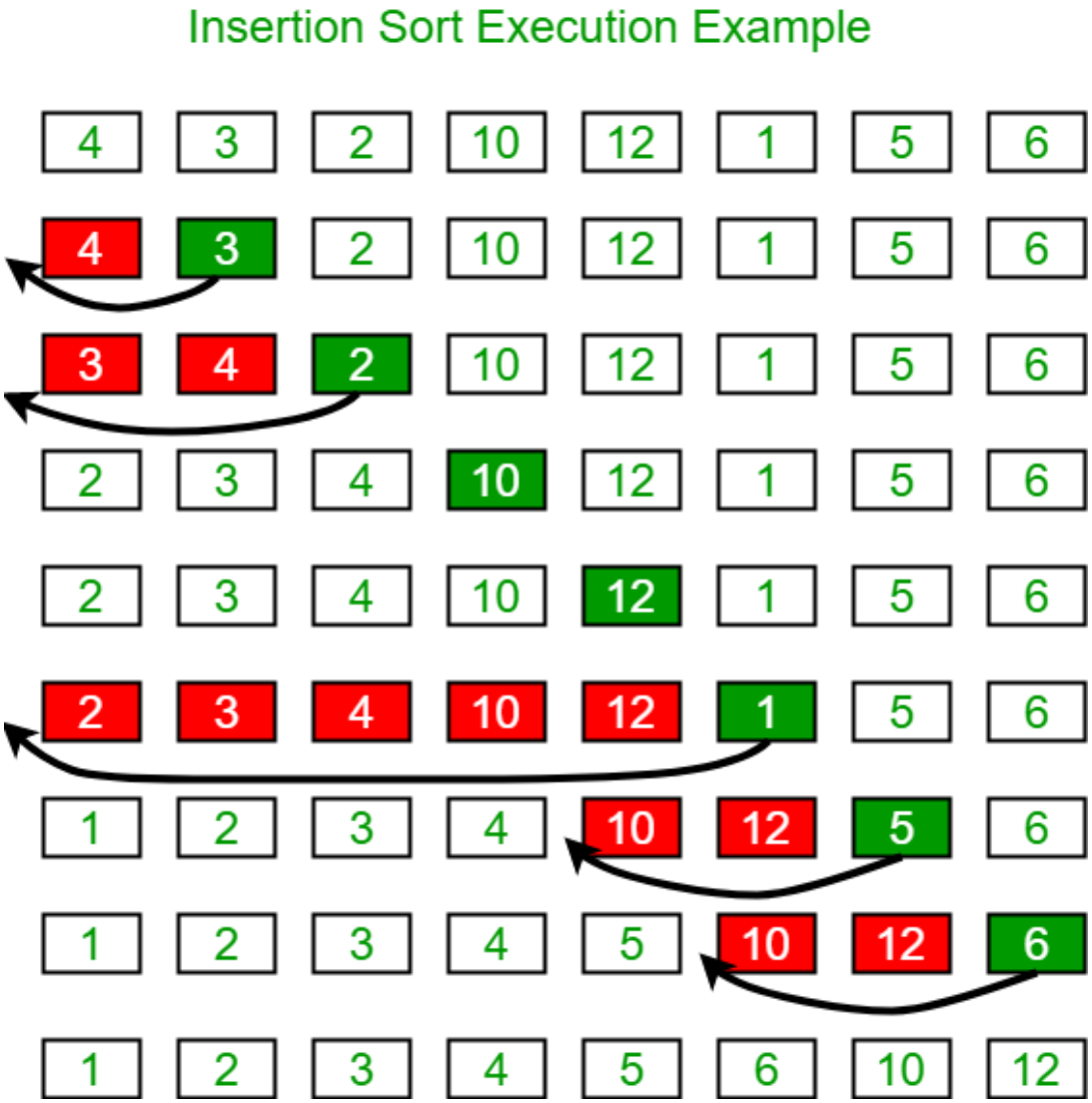
Exemplo

```
arr[] = 64 25 12 22 11

Encontre o elemento mínimo em arr[0...4] e coloque-o no
início

11 25 12 22 64

Encontre o elemento mínimo em arr[1...4] e coloque-o no
início de arr[1...4].

11 12 25 22 64

Encontre o elemento mínimo em arr[2...4] e coloque-o no
início de arr[2...4].

11 12 22 25 64

Encontre o elemento mínimo em arr[3...4] e coloque-o no
início de arr[3...4].

11 12 22 25 64
```

Insertion Sort

- Insertion Sort é um algoritmo de ordenação simples que funciona de forma semelhante à forma como você classifica as cartas de jogo em suas mãos.
- O conjunto é dividido em uma parte classificada e uma parte não classificada.
- Os valores da parte não classificada são escolhidos e colocados na posição correta na parte classificada.
- A complexidade temporal do algoritmo é $\mathbf{O}(n^2)$.
- Algoritmo
  - 1: Iterar de arr[1] até arr[n].
  - 2: Comparar o elemento atual (chave) com seu predecessor.
  - 3: Se o elemento chave for menor que seu predecessor, compará-lo com os elementos anteriores.
    Mova os elementos maiores uma posição para cima para abrir espaço para o elemento trocado.

Insertion Sort



A abordagem de divisão e conquista

- Muitos algoritmos úteis são recursivos em sua estrutura: para resolver um dado problema, eles chamam a si mesmos recursivamente uma ou mais vezes para lidar com subproblemas intimamente relacionados.
- Em geral, esses algoritmos seguem uma abordagem de divisão e conquista: eles desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas de menor tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original.

A abordagem de divisão e conquista - 2

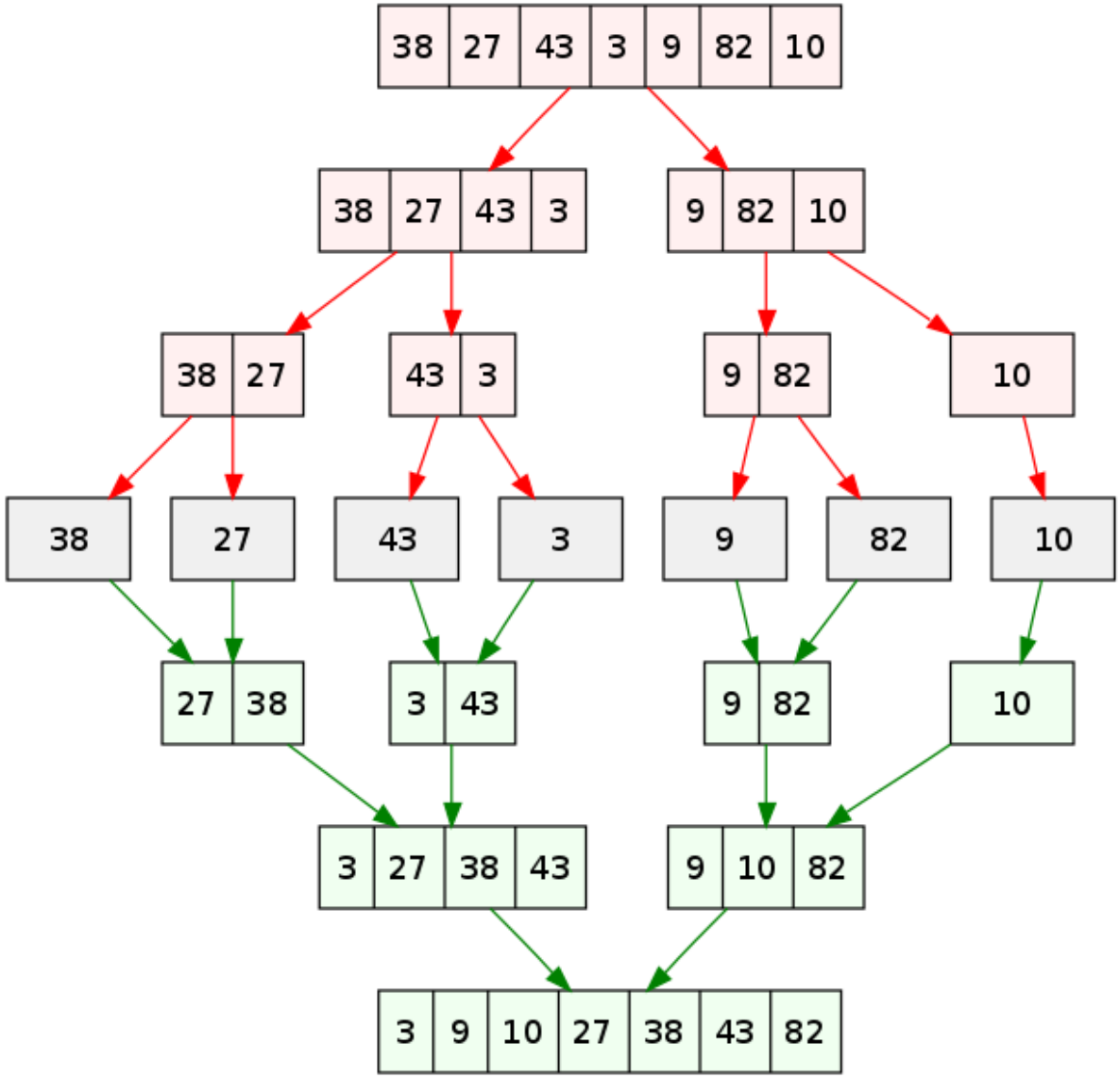
O paradigma de divisão e conquista envolve três passos em cada nível da recursão:

- **Divisão** do problema em determinado número de subproblemas que são instâncias menores do problema original.
- **Conquista** os subproblemas, resolvendo-os recursivamente. Porém, se os tamanhos dos sub-problemas forem pequenos o bastante, basta resolver os subproblemas de maneira direta.
- **Combinação** as soluções dadas aos subproblemas na solução para o problema original.

Merge Sort

- O algoritmo de ordenação por intercalação obedece rigorosamente ao paradigma de divisão e conquista.
- Intuitivamente, ele funciona do modo ilustrado a seguir.
  - **Divisão**: Divide a sequência de n elementos que deve ser ordenada em duas subsequências de n/2 elementos cada uma.
  - **Conquista**: Ordena as duas subsequências recursivamente, utilizando a ordenação por intercalação.
  - **Combinação**: Intercala as duas subsequências ordenadas para produzir a resposta ordenada.
- A recursão "extingue-se" quando a sequência a ser ordenada tiver comprimento 1, visto que nesse caso não há nenhum trabalho a ser feito, já que toda sequência de comprimento 1 já está ordenada.

Exemplo


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

Intercalação

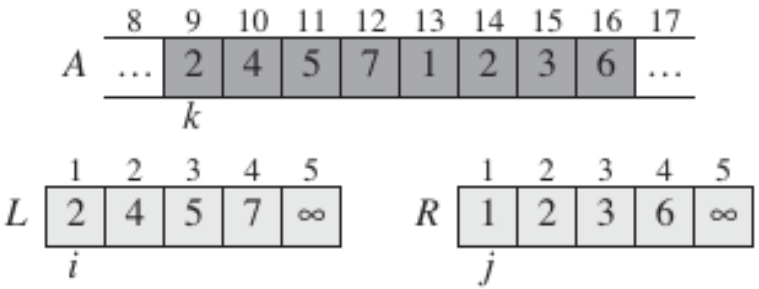
- A operação-chave do algoritmo de ordenação por intercalação é a intercalação de duas sequências ordenadas, no passo de "combinação".
- Para executar a intercalação, chamamos um procedimento auxiliar `Merge (A, p, q, r)`, onde
  - `A` é um arranjo;
  - `p`, `q` e `r` são índices de enumeração dos elementos do arranjo, tais que $\texttt{p} \leq \texttt{q} < \texttt{r}$.
- O procedimento considera que os subarranjos `A[p .. q]` e `A[q + 1 .. r]` estão em sequência ordenada.
- Ele os intercala (ou mescla) para formar um único subarranjo ordenado que substitui o subarranjo atual `A[p .. r]`.

Intercalação - 2

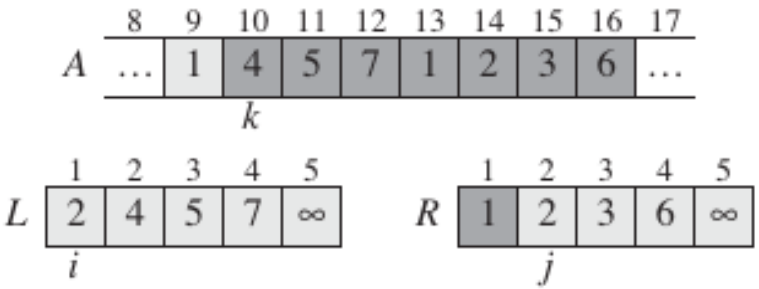
Exemplo do jogo de cartas

- Suponha que temos duas pilhas de cartas com a face para cima sobre uma mesa.
- Cada pilha está ordenada, com as cartas de menor valor em cima.
- Desejamos juntar as duas pilhas (fazendo a intercalação) em uma única pilha de saída ordenada, que ficará com a face para baixo na mesa.
- Nosso passo básico consiste em escolher a menor das duas cartas superiores nas duas pilhas viradas para cima, removê-la de sua pilha (o que exporá uma nova carta superior) e colocar essa carta com a face voltada para baixo sobre a pilha de saída.
- Repetimos esse passo até uma pilha de entrada se esvaziar e, então, simplesmente pegamos a pilha de entrada restante e a colocamos virada para baixo sobre a pilha de saída.

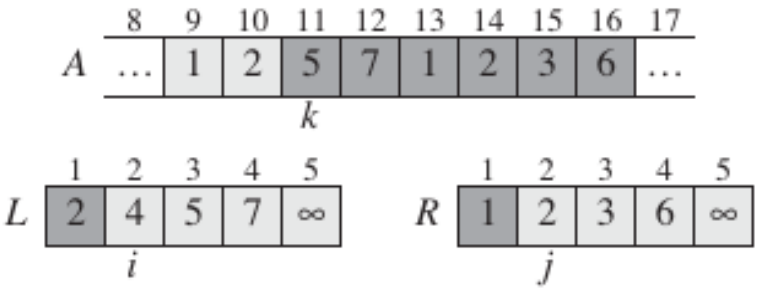
Exemplo



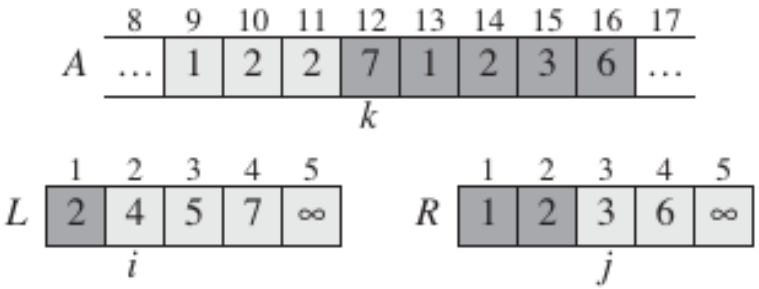
(a)



(b)

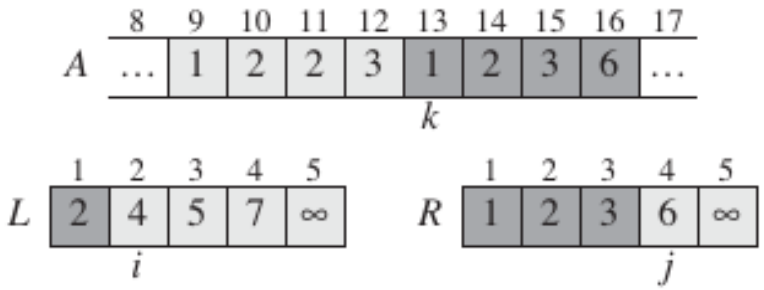


(c)

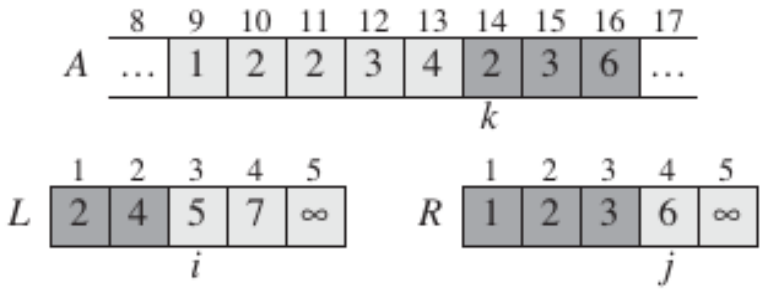


(d)

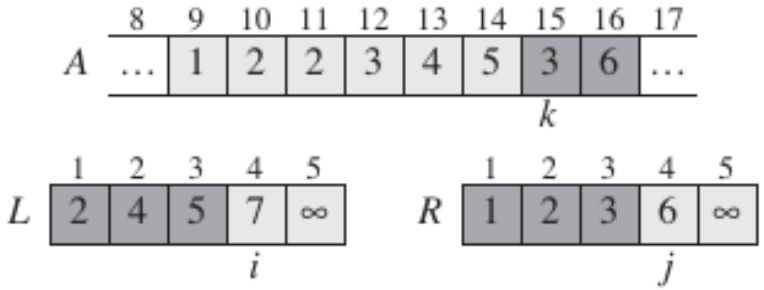
Exemplo - 2



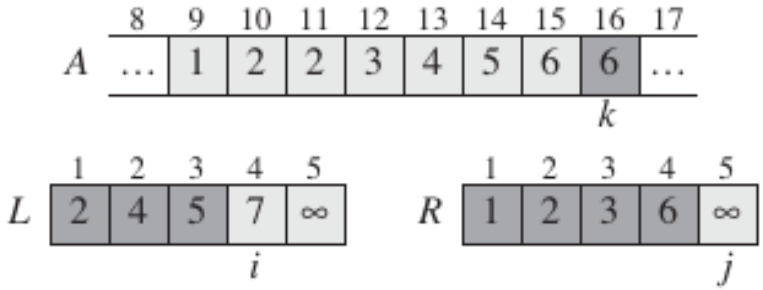
(e)



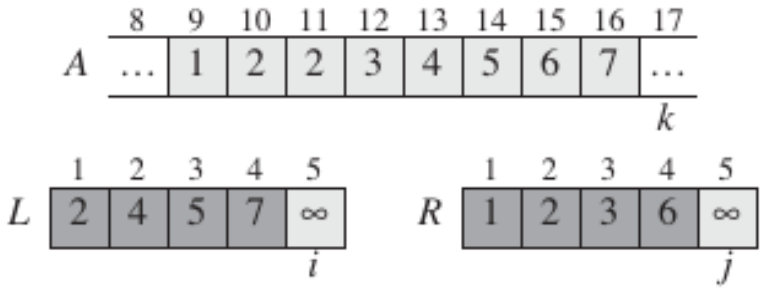
(f)



(g)



(h)



(i)

Pseudo código: Merge

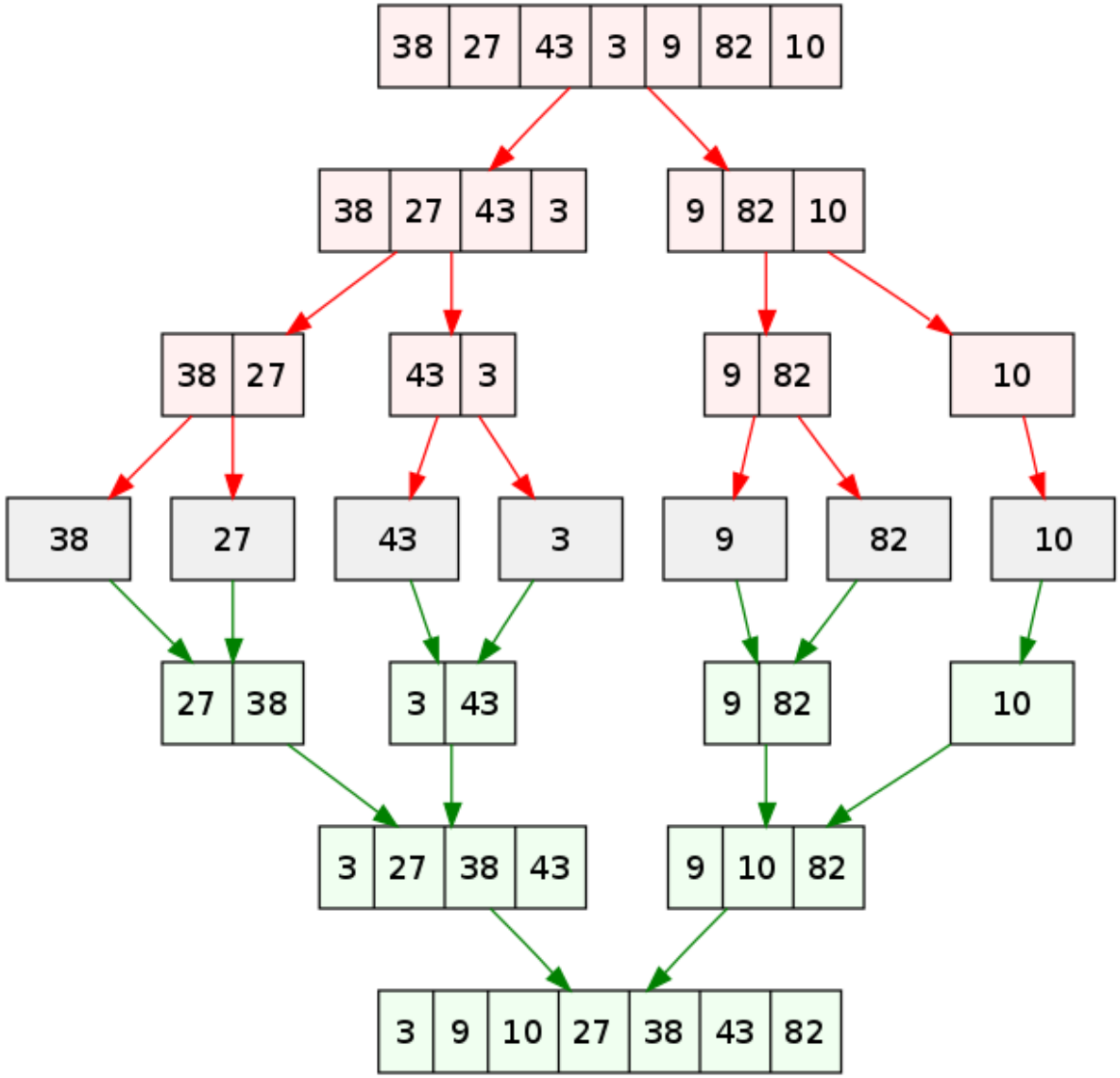
```
MERGE(A, p, q, r)
1   n_1 = q − p + 1
2   n_2 = r − q
3   sejam L[1..n_1 + 1] e R[1.. n_2 + 1] novos arranjos
4   for i = 1 to n_1
5       L[i] = A[p + i − 1]
6   for j = 1 to n_2
7       R[j] = A[q + j]
8   L[n_1 + 1] = ∞
9   R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          then A[k] = L[i]
15               i = i + 1
16          else A[k] = R[j]
17               j = j + 1
```

Pseudo código: Merge Sort

- Agora podemos usar o procedimento `Merge` como uma subrotina no algoritmo de ordenação por intercalação.
- O procedimento `Merge-Sort(A, p, r)` ordena os elementos do subarranjo `A[p .. r]`.
- Se `p` $\geq$ `r`, o subarranjo tem no máximo um elemento e, portanto, já está ordenado.
- Caso contrário, a etapa de divisão simplesmente calcula um índice `q` que subdivide `A[p .. r]` em dois subarranjos: `A[p .. q]`, contendo `n/2` elementos, e `A[q + 1 .. r]`, contendo `n/2` elementos.

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q = ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

Exemplo


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

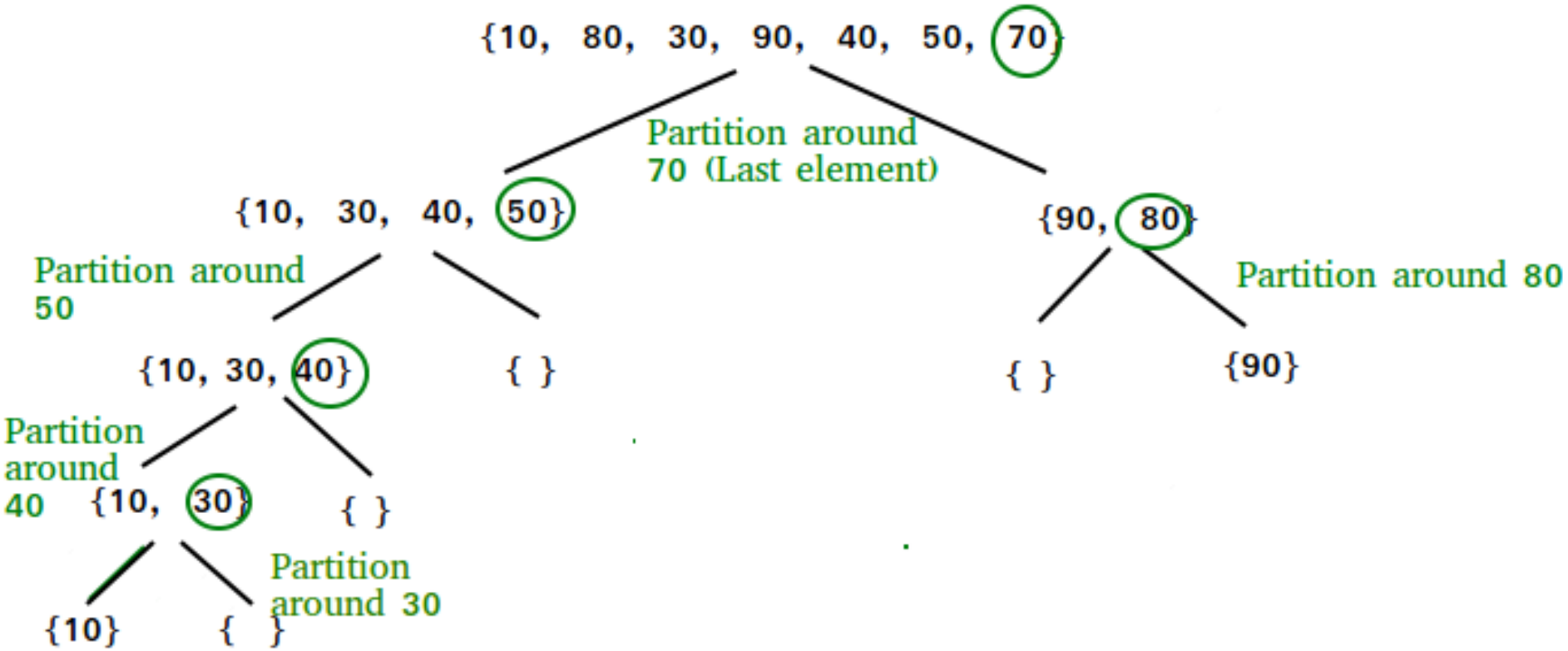
Merge Sort: complexidade temporal

- A complexidade temporal do algoritmo é $\mathbf{O}(n \log n)$.
- Mais lento em comparação com outros algoritmos de ordenação para conjuntos menores.
- O algoritmo de ordenação por fusão requer um espaço de memória adicional para o arranjo temporário.
- Ele passa por todo o processo, mesmo que o arranjo seja classificado.

Quick Sort

- O quicksort, como a ordenação por intercalação, aplica o paradigma de divisão e conquista.
- Ele escolhe um elemento como pivô e separa o conjunto dado em torno do pivô escolhido.
- Há muitas versões diferentes do QuickSort que escolhem o pivô de diferentes maneiras.
  - Sempre escolha o primeiro elemento como pivô.
  - Sempre escolha o último elemento como pivô (implementado abaixo)
  - Escolha um elemento aleatório como pivô.
  - Escolha a mediana como pivô.

Exemplo


{10, 80, 30, 90, 40, 50, 70}
Partition around
70 (Last element)
{10, 30, 40, 50}
{90, 80}
Partition around
50
Partition around 80
{10, 30, 40}
{ }
{ }
{90}
Partition
around
40
{10, 30}
{ }
Partition
around 30
{10}
{ }

QuickSort - 2

- **Divisão:** Particionar (reorganizar) o arranjo `A[p .. r]` em dois subarranjos (possivelmente vazios) `A[p .. q - 1]` e `A[q + 1 .. r]` tais que, cada elemento de `A[p .. q - 1]` é menor ou igual a `A[q]` que, por sua vez, é menor ou igual a cada elemento de `A[q + 1 .. r]`.
  Calcular o índice `q` como parte desse procedimento de particionamento.
- **Conquista:** Ordenar os dois subarranjos `A[p .. q -1]` e `A[q + 1 .. r]` por chamadas recursivas a quicksort.
- **Combinação:** Como os subarranjos já estão ordenados, não é necessário nenhum trabalho para combiná-los: o arranjo `A[p .. r]` inteiro agora está ordenado.

Pseudo código: QuickSort

```
QUICKSORT(A, p, r)
1  if p < r
2        q = PARTITION(A, p, r)
3        QUICKSORT(A, p, q − 1)
4        QUICKSORT(A, q + 1, r)
```

- Para ordenar um arranjo A inteiro, a chamada inicial é `QUICKSORT (A, 1, A.comprimento)`

Particionamento do arranjo

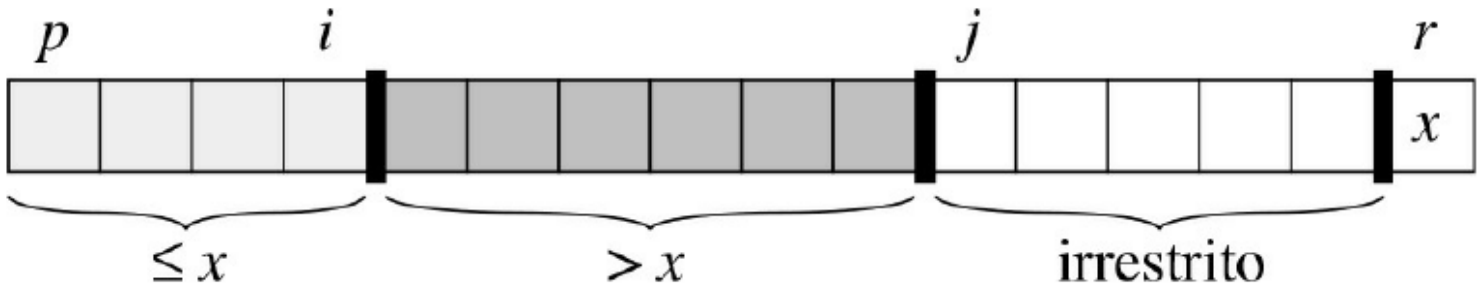
- A chave para o algoritmo é o procedimento `PARTITION`, que reorganiza o subarranjo `A[p .. r]` no lugar.
- `PARTITION` sempre seleciona um elemento `x = A[r]` como um elemento pivô ao redor do qual particionar o subarranjo `A[p .. r]`.

Pseudo código: Partition

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            trocar A[i] por A[j]
7  trocar A[i + 1] por A[r]
8  return i + 1
```

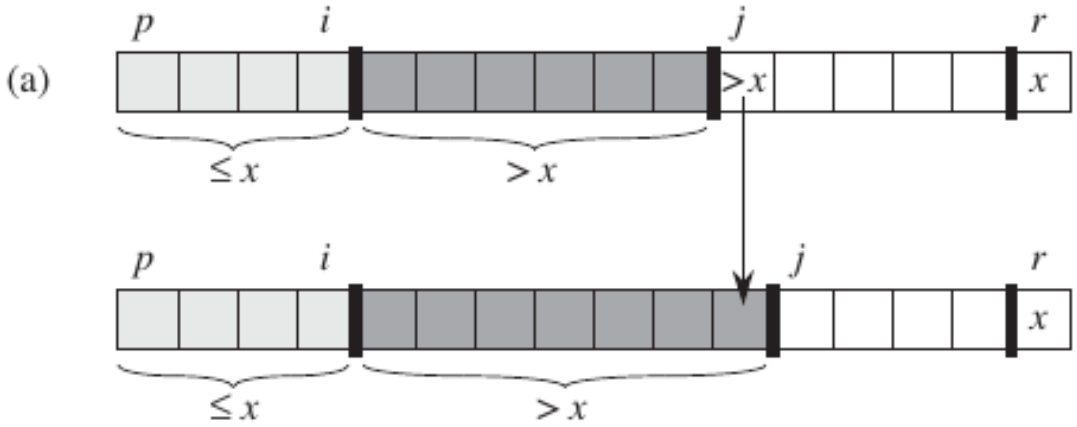
Pseudo código: analise

- À medida que é executado, o procedimento reparte o arranjo em quatro regiões (possivelmente vazias).
- No início de cada iteração do laço das linhas 3-6, para qualquer índice `k` do arranjo,
  - 1. Se `p` $\leq$ `k` $\leq$ `i`, então `A[k]` $\leq$ `x`.
  - 2. Se `i + 1` $\leq$ `k` $\leq$ `j - 1`, então `A[k] > x`.
  - 3. Se `k = r`, então `A[k] = x`.
- Os índices entre `j` e `r - 1` não são abrangidos por nenhum dos três casos, e os valores nessas entradas não têm nenhuma relação particular com o pivô `x`.

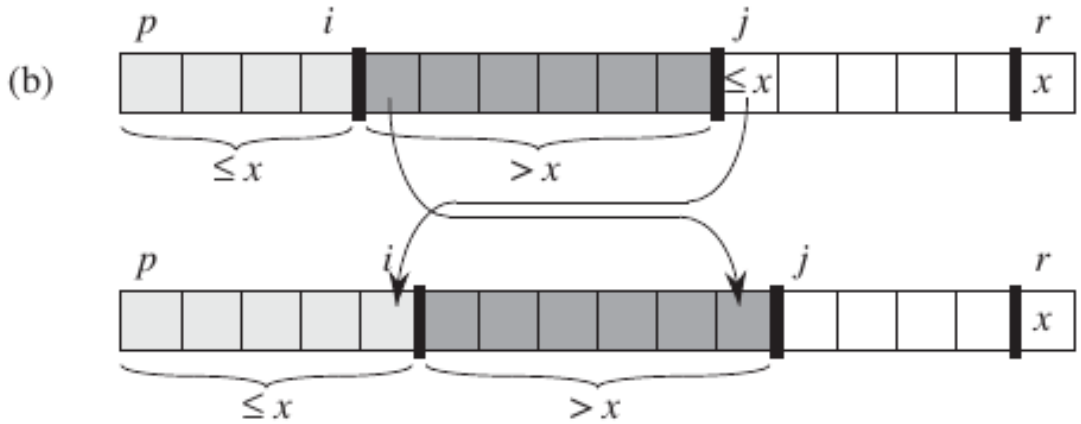


Pseudo código: analise - 2

- Se `A[j] > x`, a única ação é incrementar `j`.



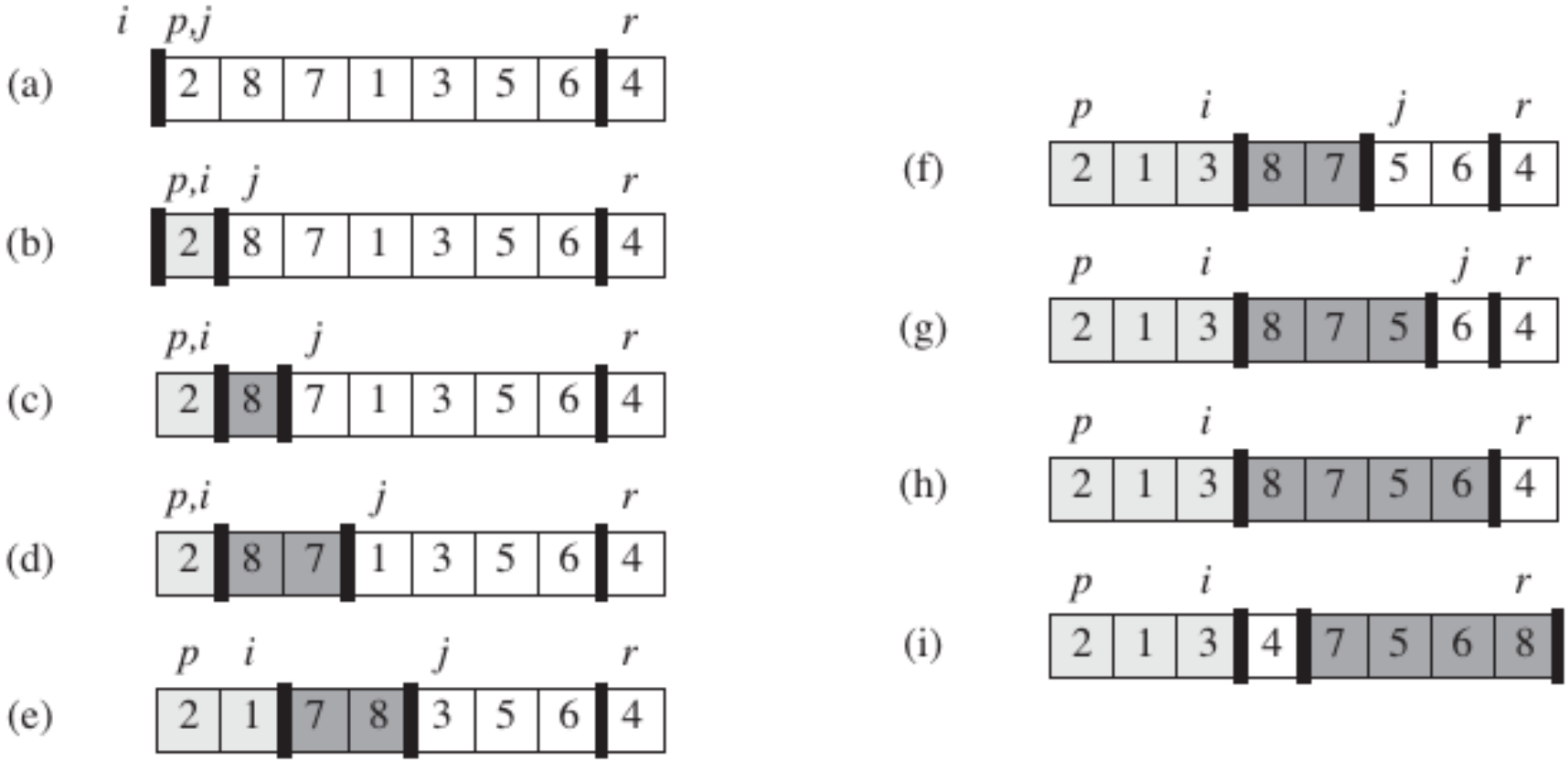
- Se `A[j]` $\leq$ `x`, o índice `i` é incrementado, `A[i]` e `A[j]` são permutados e, então, `j` é incrementado.



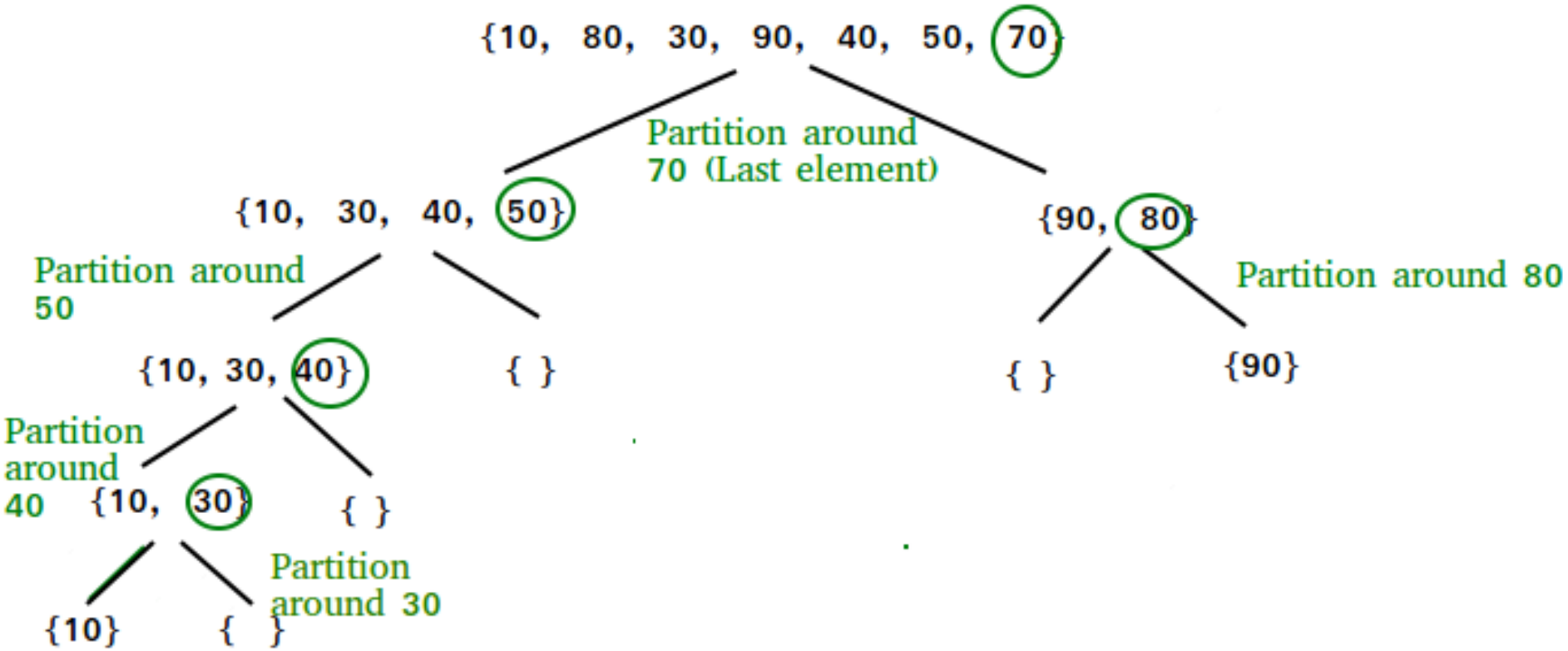
Pseudo código: analise - 3

- As duas linhas finais de `PARTITION` terminam permutando o elemento pivô pelo elemento maior que `x` na extrema esquerda e, com isso, deslocam o pivô até seu lugar correto no arranjo particionado; em seguida, retornam o novo índice do pivô.

Exemplo: Partition



Exemplo: QuickSort



Complexidade temporal de QuickSort

- Pior caso: n^2
  O pior caso ocorre quando o processo de partição sempre escolhe o maior ou menor elemento como pivô.
  Se considerarmos acima a estratégia de partição onde o último elemento é sempre escolhido como pivô, o pior caso ocorreria quando o arranjo já está classificado em ordem crescente ou decrescente.
- Melhor caso: $n \log n$
  O melhor caso ocorre quando o processo de partição sempre escolhe o elemento central como pivô.
- Caso médio: $n \log n$

Por que QuickSort é preferido ao MergeSort para arranjo?

- QuickSort em sua forma geral é uma classificação no local (ou seja, não requer nenhum armazenamento extra) enquanto que a MergeSort requer um armazenamento extra de N, N denotando o tamanho do arranjo.
  A alocação e desalocação do espaço extra utilizado para MergeSort aumenta o tempo de execução do algoritmo.
- Comparando a complexidade média, descobrimos que ambos os tipos têm complexidade média de $\mathbf{O}(N \log N)$, mas as constantes são diferentes.
  Para os arranjos, MergeSort perde devido ao uso de espaço extra.
- A maioria das implementações práticas de QuickSort usam versões aleatórias.
  A versão aleatória tem a complexidade temporal esperada de $\mathbf{O}(N \log N)$.
  O pior caso é possível na versão aleatória, mas não ocorre para um padrão em particular.

Por que o MergeSort é preferido ao QuickSort para listas ligadas?

- No caso de listas ligadas, o caso é diferente principalmente devido à diferença na alocação de memória das matrizes e listas vinculadas.
- Ao contrário dos arranjos, os nós das listas ligadas podem não estar adjacentes na memória.
- Ao contrário dos arranjos, na lista ligada, podemos inserir itens no meio em $\mathbf{O}(1)$ tempo.
- Portanto, a operação de fusão do tipo MergeSort pode ser implementada sem espaço extra para listas vinculadas.