DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estrutura de Dados

Ordenação: Shellsort e MergeSort

Professores: Anisio Lacerda
Wagner Meira Jr.

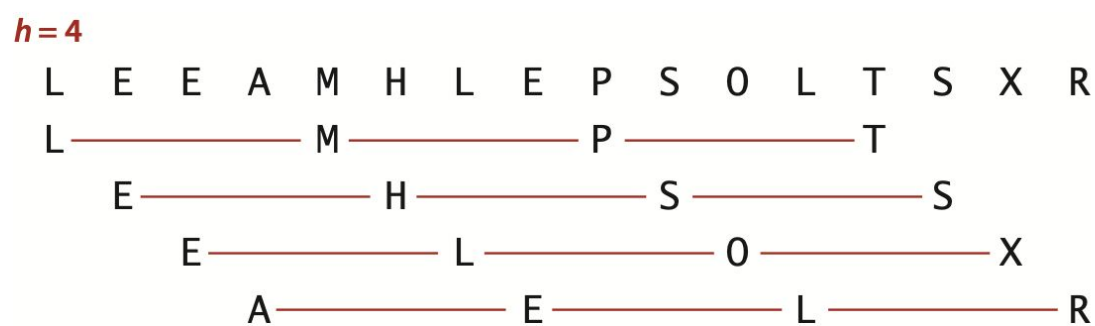
Shellsort

- Proposto por Donald Shell em 1959
- É uma extensão do **InsertionSort**
- Problemas com o **InsertionSort**: sempre troca elementos vizinhos
- O método do Shell permite a troca de elementos distantes
- Trocamos itens separados por ***h*** posições.
- Vamos reduzindo o valor de h até chegar em 1.

Shellsort: ideia

Reorganizar o vetor de entrada tal que cada *h*-ésima posição produza uma subsequência ordenada

Após essa reorganização temos um vetor *h-sorted* *h* subsequências ordenadas independentes, intercaladas



An *h*-sorted sequence is *h* interleaved sorted subsequences

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- h = 4

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- h = 4

45	56	12	43	**95**	19	8	67

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- h = 4

45	56	12	43	**95**	19	8	67
45	**19**	12	43	95	**56**	8	67

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- h = 4

45	56	12	43	**95**	19	8	67
45	**19**	12	43	95	**56**	8	67
45	19	**8**	43	95	56	**12**	67

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- h = 4

45	56	12	43	**95**	19	8	67
45	**19**	12	43	95	**56**	8	67
45	19	**8**	43	95	56	**12**	67
45	19	8	**43**	95	56	12	**67**

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- h = 2

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- h = 2

45	19	**8**	43	**95**	56	**12**	67

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- h = 2

45	19	**8**	43	**95**	56	**12**	67
8	19	12	43	45	56	95	67

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- h = 2

45	19	**8**	43	**95**	56	**12**	67
8	19	12	43	45	56	95	67
8	**19**	12	**43**	45	**56**	95	**67**

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- h = 2

45	19	**8**	43	**95**	56	**12**	67

8	19	12	43	45	56	95	67

8	**19**	12	**43**	45	**56**	95	**67**

8	19	12	43	45	56	95	67

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- $h = 1$

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- h = 1

8	19	12	43	45	56	95	67

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- h = 1

8	19	12	43	45	56	95	67

8	**19**	**12**	**43**	**45**	**56**	**95**	**67**

Shellsort - Exemplo

45	56	12	43	95	19	8	67

- h = 1

8	19	12	43	45	56	95	67

8	**19**	**12**	**43**	**45**	**56**	**95**	**67**

8	12	19	43	45	56	67	95

Shellsort

- Uma escolha simples de sequência para *h* é $n/2^i$.

```
void shellSort(int array[], int n) {
  for (int h = n / 2; h > 0; h /= 2) {
    for (int i = h; i < n; i += 1) {
      int temp = array[i];
      int j;
      for (j = i; j >= h && array[j-h] > temp; j-= h) {
        array[j] = array[j - h];
      }
      array[j] = temp;
    }
  }
}
```

Shellsort - Análise

- A complexidade de tempo depende da escolha da sequência.
- Para a sequência $n/2^i$ o Shellsort é $O(n^2)$.
- Existem sequências conhecidas as quais conseguimos obter complexidade um pouco melhor.
- A complexidade para algumas sequências utilizadas ainda é um problema em aberto.

Shellsort - Análise

- É estável utilizando a sequência $n/2^i$?

Shellsort - Análise

- É estável utilizando a sequência $n/2^i$?

2	6	**5**	**5**

Shellsort - Análise

- É estável utilizando a sequência $n/2^i$?

2	6	5	5

2	5	5	6

Shellsort - Análise

- É estável utilizando a sequência $n/2^i$?

2	6	**5**	**5**

2	**5**	**5**	6

Não!

Shellsort

Pior caso em comparações: $O(N^{3/2})$

Não existe prova do caso médio

Tempo de execução aceitável para vetores moderadamente grandes

Pequeno quantidade de código

- código em hardware ou sistemas embarcados

Não usa espaço extra

Mergesort

Mergesort

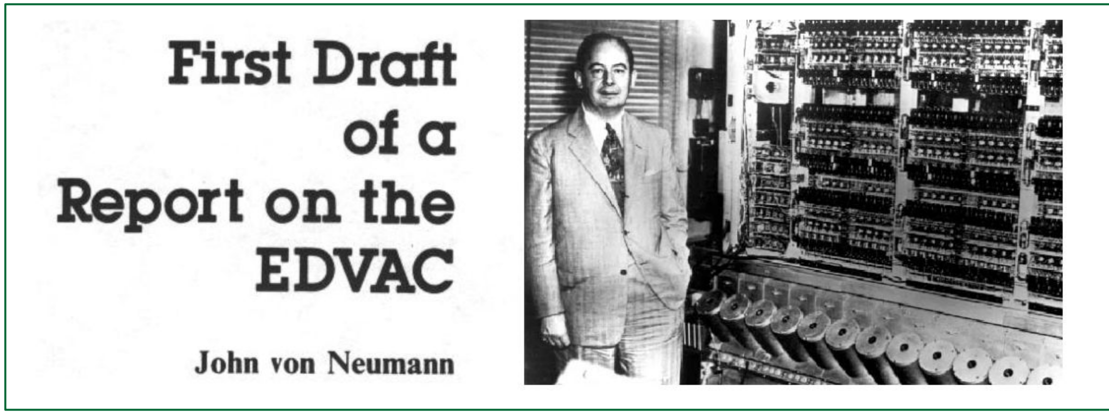
Python sort

- Timsort (2.3 - 3.10), Powersort (3.11)

Java sort for objects

Perl

Firefox JavaScript



Mergesort - Ideia

Divide o vetor em duas metades

Recursivamente ordena cada metade

Une (*merge*) as duas metades

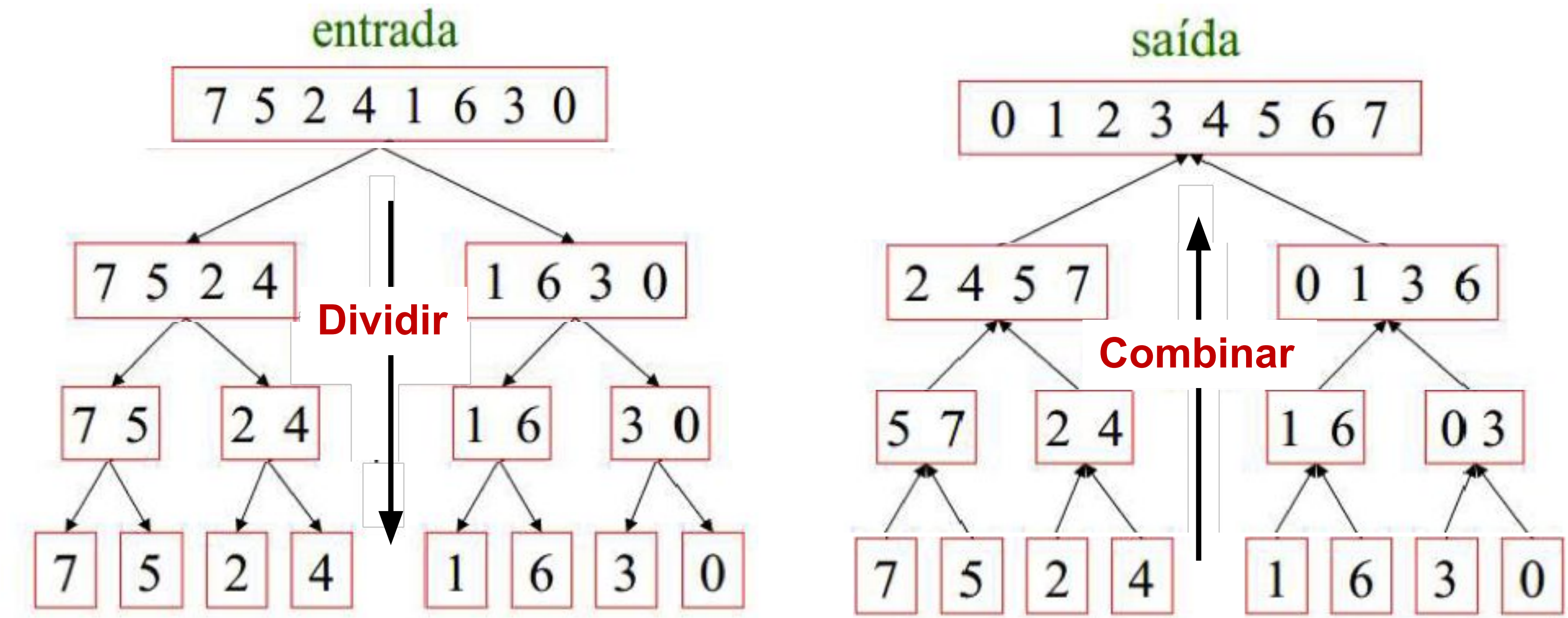
input	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
sort left half	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
sort right half	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge results	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

Mergesort overview

Merge Sort – Visão Geral

Ideia geral:

Dividir → Ordena os pares → Combina ordenando



Merge Sort - Pseudocódigo

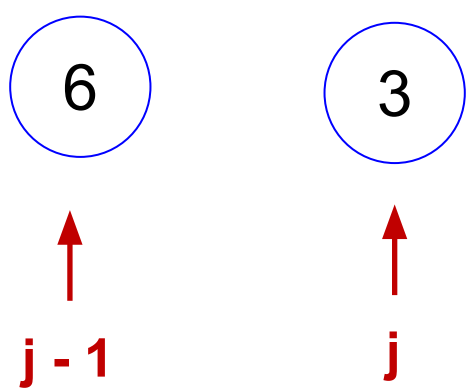
```
MergeSort(e,d)
{
   if (e < d )
   {
      meio = (e+d)/2;
      MergeSort(e, meio);
      MergeSort(meio+1, d);
      Merge(e, meio, d);
   }
}
```

Divide

Combina

Como juntar?

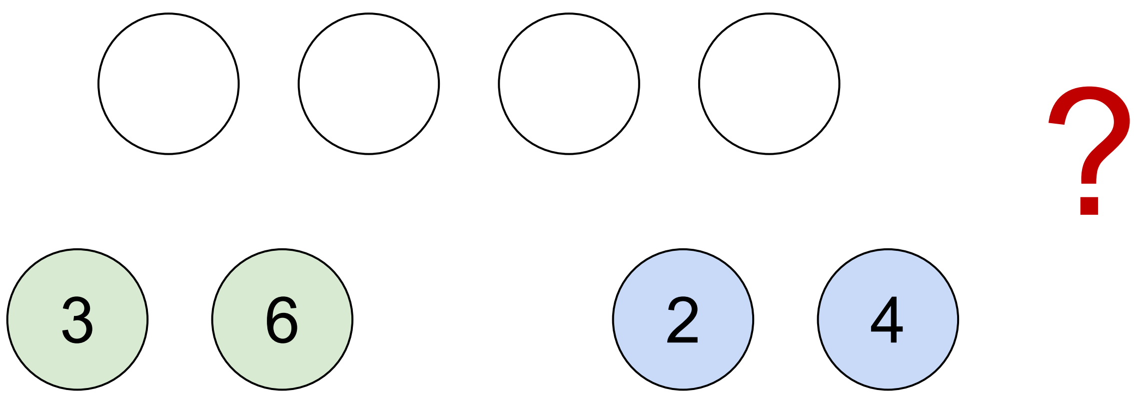
- Quando acabo de dividir o vetor, chego na situação em que cada vetor tem apenas 1 elemento, então começo a juntar
- Como juntar 2 elementos?




```
if(values[j] < values[j-1])
  swap(values, j-1, j);
```

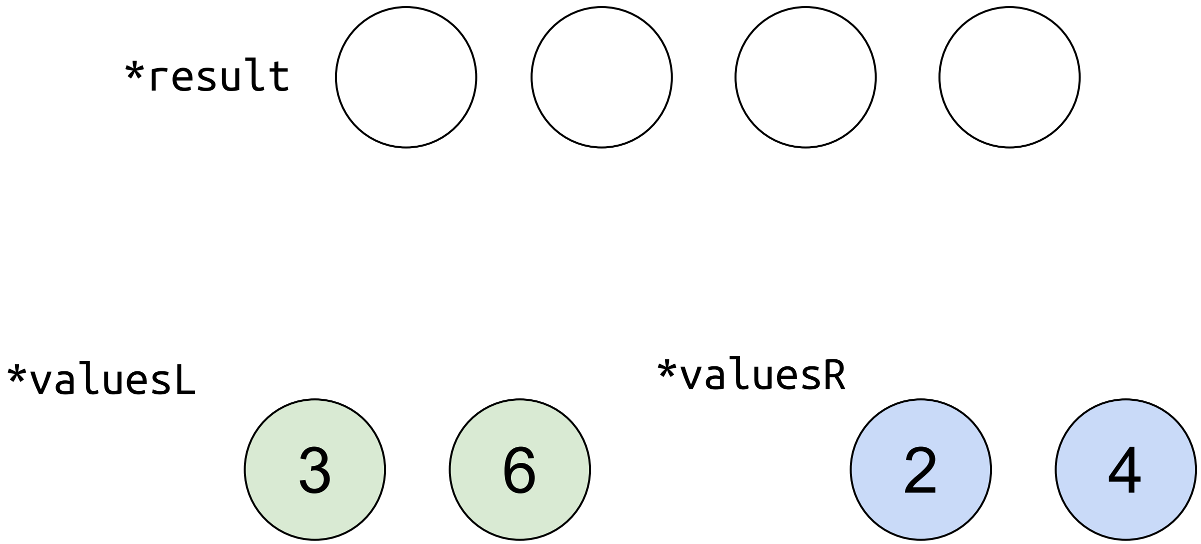
Merge

- Se eu tiver 2 vetores de 2 elementos ordenados, como criar um **novo** vetor de 4 elementos ainda ordenado?



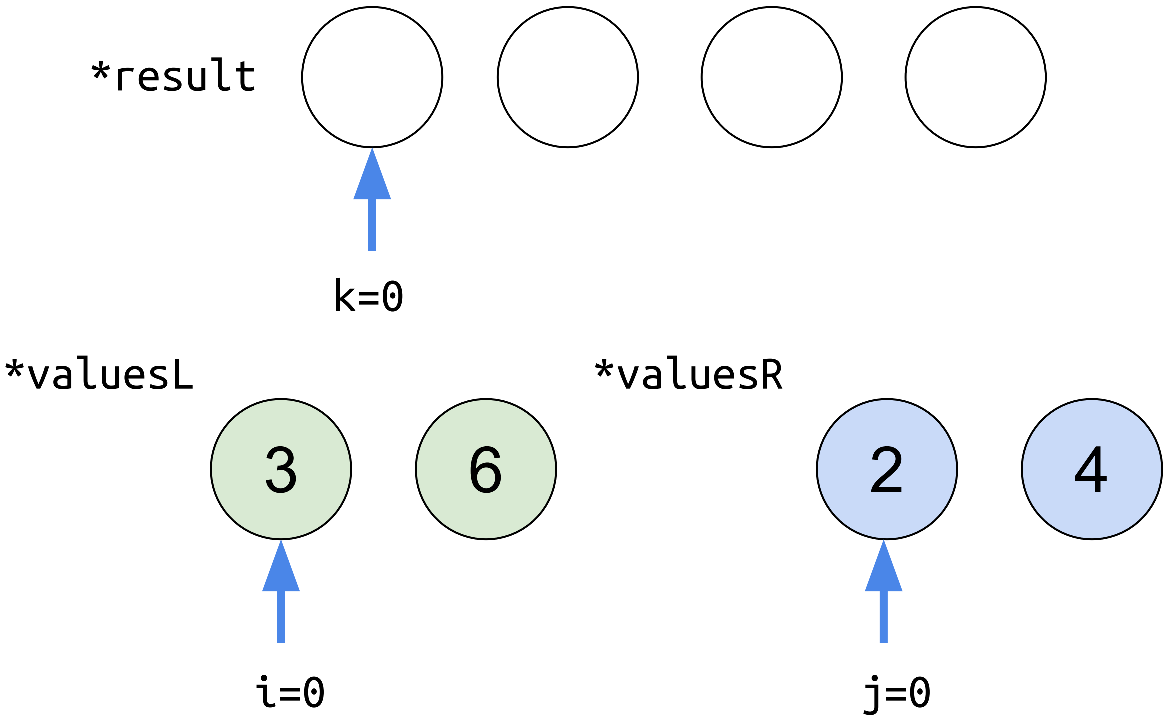
Merge

```c
int *merge(int *valuesL, int *valuesR, int nl  int nr) {
  int *result = (int *) malloc((nl+nr) * sizeof(int));
  //...
  return result
}
```

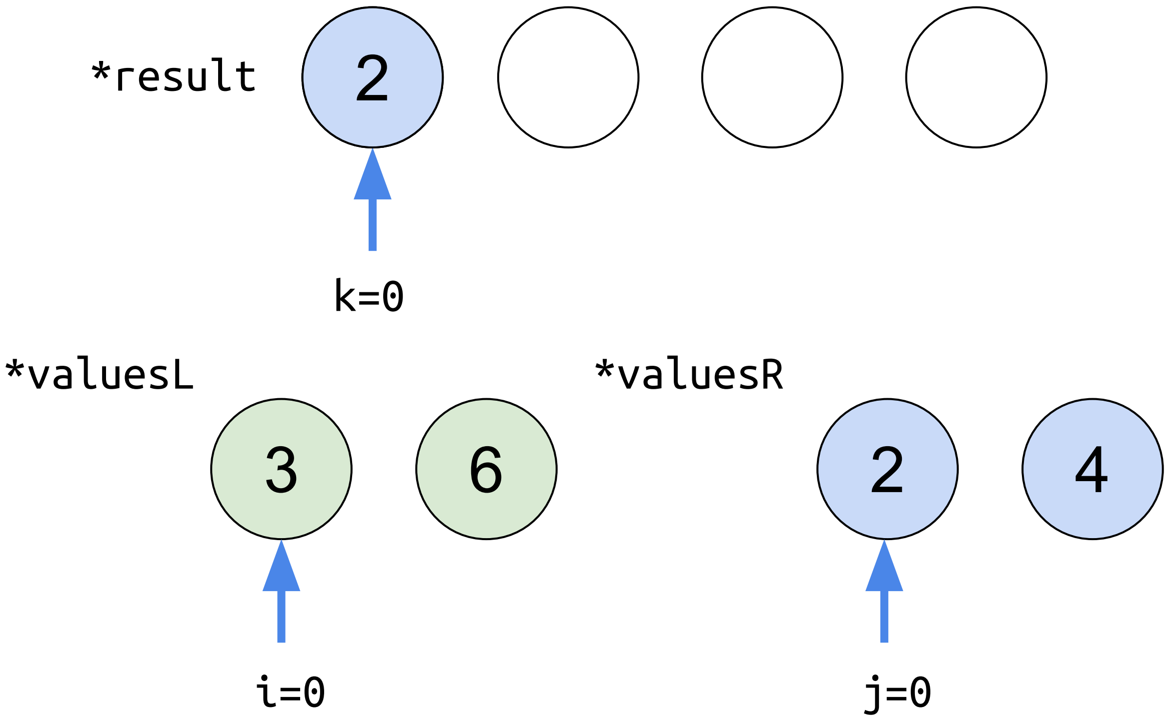



```c
nl = 2;
nr = 2;
```

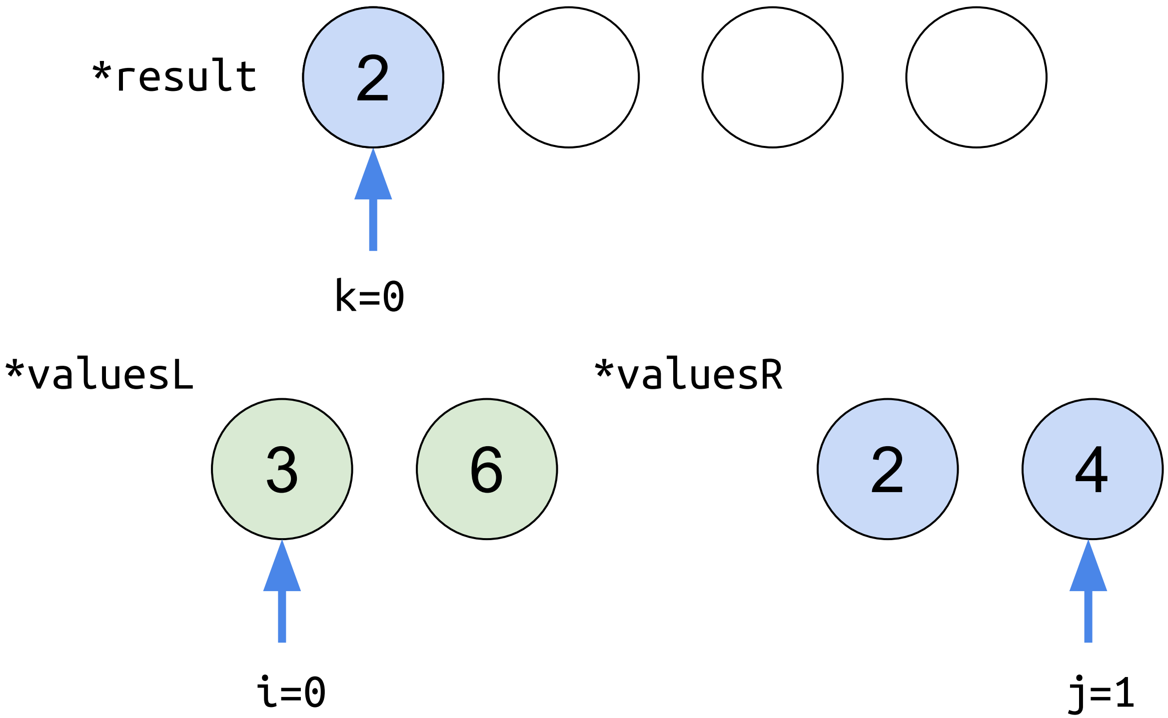
Merge



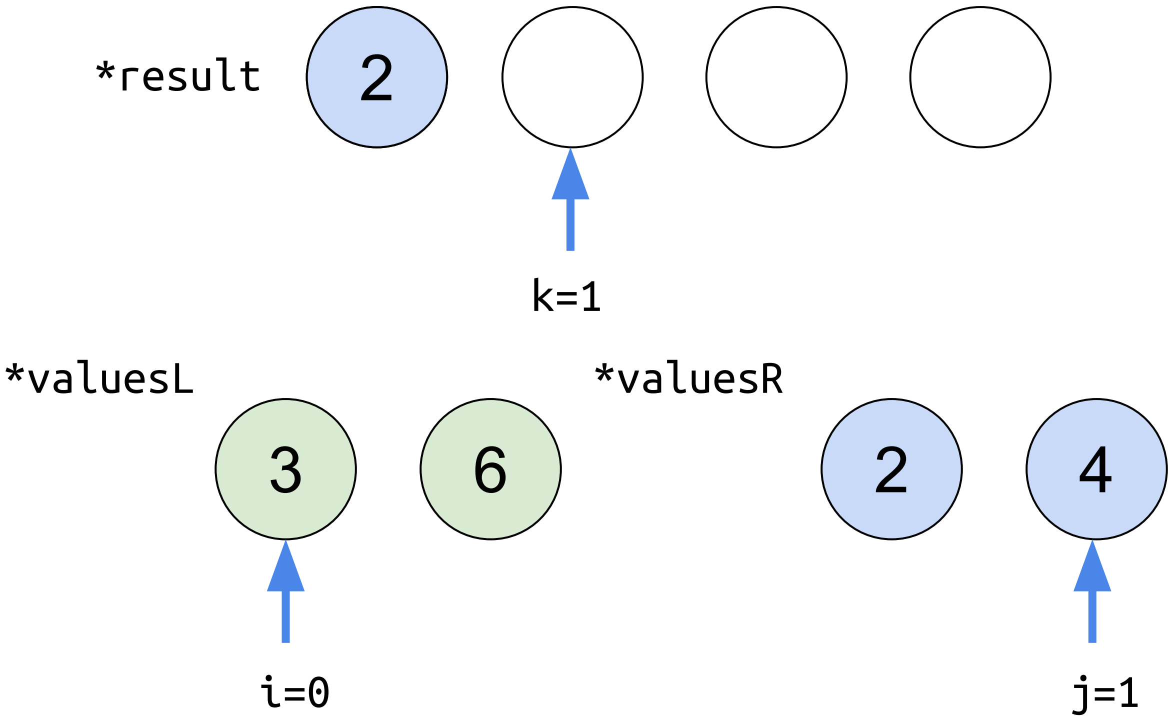
Merge



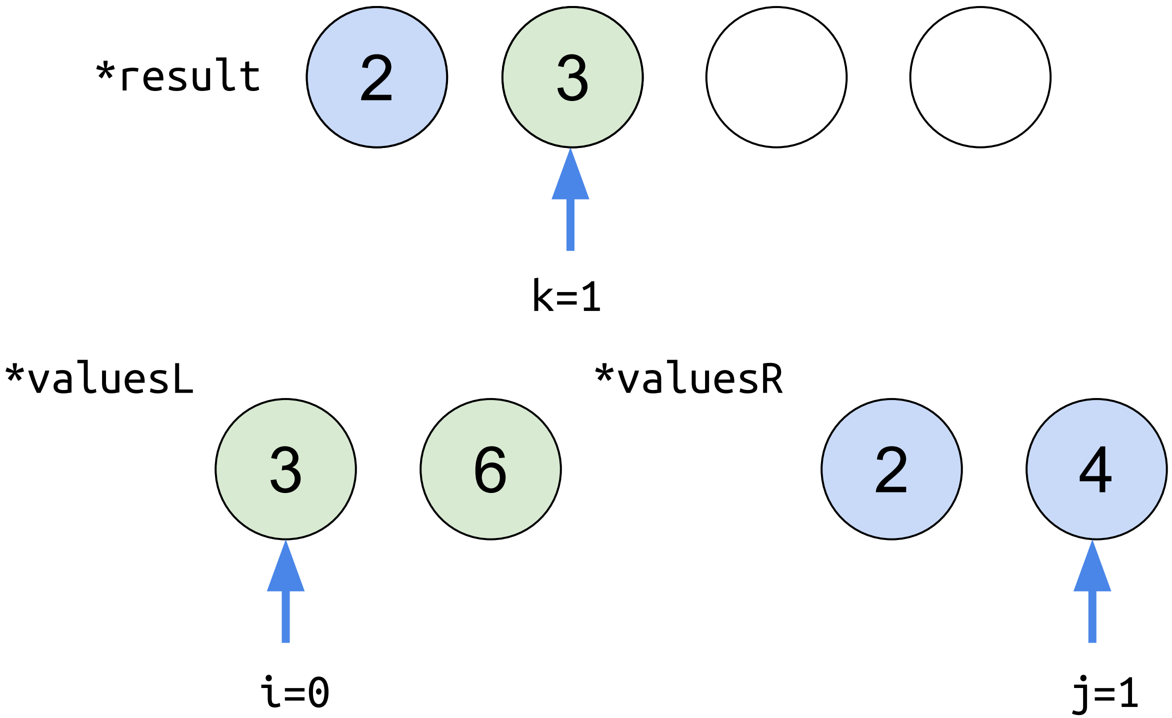
Merge



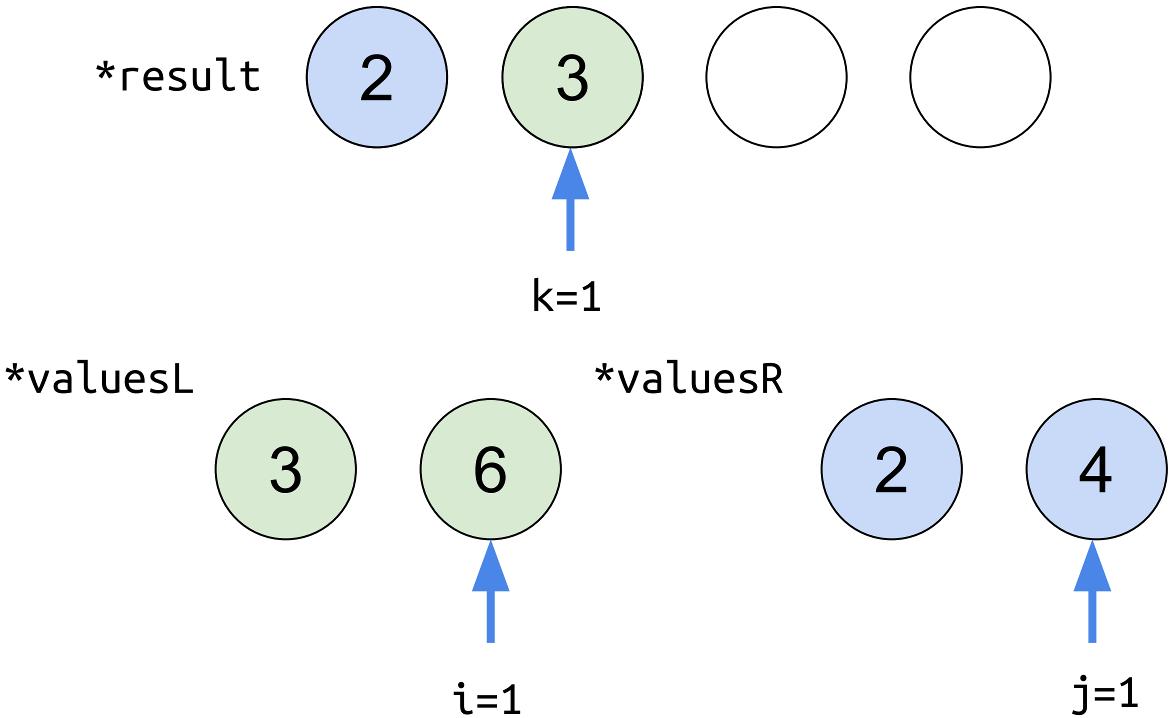
Merge



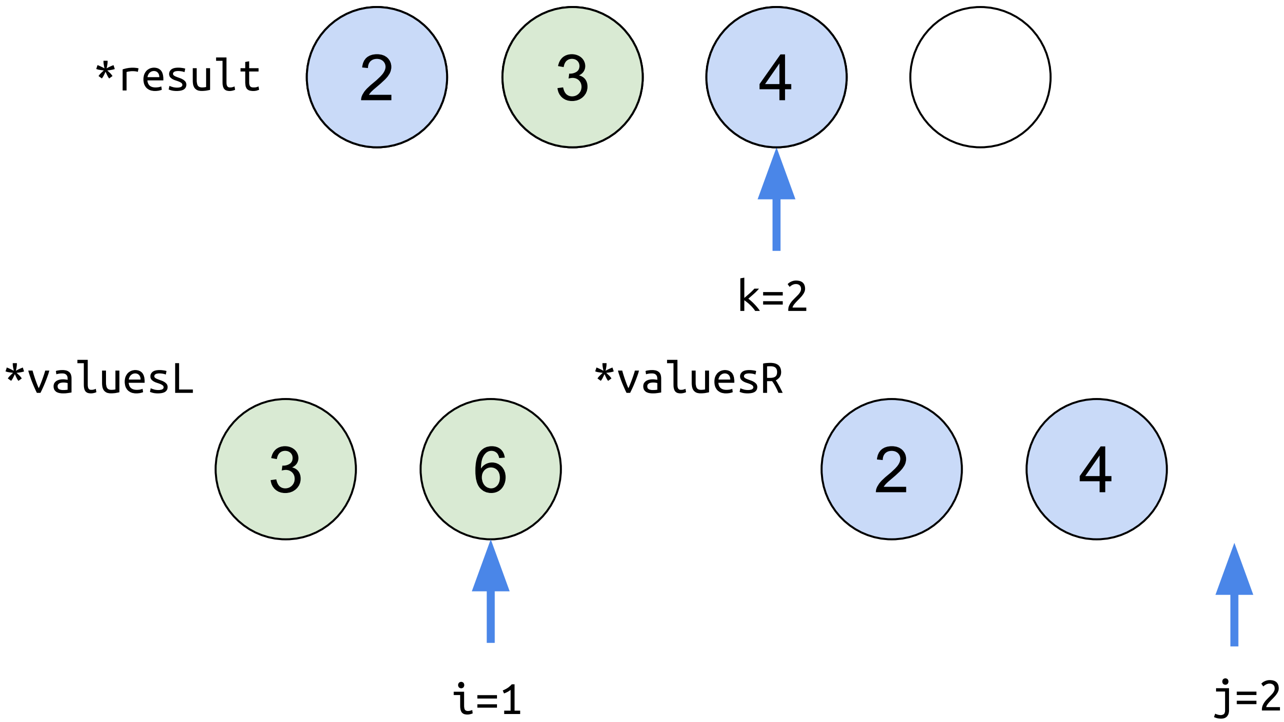
Merge



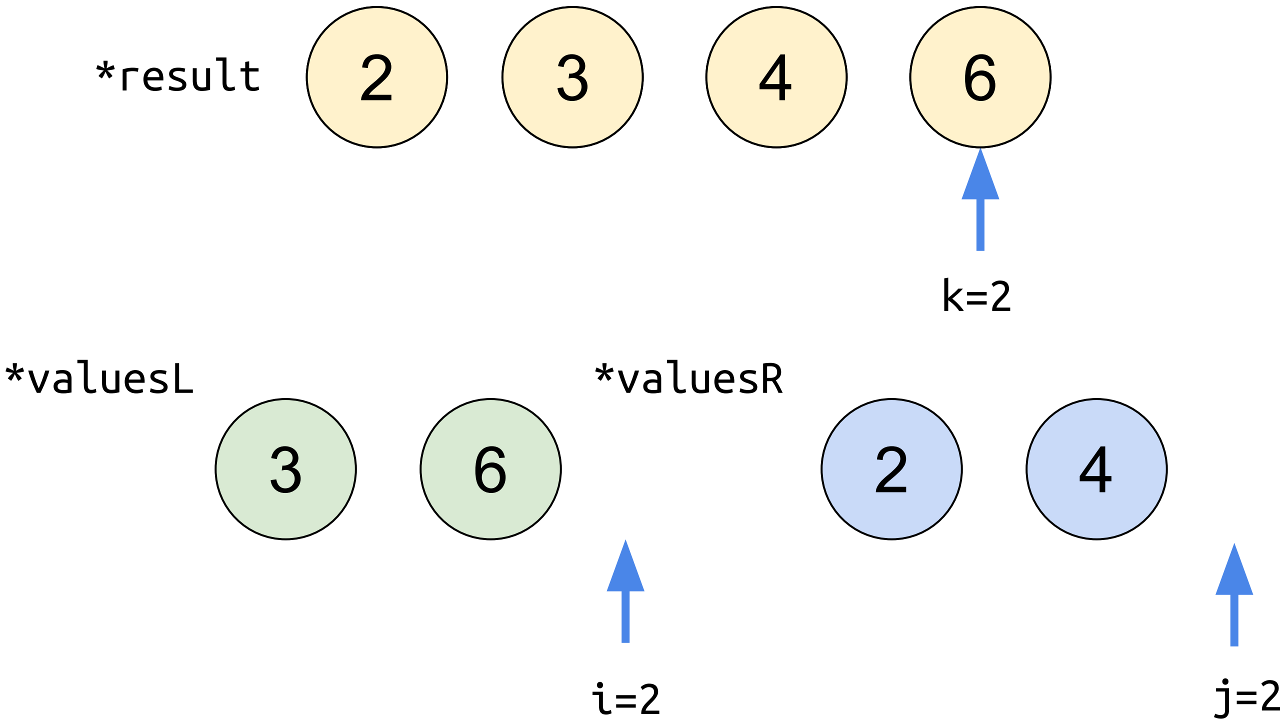
Merge



Merge



Merge

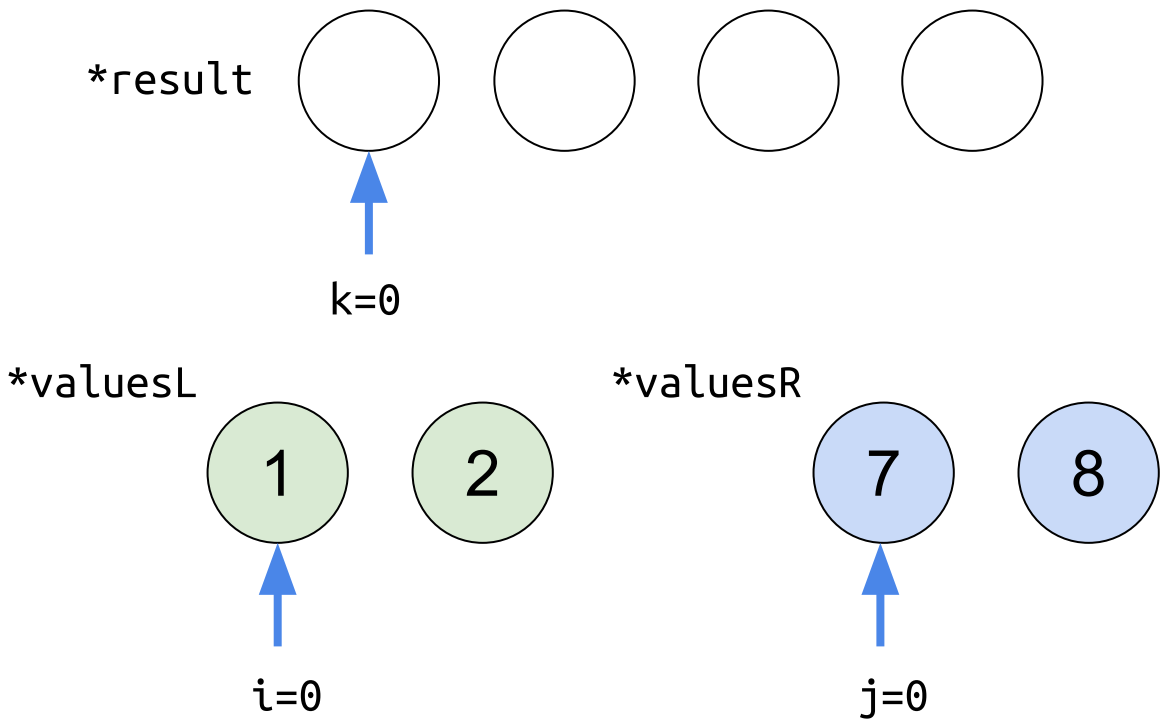


Merge - Código

- Copia o menor de cada lado
- Até não ter mais de onde copiar de 1 dos lados
- Podemos ficar com elementos restantes

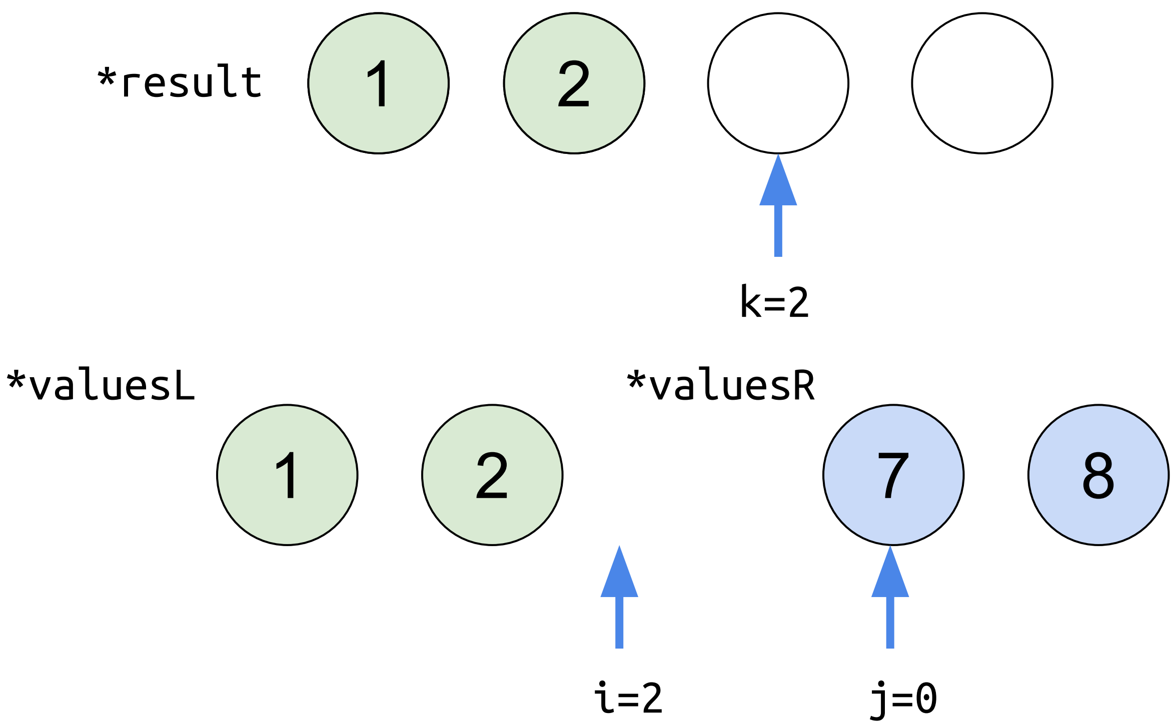
```
while (i < nl && j < nr) {
   if (valuesL[i] < valuesR[j]) {
      result[k] = valuesL[i];
      i++;
   } else {
      result[k] = valuesR[j];
      j++;
   }
   k++;
}
```

Merge – E se sobrar elementos de um lado?

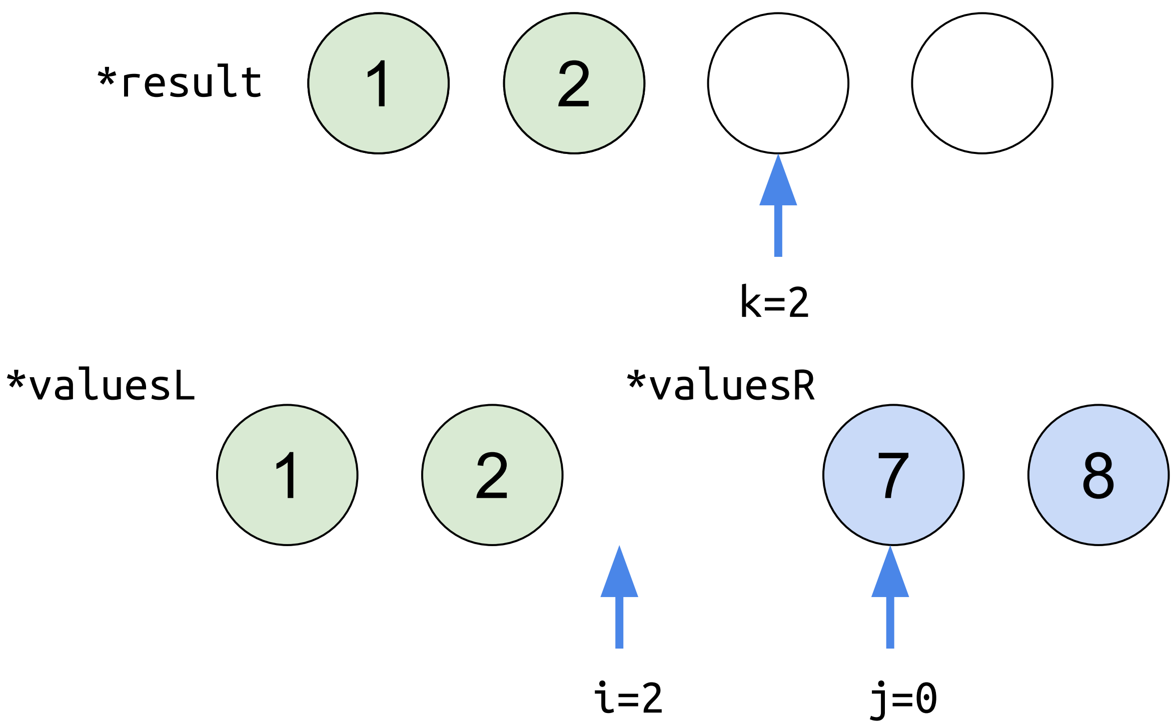


Merge – E se sobrar elementos de um lado?

```
if (k < nl + nr) {
  for(; i < nl; i++) {
    result[k] = valuesL[i];
    k++;
  }
  for(; j < nr; j++) {
    result[k] = valuesR[j];
    k++;
  }
}
```



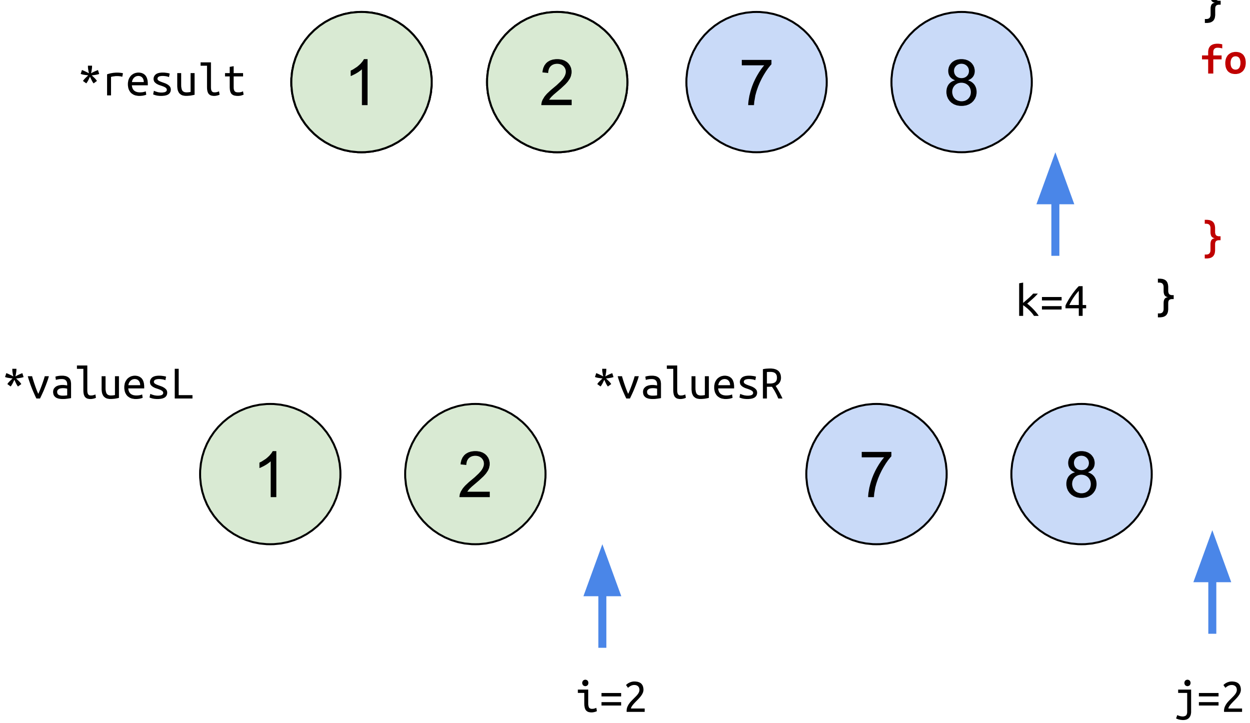
Merge – E se sobrar elementos de um lado?




```
if (k < nl + nr) {
  for(; i < nl; i++) {
    result[k] = valuesL[i];
    k++;
  }
  for(; j < nr; j++) {
    result[k] = valuesR[j];
    k++;
  }
}
```

Merge – E se sobrar elementos de um lado?

```
if (k < nl + nr) {
  for(; i < nl; i++) {
    result[k] = valuesL[i];
    k++;
  }
  for(; j < nr; j++) {
    result[k] = valuesR[j];
    k++;
  }
}
```



Função Merge

```c
int *merge(int *valuesL, int
*valuesR, int nl, int nr) {
  int *result = (int *)
malloc((nl+nr) * sizeof(int));
  int i = 0;
  int j = 0;
  int k = 0;
  while (i < nl && j < nr) {
    if (valuesL[i] < valuesR[j]) {
      result[k] = valuesL[i];
      i++;
    } else {
      result[k] = valuesR[j];
      j++;
    }
    k++;
  }
if (k < nl + nr) {
    for(; i < nl; i++) {
      result[k] = valuesL[i];
      k++;
    }
    for(; j < nr; j++) {
      result[k] = valuesR[j];
      k++;
    }
  }
  return result;
}
```

Visualização do passo a passo: http://pythontutor.com/c.html#mode=edit

MERGESORT - ANÁLISE

Mergesort - análise empírica

Estimativas de tempo de execução:

Laptop: **10^8** comparações/sec

Supercomputador: **10^{12}** comparações/sec

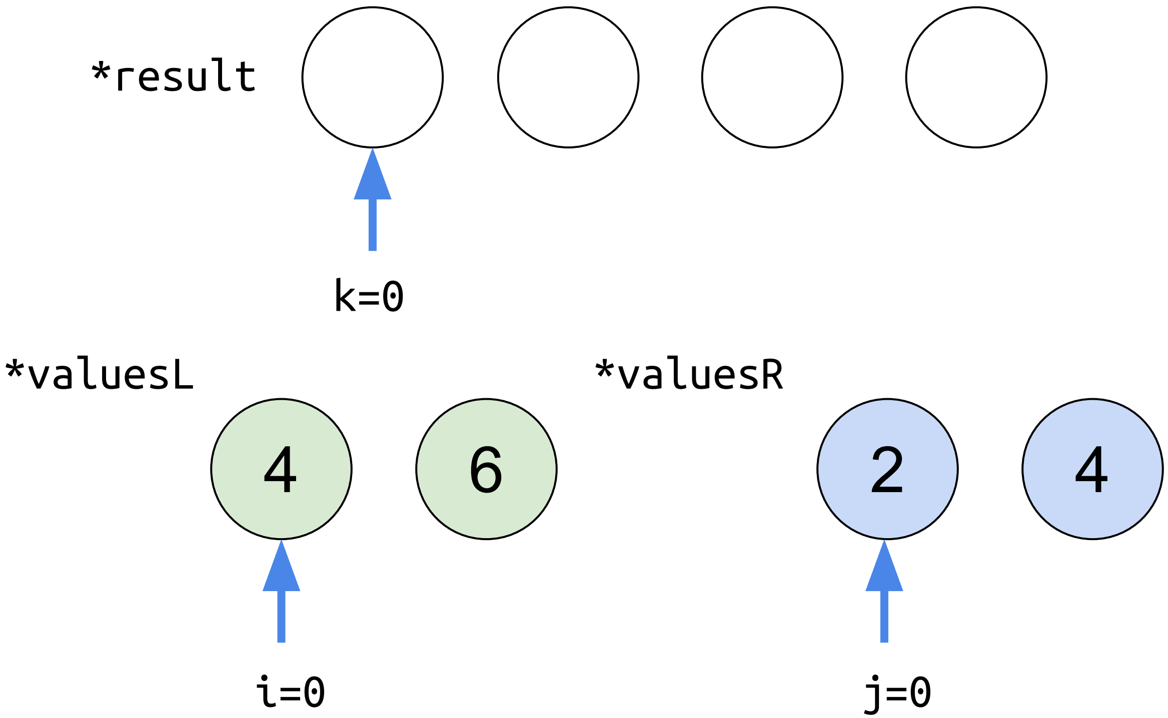
	insertion sort (N^2)			mergesort (N log N)		
computer	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min
super	instant	1 second	1 week	instant	instant	instant

Algoritmos eficientes são melhores que supercomputadores!

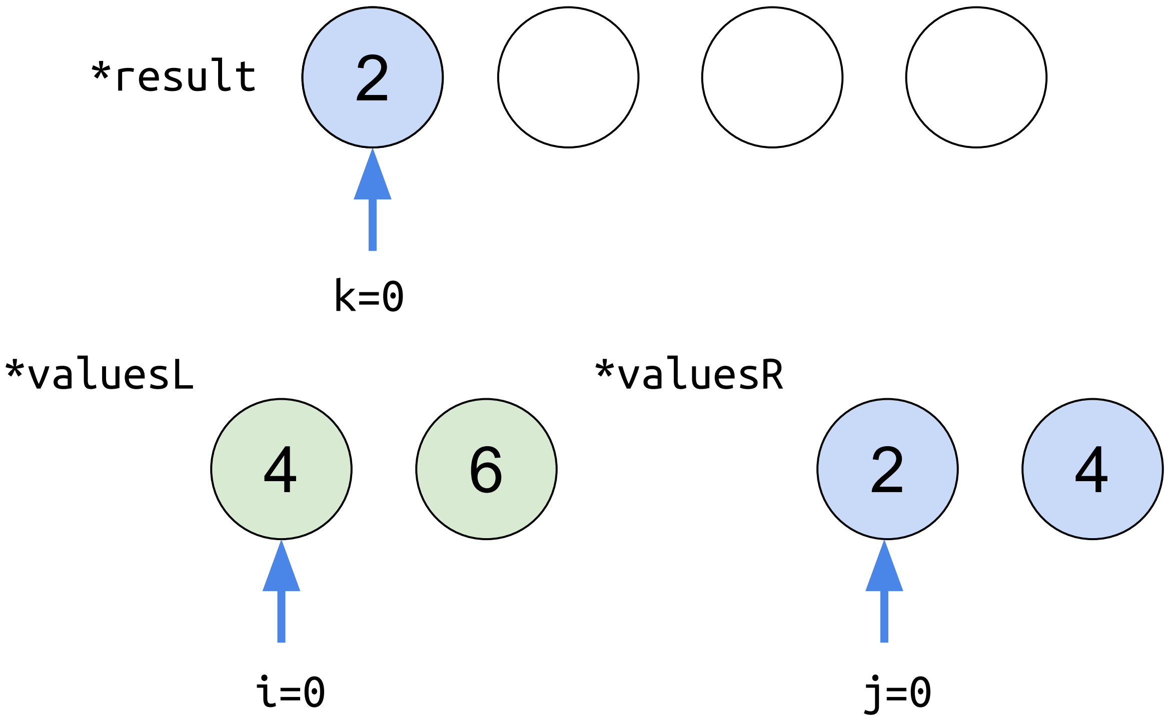
O método é estável?

Vamos ver um exemplo...

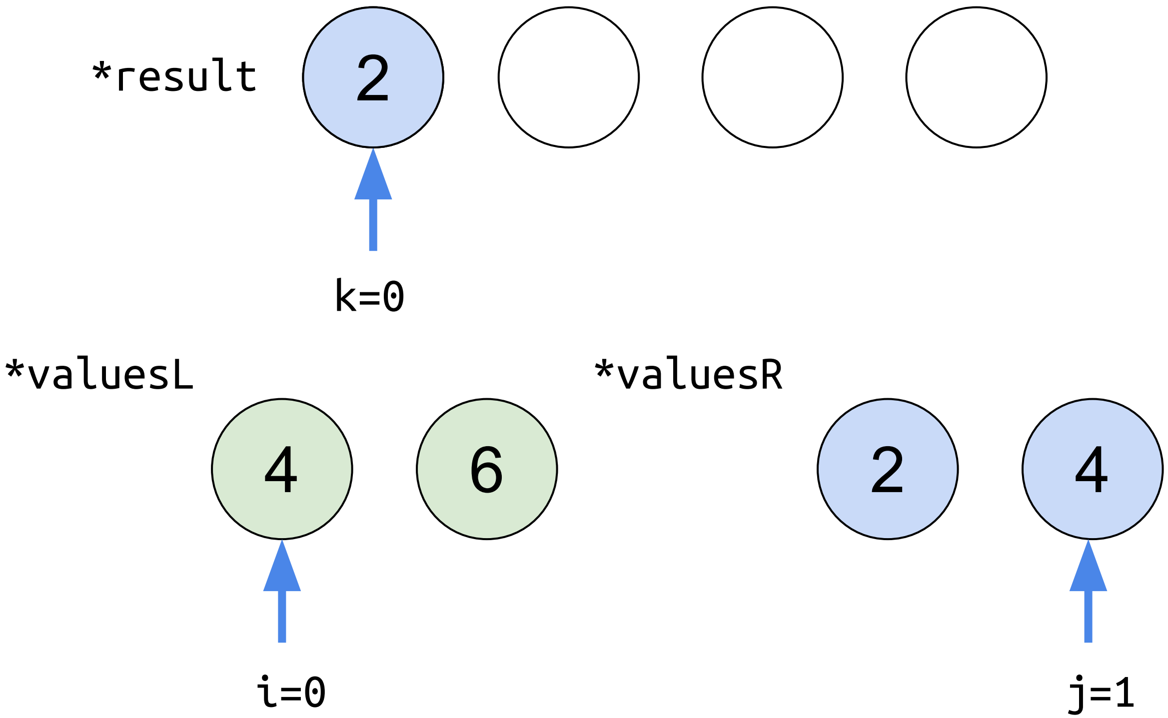
Merge



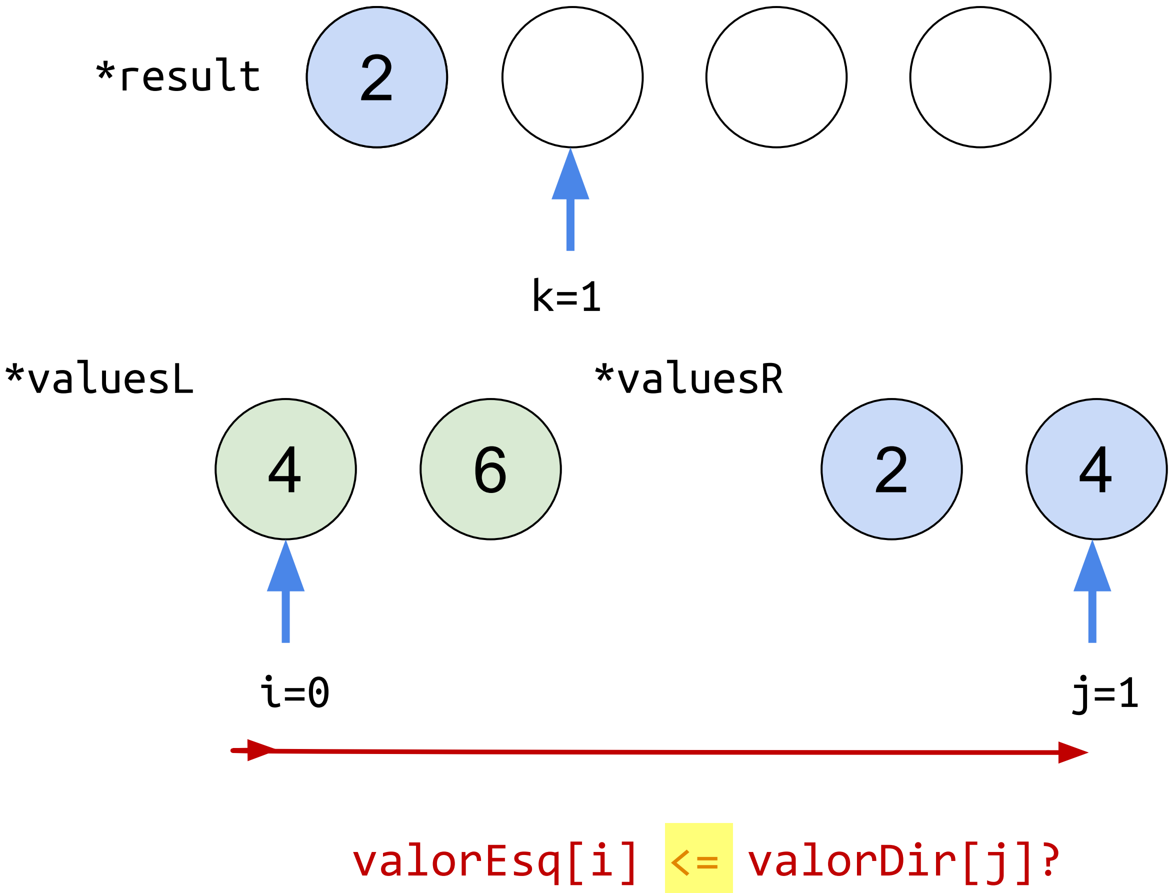
Merge



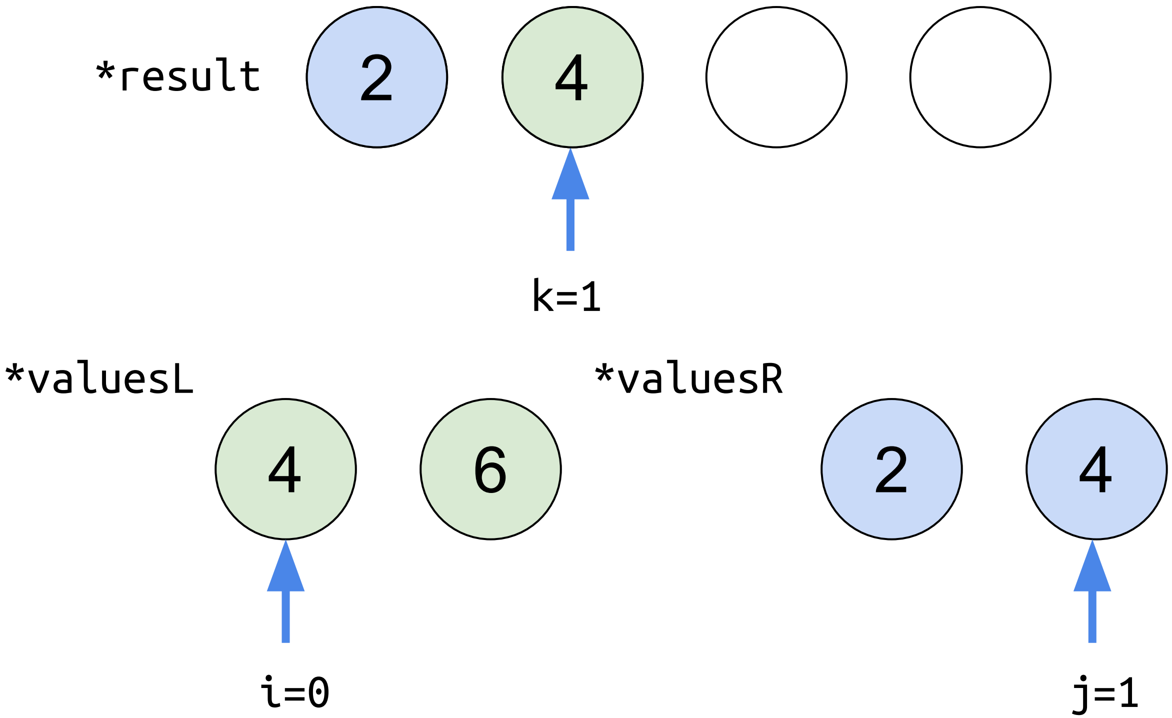
Merge



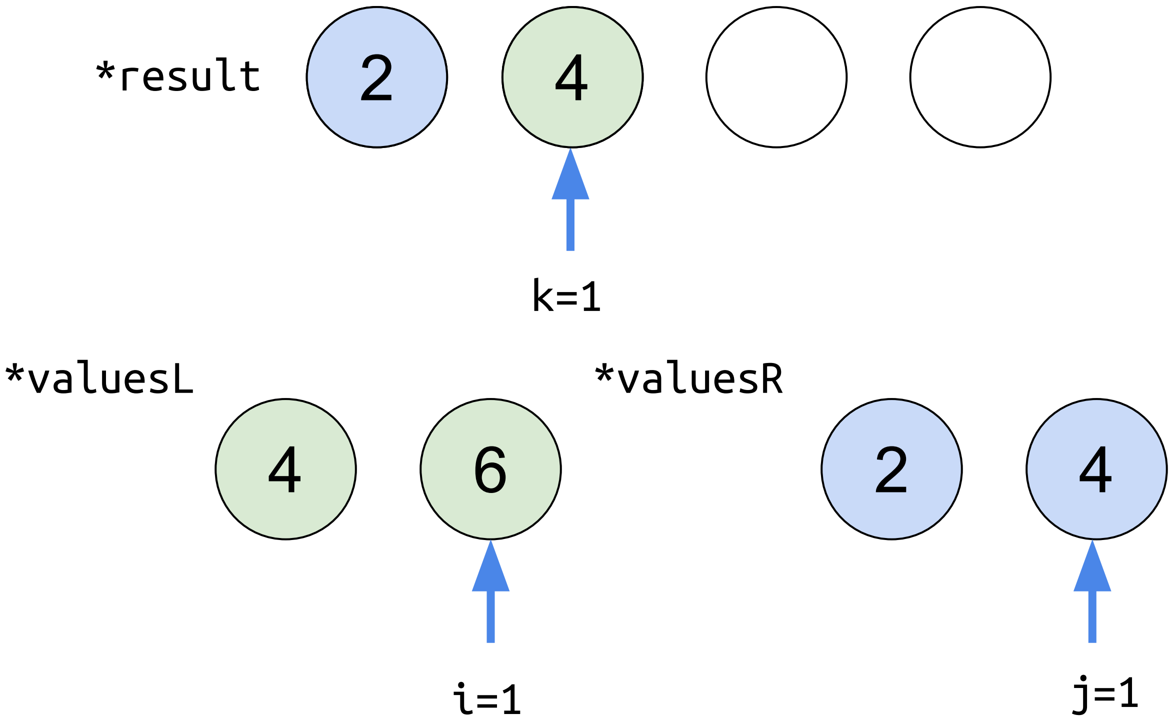
Merge



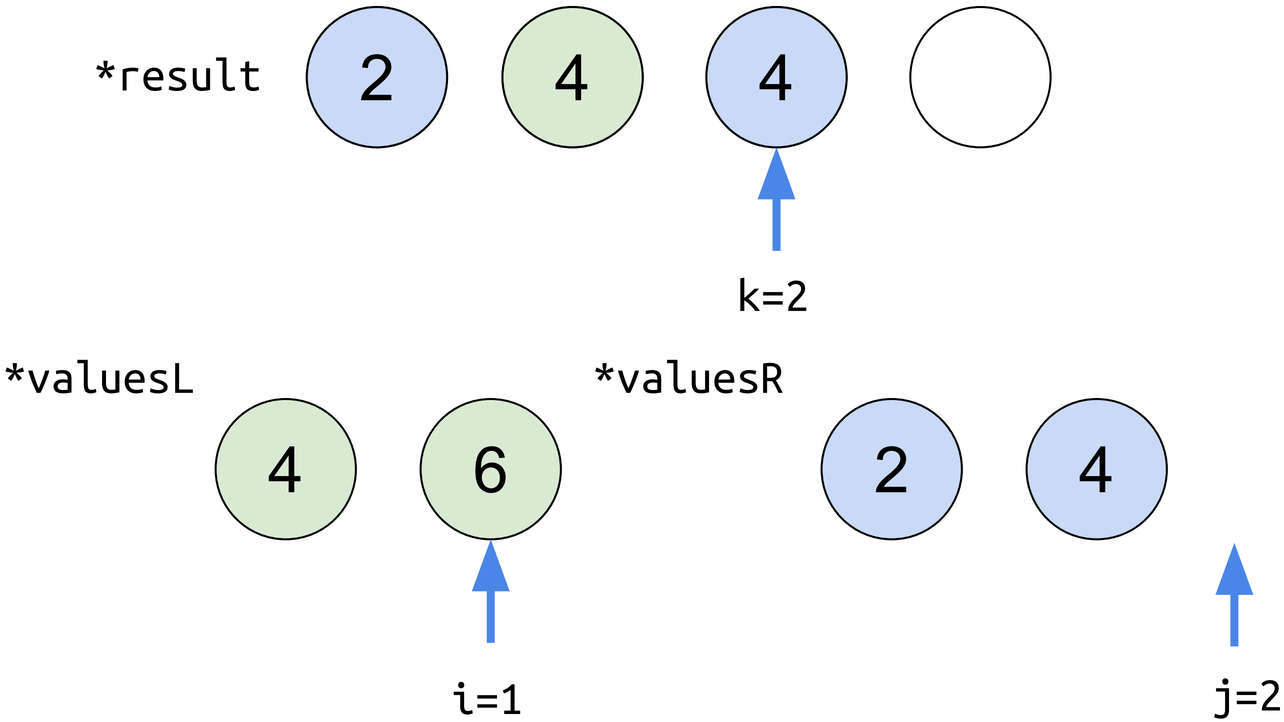
Merge



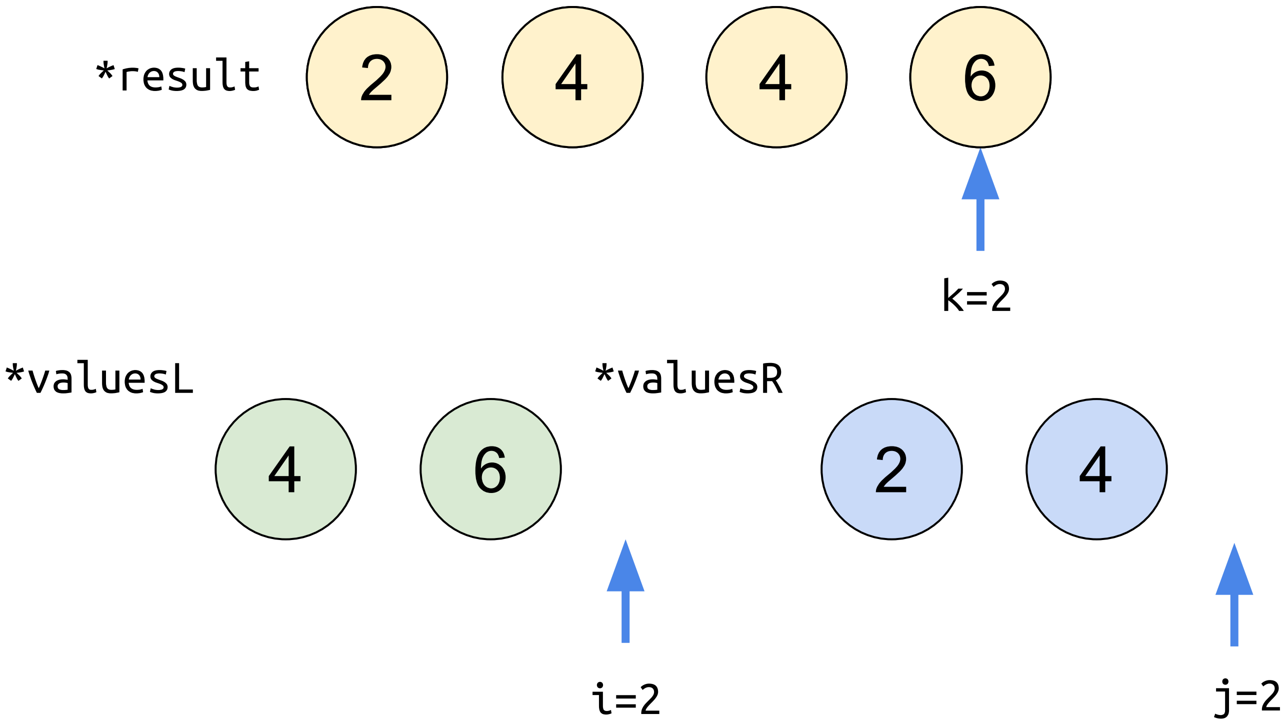
Merge



Merge



Merge



Mergesort - Características

- O algoritmo é **estável**?
  - Sim, pois se os elementos forem iguais, eles não são trocados de ordem

- O algoritmo é **adaptável**?
  - Não, ele executa o mesmo número de comparações, independente da entrada.

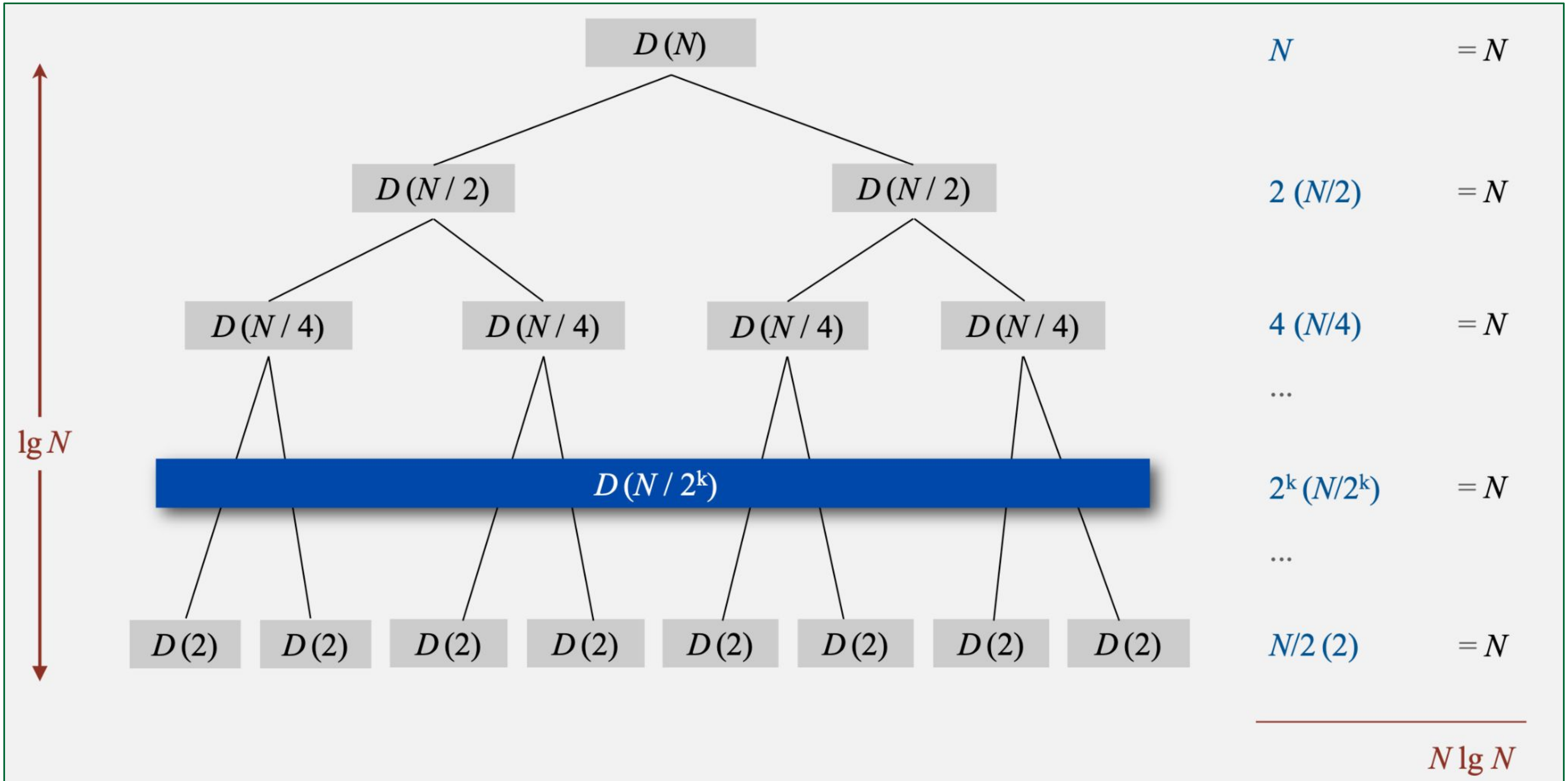
MergeSort – Análise de Complexidade

- **`MergeSort`**
  - Para n = 1, T(1) = 0
  - Para n > 1,
    - Uma vez para n/2 elementos
    - Uma vez para n/2 elementos

2 vezes

Análise de Complexidade

Assuma N sendo uma potência de 2, D(1) = 0



MergeSort – Análise de Complexidade

- **MergeSort**
  - Para n = 1, T(1) = 0
  - Para n > 1,
    - Uma vez para n/2 elementos
    - Uma vez para n/2 elementos

2 vezes

- **Operação de Merge**
  - Custo linear: n

- Logo: T(n) = 2T(n/2) + n = *O(n log n)*

Com o Teorema Mestre

- Formato da Equação de Recorrência:

$$T(n) = aT(n/b) + f(n)$$

onde $a \geq 1, b > 1$ e $f(n)$ positiva

- Para: T(n) = 2T(n/2) + n

Qual caso se aplica?

1. $T(n) = \Theta(n^{\log_b a})$, se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$,
2. $T(n) = \Theta(n^{\log_b a} \log n)$, se $f(n) = \Theta(n^{\log_b a})$,
3. $T(n) = \Theta(f(n))$, se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguma constante $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e todo n a partir de um valor suficientemente grande.

Com o Teorema Mestre

- Formato da Equação de Recorrência:

$$T(n) = aT(n/b) + f(n)$$

onde $a \geq 1, b > 1$ e $f(n)$ positiva

- Para: T(n) = 2T(n/2) + n

Temos: $a = 2$, $b = 2$, $f(n) = n$ e $n^{\log_b a} = n^{\log_2 2} = n$

Caso 2:

$$T(n) = \Theta(n^{\log_b a} \log n), \text{ se } f(n) = \Theta(n^{\log_b a})$$

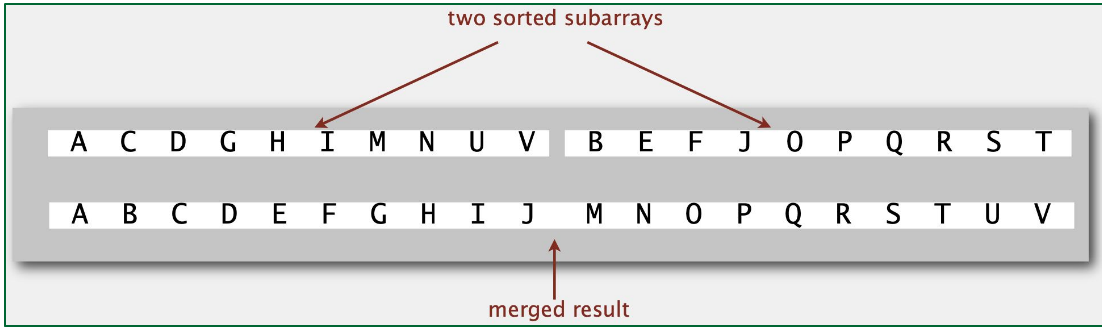
O caso 2 se aplica porque, $f(n) = \Theta(n^{\log_b a}) = \Theta(n)$, *assim*, $T(n) = \Theta(n \log n)$

Mergesort: memória

Mergesort usa espaço extra de memória

vetor `result`

proporcional a N



Def. Um algoritmo de ordenação é **in-place** se ele usa `<= c log N` memória extra

Ex: insertion sort, selection sort, shellsort

Desafio: Implementar mergesort **in-place** [Kronrod, 1969].

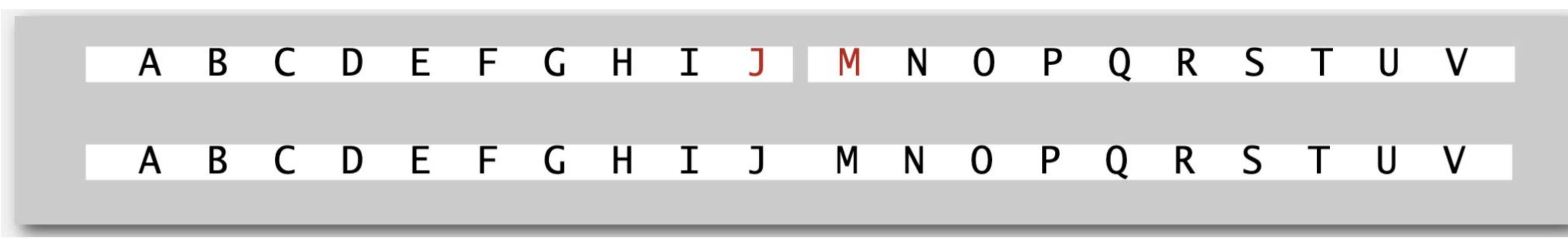
Mergesort: melhorias práticas

Usar o Insertion sort para pequenos sub-vetores

Cutoff aprox. 7 itens

Parar se o vetor já está ordenado

O maior elemento da primeira metade é <= menor elemento da segunda metade?



Complexidade da ordenação

Complexidade da ordenação

Complexidade computacional. Arcabouço para estudar a eficiência de resolver um problema particular X

Problema: ordenação

Complexidade da ordenação

Modelo computacional: operações permitidas

Modelo de custo: contador de operações

Limite superior: garantia de custo de **algum** alg. para X

Limite inferior: limite de custo provado de **todo** alg. para X

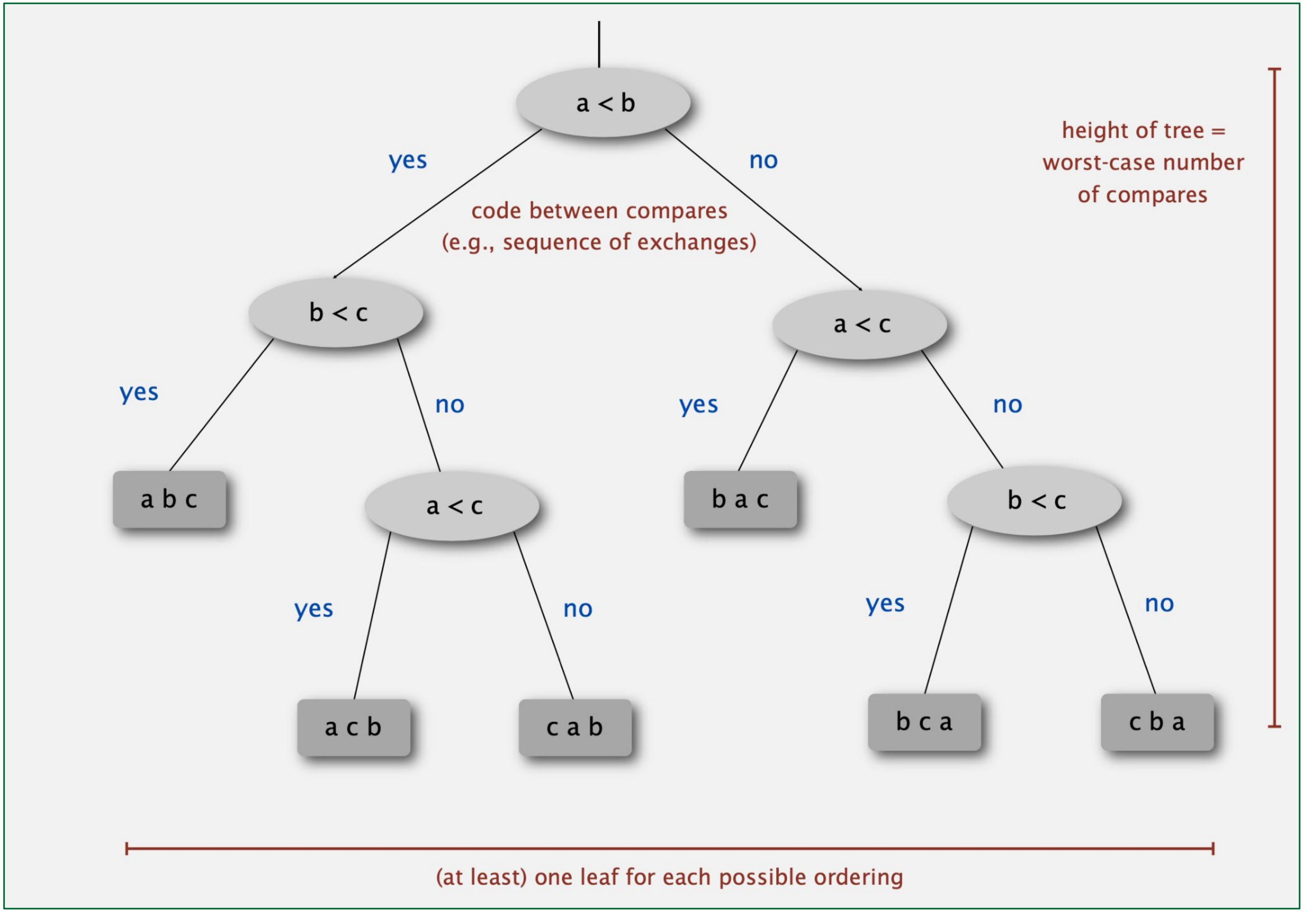
Algoritmo ótimo: algoritmo com o melhor custo possível para X

lower bound ~ upper bound

Ex: ordenação

- Modelo computacional: árvore de decisão
- Modelo de custo: # comparações
- Limite superior: ~ N lg N from mergesort
- Limite inferior: ?
- Algoritmo ótimo: ?

Árvore de decisão (itens a, b, c)



Limite inferior da ordenação por comparação de chaves

Proposição. Qualquer algoritmo de ordenação por comparação de chaves usa pelo menos log(N!) ~ N lg N comparações no pior-caso

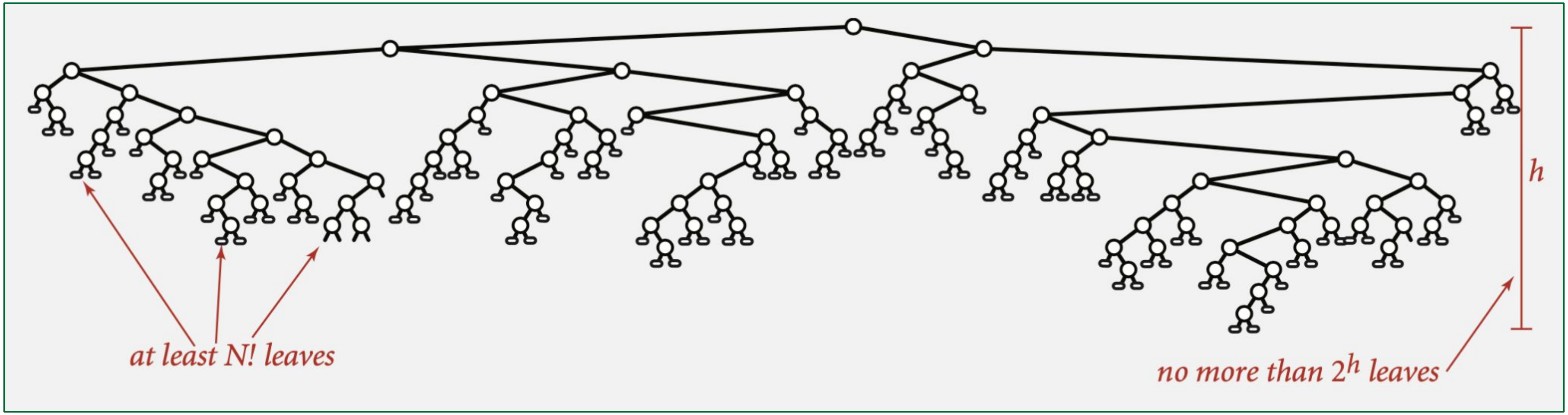
Prova.

- Assuma N valores distintos $a_1 \ldots a_N$
- No pior caso a *altura* da árvore de decisão é *h*
- Uma árvore binária de altura *h* tem no máximo *2h* folhas
- N! diferentes ordenações => pelo menos N! folhas

Limite inferior da ordenação por comparação de chaves

Prova.

- Assuma N valores distintos $a_1 \ldots a_N$
- No pior caso a *altura* da árvore de decisão é *h*
- Uma árvore binária de altura *h* tem no máximo *2h* folhas
- N! diferentes ordenações => pelo menos N! folhas



Limite inferior da ordenação por comparação de chaves

Prova.

- Assuma N valores distintos $a_1 \ldots a_N$
- No pior caso a *altura* da árvore de decisão é *h*
- Uma árvore binária de altura *h* tem no máximo *2h* folhas
- N! diferentes ordenações => pelo menos N! folhas

$$2^h \geq \#\,\text{leaves} \geq N!$$

$$\Rightarrow\ h \geq \lg(N!) \sim N \lg N$$

Stirling's formula

Complexidade da ordenação

Modelo computacional: operações permitidas
Modelo de custo: contador de operações
Limite superior: garantia de custo de **algum** alg. para X
Limite inferior: limite de custo provado de **todo** alg. para X
Algoritmo ótimo: algoritmo com o melhor custo possível para X

lower bound ~ upper bound

Ex: ordenação

- Modelo computacional: árvore de decisão
- Modelo de custo: # comparações
- Limite superior: ~ N lg N from mergesort
- Limite inferior: ~ N lg N
- **Algoritmo ótimo: mergesort**

Complexidade da ordenação

Limite inferior não é válido se o algoritmo usa informação extra

- Ordenação inicial da entrada
  - Entrada ordenada: Insertion sort n-1 comparações
- Distribuição das chaves
  - Duplicatas: 3-way quicksort
- Representação das chaves
  - Dígitos/characters: radix sort

Mergesort: Considerações Finais

- Vantagens
  - Deve ser considerado quando alto custo de pior caso não pode ser tolerável.
    - É ótimo no número de comparações
- Desvantagens
  - Requer espaço extra proporcional a n.
    - Não é ótimo em complexidade de espaço
- Comumente adaptado para ordenação em memória secundária