

MC-202 — Unidade 11

Intercalação e Ordenação por Intercalação

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

2º semestre/2017

Na unidade anterior...

Vimos três algoritmos de ordenação:

Na unidade anterior...

Vimos três algoritmos de ordenação:

- `selectionsort`

Na unidade anterior...

Vimos três algoritmos de ordenação:

- `selectionsort`
- `bubblesort`

Na unidade anterior...

Vimos três algoritmos de ordenação:

- `selectionsort`
- `bubblesort`
- `insertionsort`

Na unidade anterior...

Vimos três algoritmos de ordenação:

- `selectionsort`
- `bubblesort`
- `insertionsort`

Apesar do `insertionsort` ser melhor na prática, os três algoritmos são $O(n^2)$

Na unidade anterior...

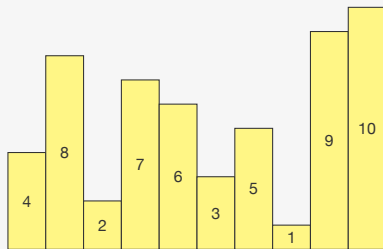
Vimos três algoritmos de ordenação:

- `selectionsort`
- `bubblesort`
- `insertionsort`

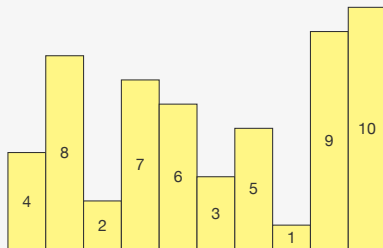
Apesar do `insertionsort` ser melhor na prática, os três algoritmos são $O(n^2)$

Nessa unidade veremos um algoritmo $O(n \log n)$

Estratégia: recursão

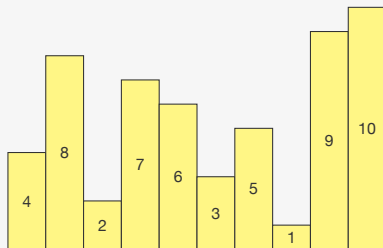


Estratégia: recursão



Como ordenar a primeira metade do vetor?

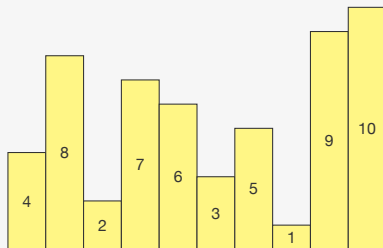
Estratégia: recursão



Como ordenar a primeira metade do vetor?

- usamos uma função `ordenar(int *v, int l, int r)`

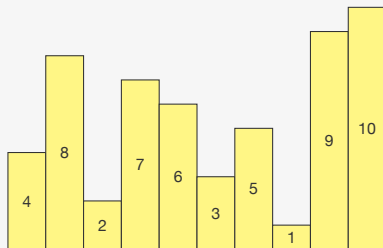
Estratégia: recursão



Como ordenar a primeira metade do vetor?

- usamos uma função `ordenar(int *v, int l, int r)`
 - poderia ser `bubblesort`, `selectionsort` ou `insertionsort`

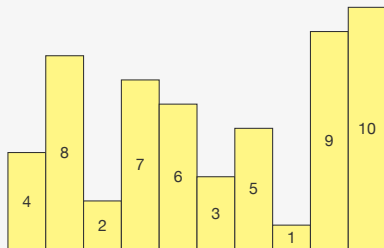
Estratégia: recursão



Como ordenar a primeira metade do vetor?

- usamos uma função `ordenar(int *v, int l, int r)`
 - poderia ser `bubblesort`, `selectionsort` ou `insertionsort`
 - mas vamos fazer algo melhor do que isso

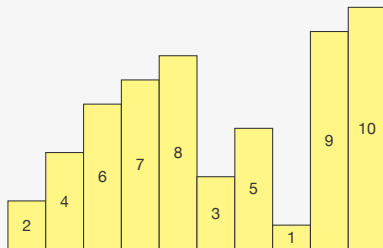
Estratégia: recursão



Como ordenar a primeira metade do vetor?

- usamos uma função `ordenar(int *v, int l, int r)`
 - poderia ser `bubblesort`, `selectionsort` ou `insertionsort`
 - mas vamos fazer algo melhor do que isso
- executamos `ordenar(v, 0, 4);`

Estratégia: recursão

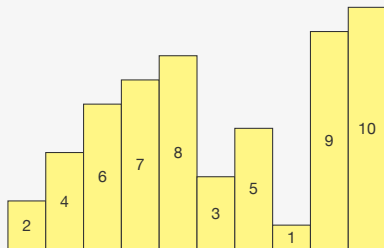


Como ordenar a primeira metade do vetor?

- usamos uma função `ordenar(int *v, int l, int r)`
 - poderia ser `bubblesort`, `selectionsort` ou `insertionsort`
 - mas vamos fazer algo melhor do que isso
- executamos `ordenar(v, 0, 4);`

E se quiséssemos ordenar a segunda parte?

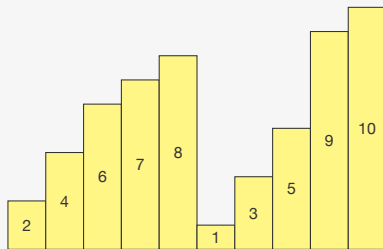
Ordenando a segunda parte



Para ordenar a segunda metade:

- executamos `ordenar(v, 5, 9);`

Ordenando a segunda parte



Para ordenar a segunda metade:

- executamos `ordenar(v, 5, 9);`

Ordenando todo o vetor

Suponha que temos um vetor com as suas duas metades já ordenadas

Ordenando todo o vetor

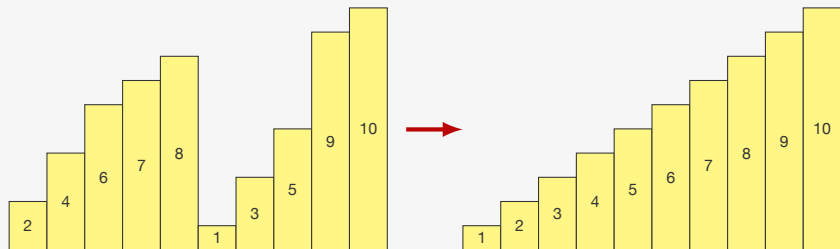
Suponha que temos um vetor com as suas duas metades já ordenadas

- Como ordenar todo o vetor?

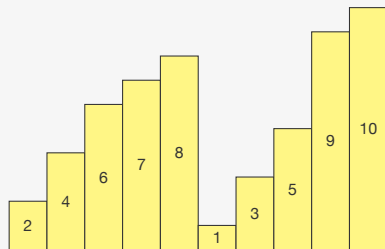
Ordenando todo o vetor

Suponha que temos um vetor com as suas duas metades já ordenadas

- Como ordenar todo o vetor?

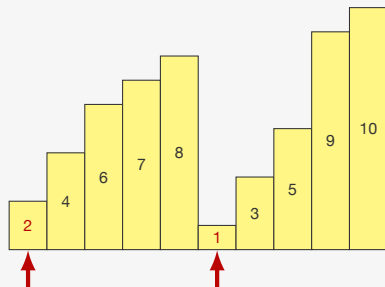


Intercalando



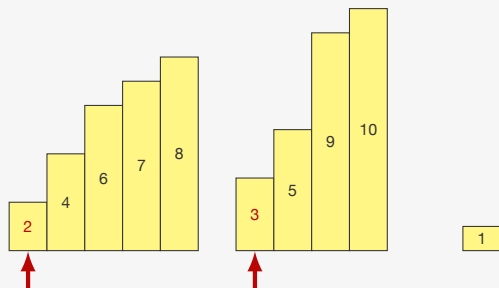
- Percorreremos os dois subvetores

Intercalando



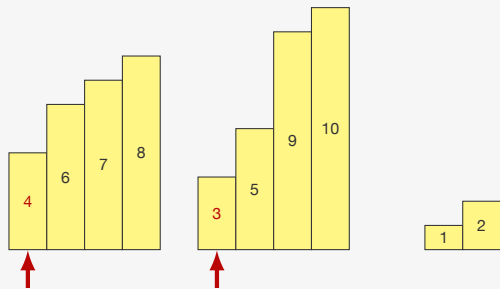
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

Intercalando



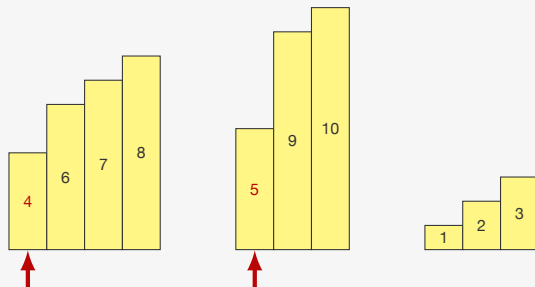
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

Intercalando



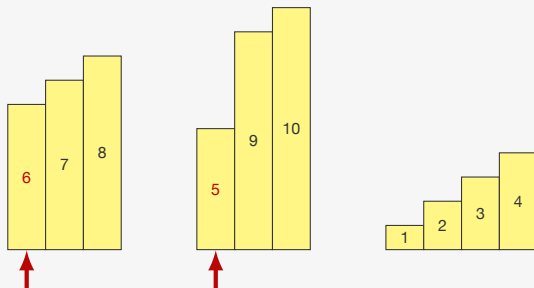
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

Intercalando



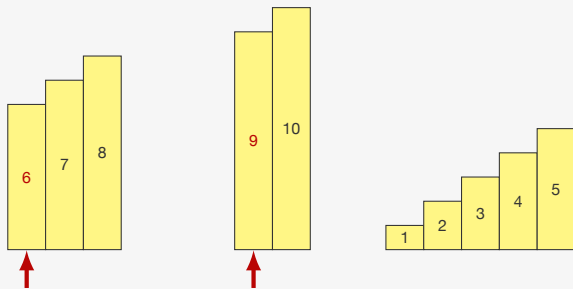
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

Intercalando



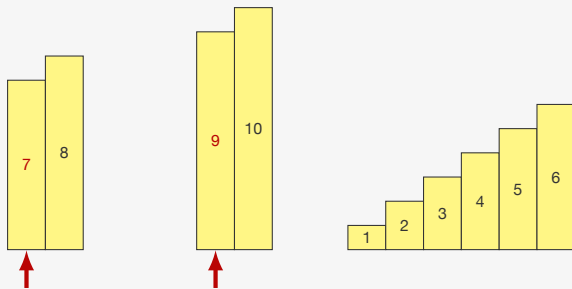
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

Intercalando



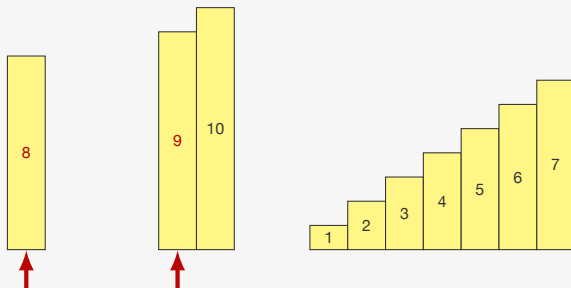
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

Intercalando



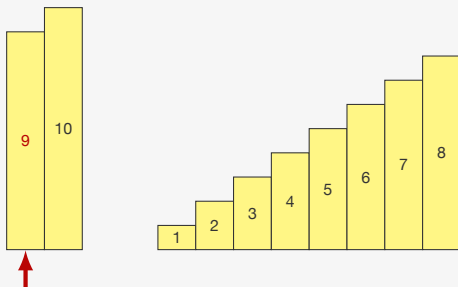
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

Intercalando



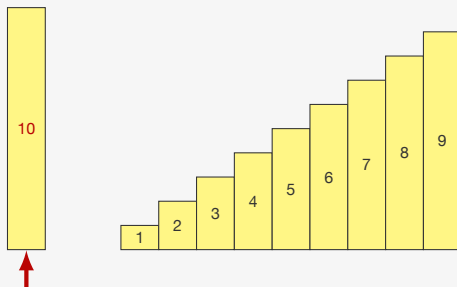
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar

Intercalando



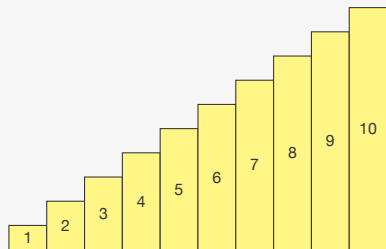
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar
- Depois copiamos o restante

Intercalando



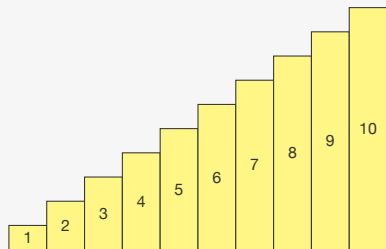
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar
- Depois copiamos o restante

Intercalando



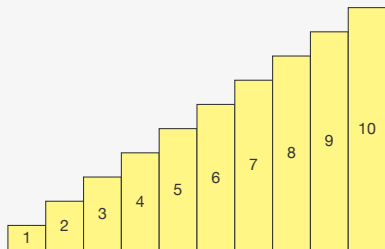
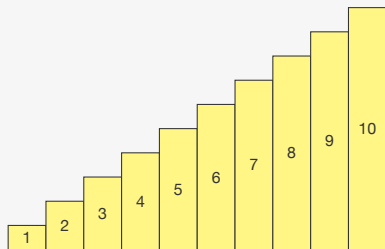
- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar
- Depois copiamos o restante

Intercalando



- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar
- Depois copiamos o restante
- No final, copiamos do vetor auxiliar para o original

Intercalando



- Percorremos os dois subvetores
- Pegamos o **mínimo** e inserimos em um vetor auxiliar
- Depois copiamos o restante
- No final, copiamos do vetor auxiliar para o original

Divisão e conquista

Observação:

Divisão e conquista

Observação:

- A recursão parte do princípio que é mais fácil resolver problemas menores

Divisão e conquista

Observação:

- A recursão parte do princípio que é mais fácil resolver problemas menores
- Para certos problemas, podemos dividi-lo em duas ou mais partes

Divisão e conquista

Observação:

- A recursão parte do princípio que é mais fácil resolver problemas menores
- Para certos problemas, podemos dividi-lo em duas ou mais partes

Divisão e conquista:

Divisão e conquista

Observação:

- A recursão parte do princípio que é mais fácil resolver problemas menores
- Para certos problemas, podemos dividi-lo em duas ou mais partes

Divisão e conquista:

- **Divisão:** Quebramos o problema em vários subproblemas menores

Divisão e conquista

Observação:

- A recursão parte do princípio que é mais fácil resolver problemas menores
- Para certos problemas, podemos dividi-lo em duas ou mais partes

Divisão e conquista:

- **Divisão:** Quebramos o problema em vários subproblemas menores
 - ex: quebramos um vetor a ser ordenado em dois

Divisão e conquista

Observação:

- A recursão parte do princípio que é mais fácil resolver problemas menores
- Para certos problemas, podemos dividi-lo em duas ou mais partes

Divisão e conquista:

- **Divisão:** Quebramos o problema em vários subproblemas menores
 - ex: quebramos um vetor a ser ordenado em dois
- **Conquista:** Combinamos a solução dos problemas menores

Divisão e conquista

Observação:

- A recursão parte do princípio que é mais fácil resolver problemas menores
- Para certos problemas, podemos dividi-lo em duas ou mais partes

Divisão e conquista:

- **Divisão:** Quebramos o problema em vários subproblemas menores
 - ex: quebramos um vetor a ser ordenado em dois
- **Conquista:** Combinamos a solução dos problemas menores
 - ex: intercalamos os dois vetores ordenados

Ordenação por intercalação (*MergeSort*)

Intercalação:

Ordenação por intercalação (*MergeSort*)

Intercalação:

- Os dois subvetores estão armazenados em v :

Ordenação por intercalação (*MergeSort*)

Intercalação:

- Os dois subvetores estão armazenados em v :
 - O primeiro nas posições de l até m

Ordenação por intercalação (*MergeSort*)

Intercalação:

- Os dois subvetores estão armazenados em v :
 - O primeiro nas posições de l até m
 - O segundo nas posições de $m + 1$ até r

Ordenação por intercalação (*MergeSort*)

Intercalação:

- Os dois subvetores estão armazenados em v :
 - O primeiro nas posições de l até m
 - O segundo nas posições de $m + 1$ até r
- Precisamos de um vetor auxiliar do tamanho do vetor

Ordenação por intercalação (*MergeSort*)

Intercalação:

- Os dois subvetores estão armazenados em v :
 - O primeiro nas posições de l até m
 - O segundo nas posições de $m + 1$ até r
- Precisamos de um vetor auxiliar do tamanho do vetor
- Vamos considerar que o maior vetor tem tamanho MAX

Ordenação por intercalação (*MergeSort*)

Intercalação:

- Os dois subvetores estão armazenados em **v**:
 - O primeiro nas posições de **l** até **m**
 - O segundo nas posições de **m + 1** até **r**
- Precisamos de um vetor auxiliar do tamanho do vetor
- Vamos considerar que o maior vetor tem tamanho **MAX**
 - Exemplo `#define MAX 100`

Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
```

Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {  
2     int aux[MAX];  
3     int i = l, j = m + 1, k = 0;
```

Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     //intercala
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
```

Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     //intercala
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
10    //copia o resto do subvetor que não terminou
11    while (i <= m)
12        aux[k++] = v[i++];
```

Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     //intercala
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
10    //copia o resto do subvetor que não terminou
11    while (i <= m)
12        aux[k++] = v[i++];
13    while (j <= r)
14        aux[k++] = v[j++];
```

Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     //intercala
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
10    //copia o resto do subvetor que não terminou
11    while (i <= m)
12        aux[k++] = v[i++];
13    while (j <= r)
14        aux[k++] = v[j++];
15    //copia de volta para v
16    for (i = l, k=0; i <= r; i++, k++)
17        v[i] = aux[k];
18 }
```

Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     //intercala
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
10    //copia o resto do subvetor que não terminou
11    while (i <= m)
12        aux[k++] = v[i++];
13    while (j <= r)
14        aux[k++] = v[j++];
15    //copia de volta para v
16    for (i = l, k=0; i <= r; i++, k++)
17        v[i] = aux[k];
18 }
```

Quantas comparações são feitas?

Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     //intercala
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
10    //copia o resto do subvetor que não terminou
11    while (i <= m)
12        aux[k++] = v[i++];
13    while (j <= r)
14        aux[k++] = v[j++];
15    //copia de volta para v
16    for (i = l, k=0; i <= r; i++, k++)
17        v[i] = aux[k];
18 }
```

Quantas comparações são feitas?

- a cada passo, aumentamos um em *i* ou em *j*

Ordenação por intercalação (*MergeSort*)

```
1 void merge(int *v, int l, int m, int r) {
2     int aux[MAX];
3     int i = l, j = m + 1, k = 0;
4     //intercala
5     while (i <= m && j <= r)
6         if (v[i] <= v[j])
7             aux[k++] = v[i++];
8         else
9             aux[k++] = v[j++];
10    //copia o resto do subvetor que não terminou
11    while (i <= m)
12        aux[k++] = v[i++];
13    while (j <= r)
14        aux[k++] = v[j++];
15    //copia de volta para v
16    for (i = l, k=0; i <= r; i++, k++)
17        v[i] = aux[k];
18 }
```

Quantas comparações são feitas?

- a cada passo, aumentamos um em i ou em j
- no máximo $n := r - l + 1$

Ordenação por intercalação (*MergeSort*)

Ordenação:

Ordenação por intercalação (*MergeSort*)

Ordenação:

- Recebemos um **vetor** de tamanho n com limites:

Ordenação por intercalação (*MergeSort*)

Ordenação:

- Recebemos um **vetor** de tamanho n com limites:
 - O vetor começa na posição **vetor[1]**

Ordenação por intercalação (*MergeSort*)

Ordenação:

- Recebemos um **vetor** de tamanho n com limites:
 - O vetor começa na posição **vetor[l]**
 - O vetor termina na posição **vetor[r]**

Ordenação por intercalação (*MergeSort*)

Ordenação:

- Recebemos um **vetor** de tamanho n com limites:
 - O vetor começa na posição **vetor**[1]
 - O vetor termina na posição **vetor**[r]
- Dividimos o vetor em dois subvetores de tamanho $n/2$

Ordenação por intercalação (*MergeSort*)

Ordenação:

- Recebemos um **vetor** de tamanho n com limites:
 - O vetor começa na posição **vetor[l]**
 - O vetor termina na posição **vetor[r]**
- Dividimos o vetor em dois subvetores de tamanho $n/2$
- O caso base é um vetor de tamanho **0** ou **1**

Ordenação por intercalação (*MergeSort*)

Ordenação:

- Recebemos um **vetor** de tamanho n com limites:
 - O vetor começa na posição **vetor[l]**
 - O vetor termina na posição **vetor[r]**
- Dividimos o vetor em dois subvetores de tamanho $n/2$
- O caso base é um vetor de tamanho **0** ou **1**

Ordenação por intercalação (*MergeSort*)

Ordenação:

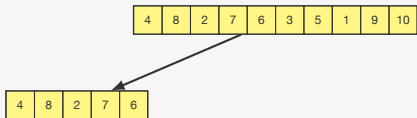
- Recebemos um **vetor** de tamanho n com limites:
 - O vetor começa na posição **vetor[l]**
 - O vetor termina na posição **vetor[r]**
- Dividimos o vetor em dois subvetores de tamanho $n/2$
- O caso base é um vetor de tamanho **0** ou **1**

```
1 void mergesort(int *v, int l, int r) {
2     int m = (l + r) / 2;
3     if (l < r) {
4         //divisão
5         mergesort(v, l, m);
6         mergesort(v, m + 1, r);
7         //conquista
8         merge(v, l, m, r);
9     }
10 }
```

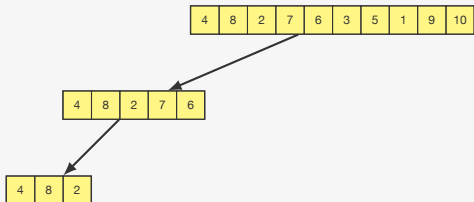
Simulação

4	8	2	7	6	3	5	1	9	10
---	---	---	---	---	---	---	---	---	----

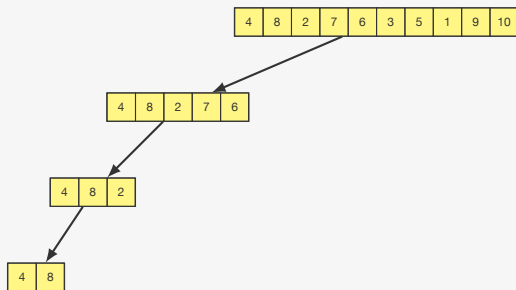
Simulação



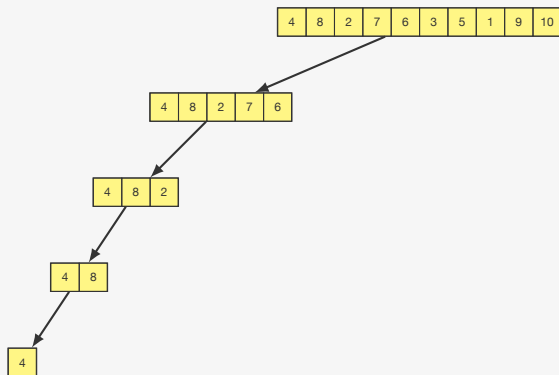
Simulação



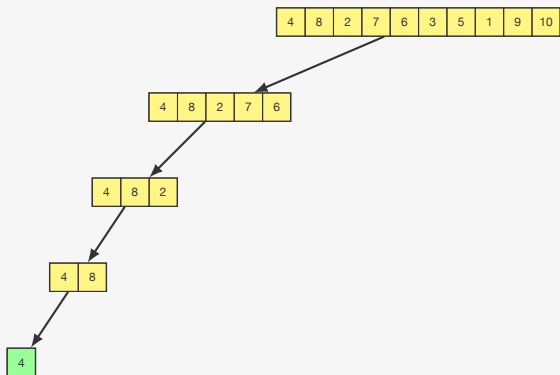
Simulação



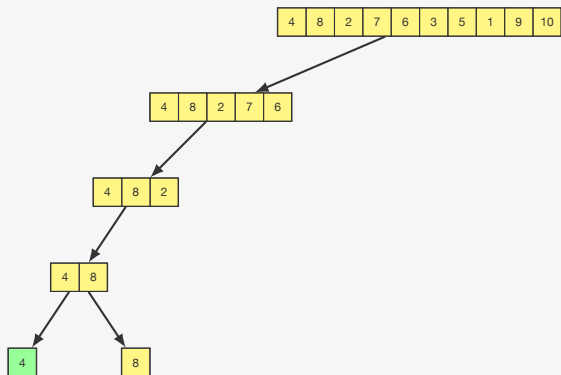
Simulação



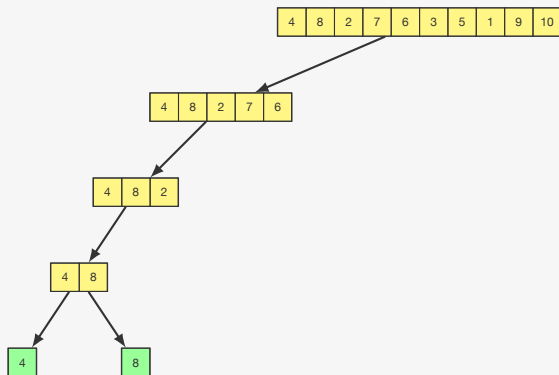
Simulação



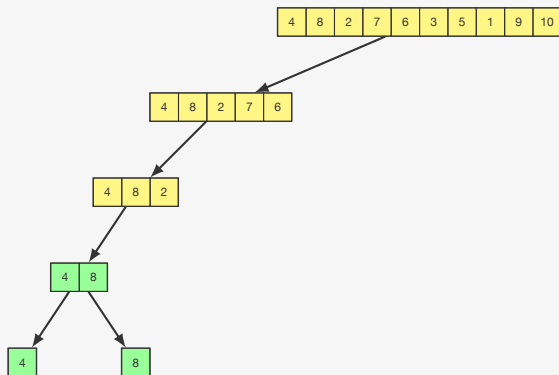
Simulação



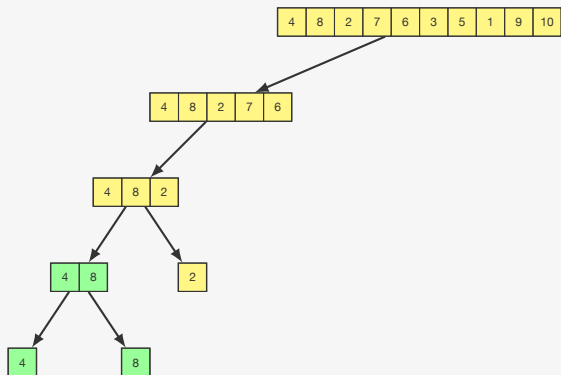
Simulação



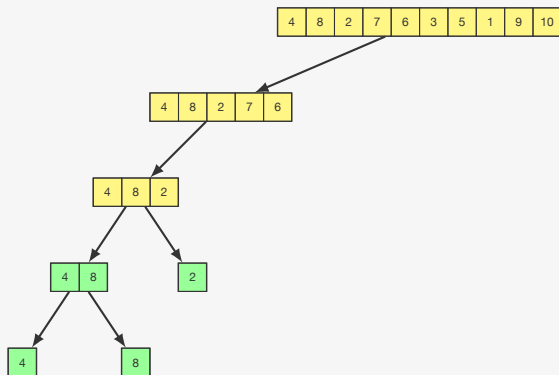
Simulação



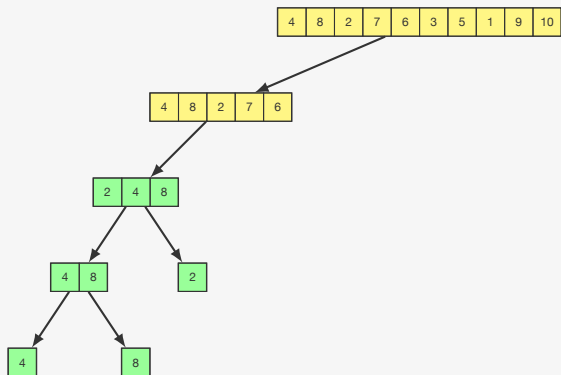
Simulação



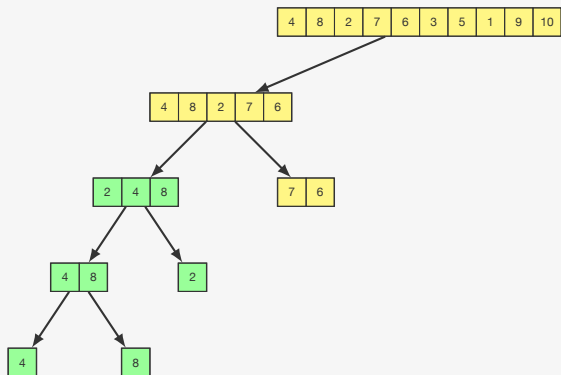
Simulação



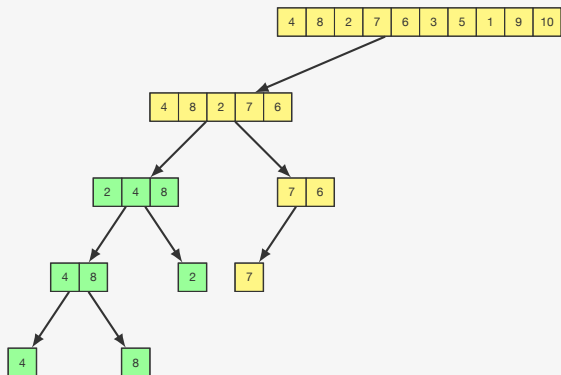
Simulação



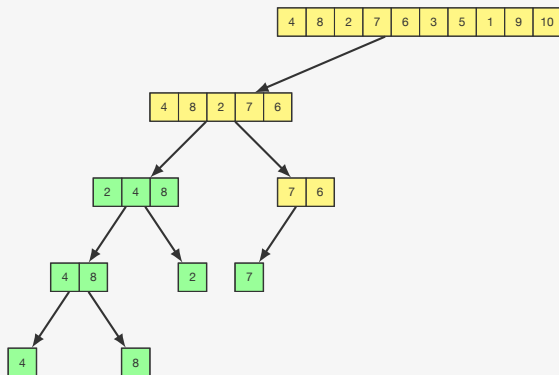
Simulação



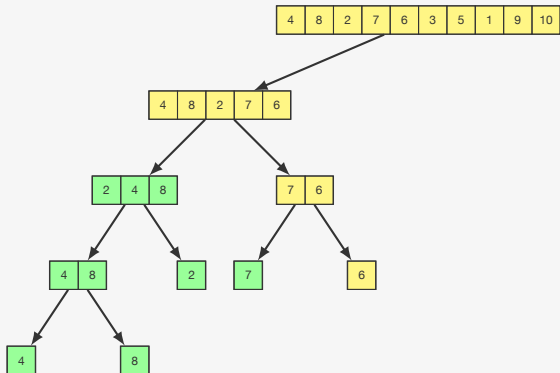
Simulação



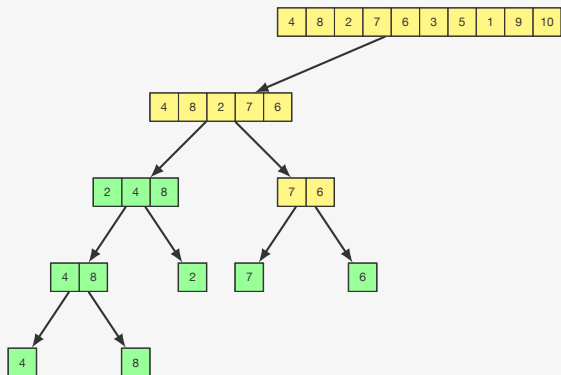
Simulação



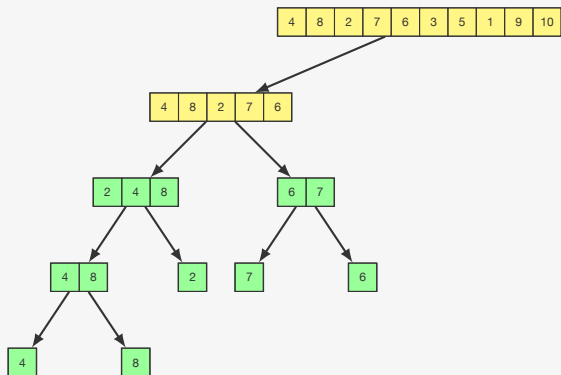
Simulação



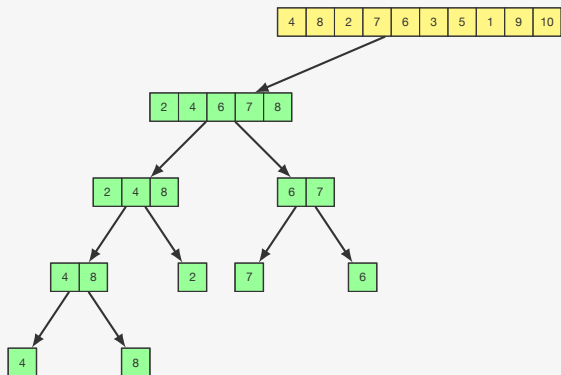
Simulação



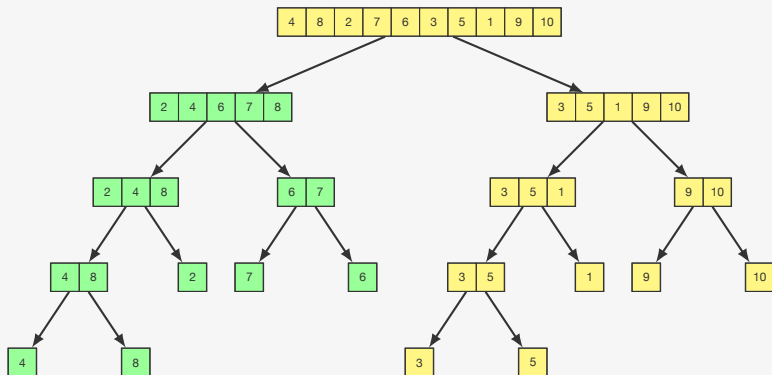
Simulação



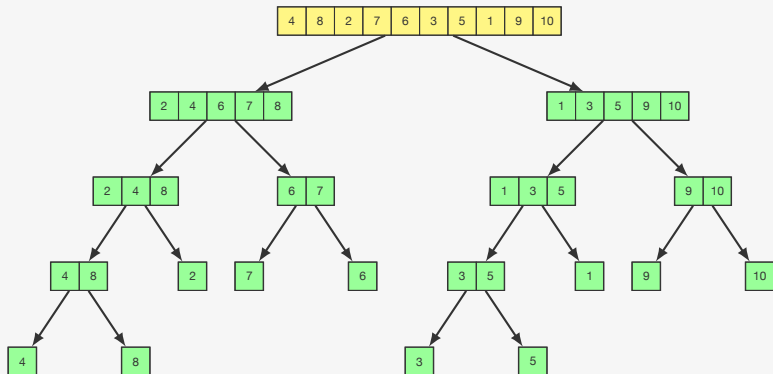
Simulação



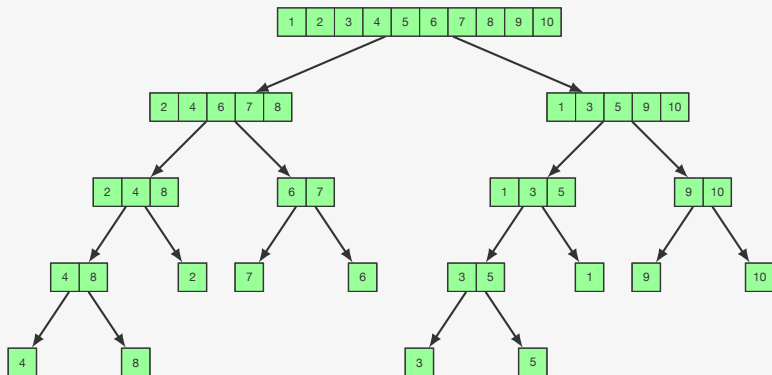
Simulação



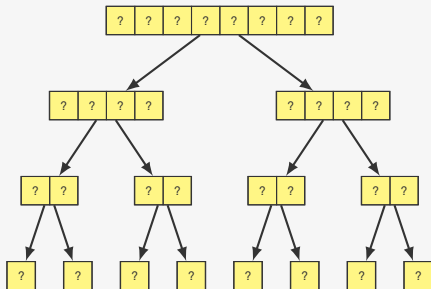
Simulação



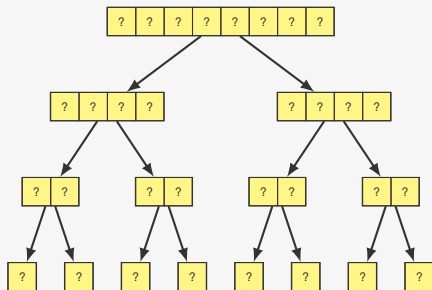
Simulação



Tempo de execução para $n = 2^l$



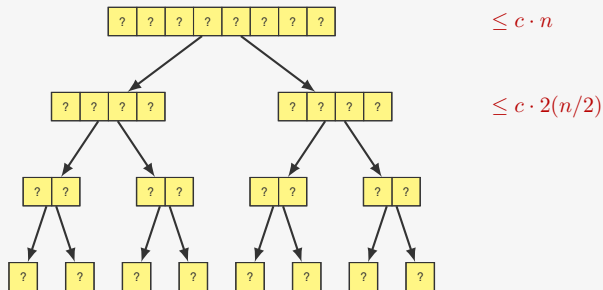
Tempo de execução para $n = 2^l$



$$\leq c \cdot n$$

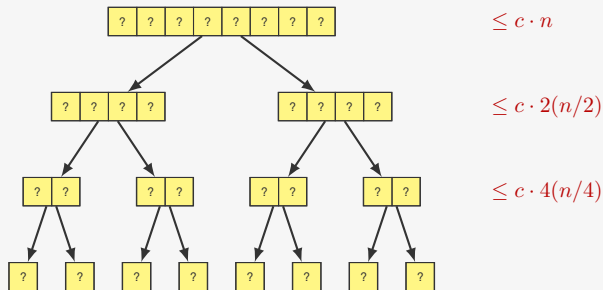
- No primeiro nível fazemos **um** merge com n elementos

Tempo de execução para $n = 2^l$



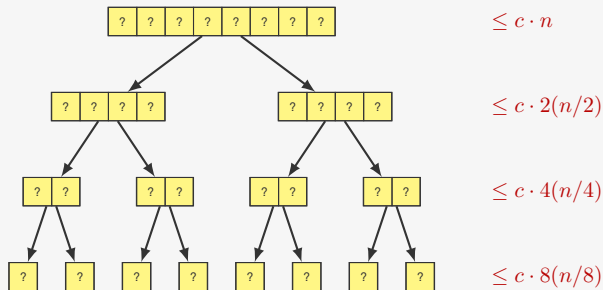
- No primeiro nível fazemos **um** merge com n elementos
- No segundo fazemos **dois** merge com $n/2$ elementos

Tempo de execução para $n = 2^l$



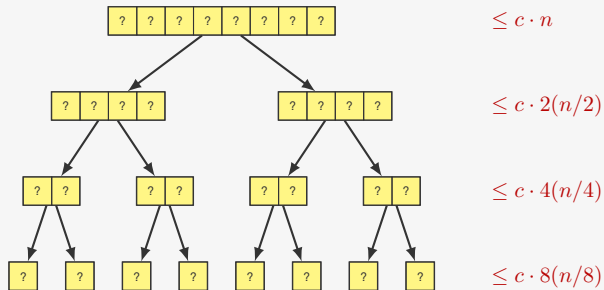
- No primeiro nível fazemos **um** merge com n elementos
- No segundo fazemos **dois** merge com $n/2$ elementos
- No $(k - 1)$ -ésimo fazemos 2^k merge com $n/2^k$ elementos

Tempo de execução para $n = 2^l$



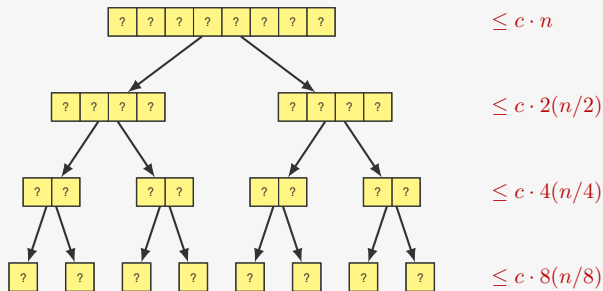
- No primeiro nível fazemos **um** merge com n elementos
- No segundo fazemos **dois** merge com $n/2$ elementos
- No $(k - 1)$ -ésimo fazemos 2^k merge com $n/2^k$ elementos
- No último gastamos tempo constante n vezes

Tempo de execução para $n = 2^l$



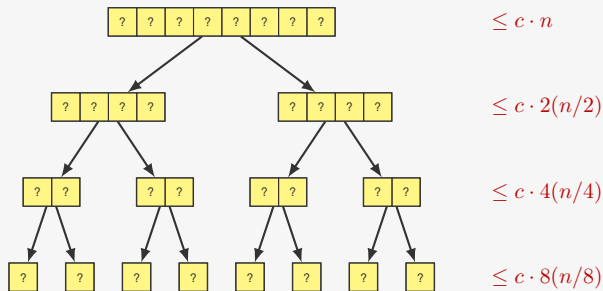
- No nível k gastamos tempo $\leq c \cdot n$

Tempo de execução para $n = 2^l$



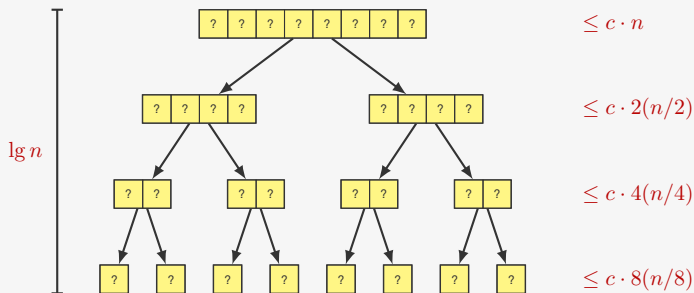
- No nível k gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?

Tempo de execução para $n = 2^l$



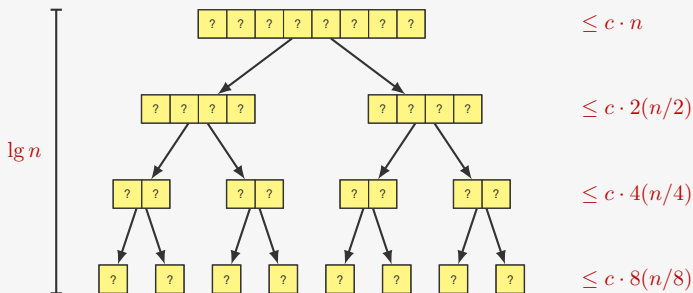
- No nível k gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?
 - Dividimos n por 2 até que fique menores ou igual a 1

Tempo de execução para $n = 2^l$



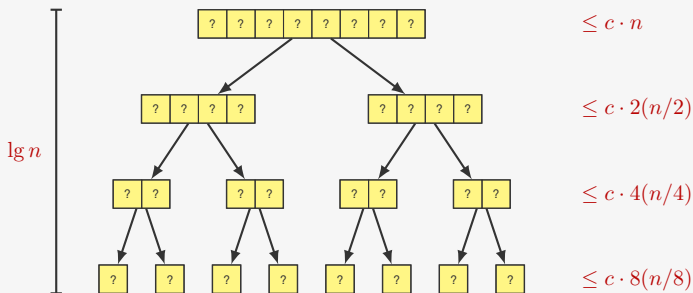
- No nível k gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?
 - Dividimos n por 2 até que fique menores ou igual a 1
 - Ou seja, $l = \log_2 n$

Tempo de execução para $n = 2^l$



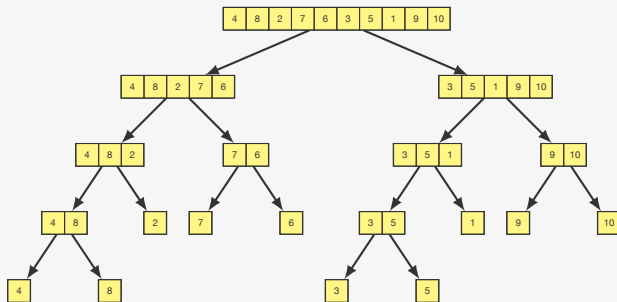
- No nível k gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?
 - Dividimos n por 2 até que fique menores ou igual a 1
 - Ou seja, $l = \log_2 n$
- Como $\log_2 n$ é muito comum, escrevemos $\lg n$

Tempo de execução para $n = 2^l$

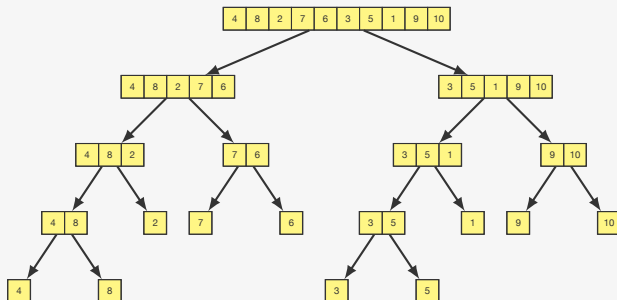


- No nível k gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?
 - Dividimos n por 2 até que fique menores ou igual a 1
 - Ou seja, $l = \log_2 n$
- Como $\log_2 n$ é muito comum, escrevemos $\lg n$
- Tempo total: $cn \lg n = O(n \lg n)$

Tempo de execução para n qualquer

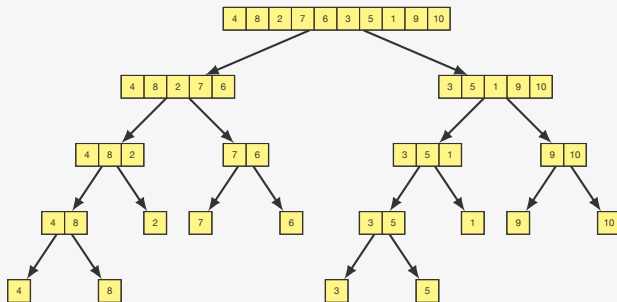


Tempo de execução para n qualquer



Qual o tempo de execução para n que não é potência de 2?

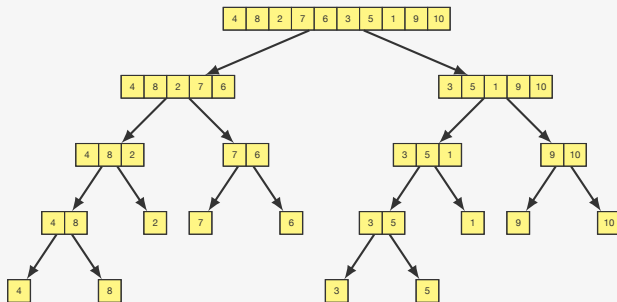
Tempo de execução para n qualquer



Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n

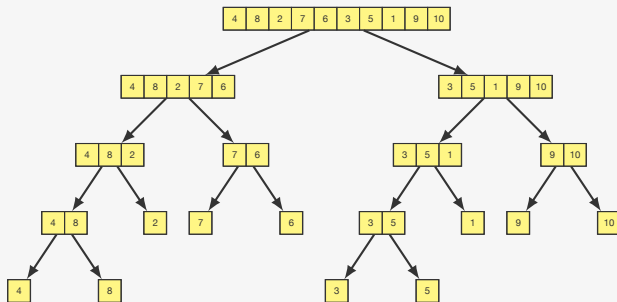
Tempo de execução para n qualquer



Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
 - Ex: Se $n = 3000$, a próxima potência é 4096

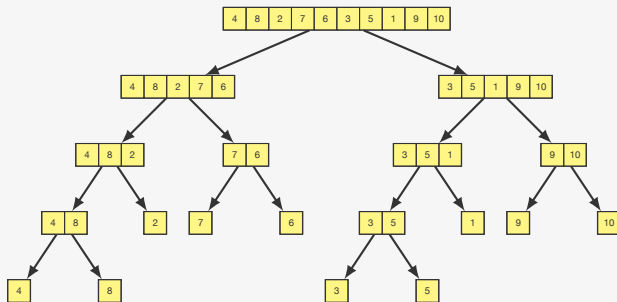
Tempo de execução para n qualquer



Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
 - Ex: Se $n = 3000$, a próxima potência é 4096
- Temos que $2^{k-1} < n < 2^k$

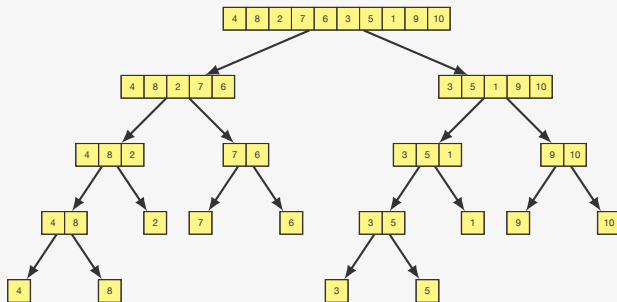
Tempo de execução para n qualquer



Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
 - Ex: Se $n = 3000$, a próxima potência é 4096
- Temos que $2^{k-1} < n < 2^k$
 - Ou seja, $2^k < 2n$

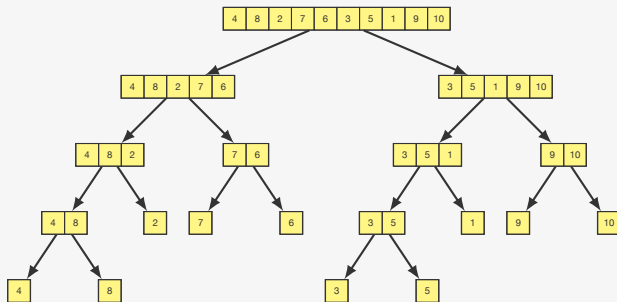
Tempo de execução para n qualquer



Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
 - Ex: Se $n = 3000$, a próxima potência é 4096
- Temos que $2^{k-1} < n < 2^k$
 - Ou seja, $2^k < 2n$
- O tempo de execução para n é menor do que

Tempo de execução para n qualquer

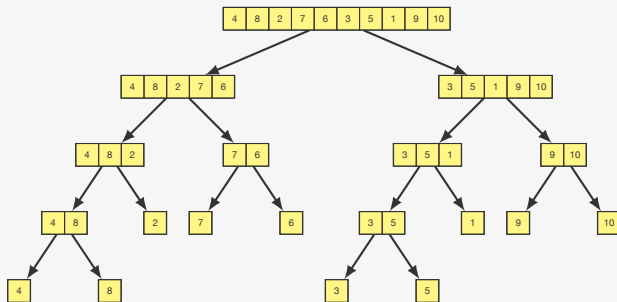


Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
 - Ex: Se $n = 3000$, a próxima potência é 4096
- Temos que $2^{k-1} < n < 2^k$
 - Ou seja, $2^k < 2n$
- O tempo de execução para n é menor do que

$$c 2^k \lg 2^k$$

Tempo de execução para n qualquer

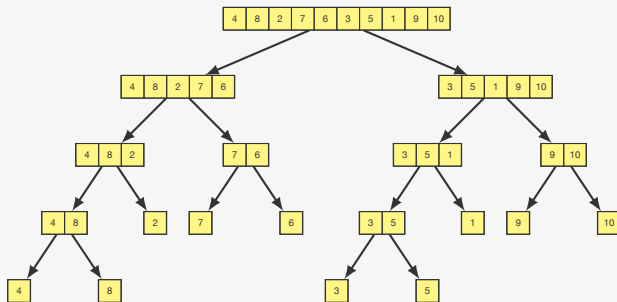


Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
 - Ex: Se $n = 3000$, a próxima potência é 4096
- Temos que $2^{k-1} < n < 2^k$
 - Ou seja, $2^k < 2n$
- O tempo de execução para n é menor do que

$$c 2^k \lg 2^k$$

Tempo de execução para n qualquer

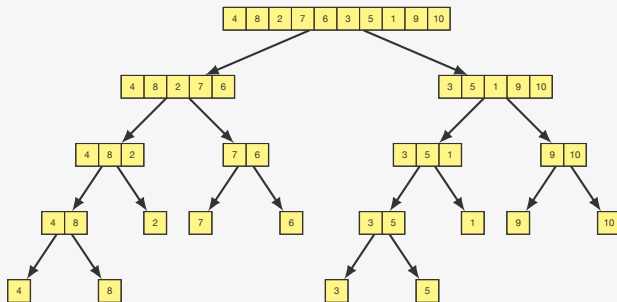


Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
 - Ex: Se $n = 3000$, a próxima potência é 4096
- Temos que $2^{k-1} < n < 2^k$
 - Ou seja, $2^k < 2n$
- O tempo de execução para n é menor do que

$$c 2^k \lg 2^k \leq 2cn \lg(2n)$$

Tempo de execução para n qualquer

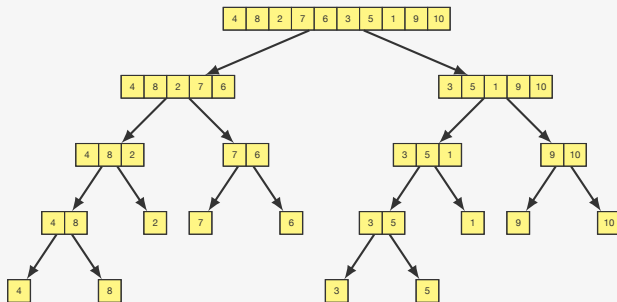


Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
 - Ex: Se $n = 3000$, a próxima potência é 4096
- Temos que $2^{k-1} < n < 2^k$
 - Ou seja, $2^k < 2n$
- O tempo de execução para n é menor do que

$$c 2^k \lg 2^k \leq 2cn \lg(2n) = 2cn(\lg 2 + \lg n)$$

Tempo de execução para n qualquer

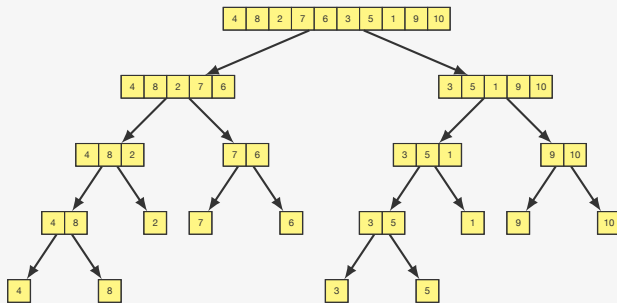


Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
 - Ex: Se $n = 3000$, a próxima potência é 4096
- Temos que $2^{k-1} < n < 2^k$
 - Ou seja, $2^k < 2n$
- O tempo de execução para n é menor do que

$$c 2^k \lg 2^k \leq 2cn \lg(2n) = 2cn(\lg 2 + \lg n) = 2cn + 2cn \lg n$$

Tempo de execução para n qualquer

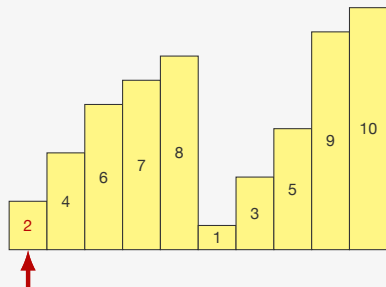


Qual o tempo de execução para n que não é potência de 2?

- Seja 2^k a próxima potência de 2 depois de n
 - Ex: Se $n = 3000$, a próxima potência é 4096
- Temos que $2^{k-1} < n < 2^k$
 - Ou seja, $2^k < 2n$
- O tempo de execução para n é menor do que

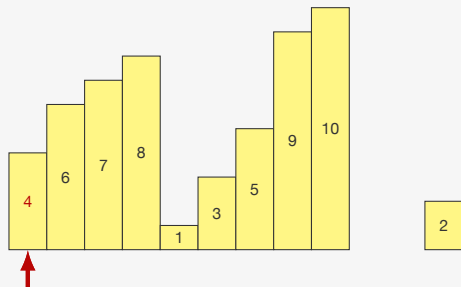
$$c 2^k \lg 2^k \leq 2cn \lg(2n) = 2cn(\lg 2 + \lg n) = 2cn + 2cn \lg n = O(n \lg n)$$

Otimizando a intercalação



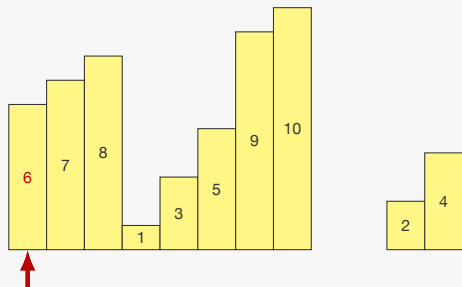
- Copiamos a primeira metade do vetor para o vetor auxiliar

Otimizando a intercalação



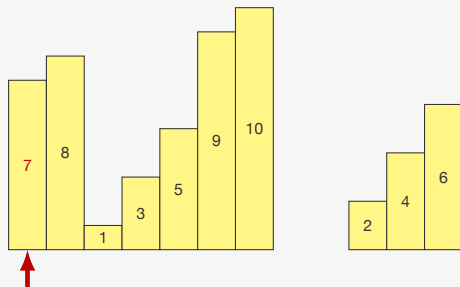
- Copiamos a primeira metade do vetor para o vetor auxiliar

Otimizando a intercalação



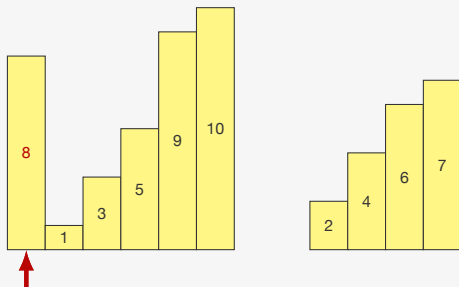
- Copiamos a primeira metade do vetor para o vetor auxiliar

Otimizando a intercalação



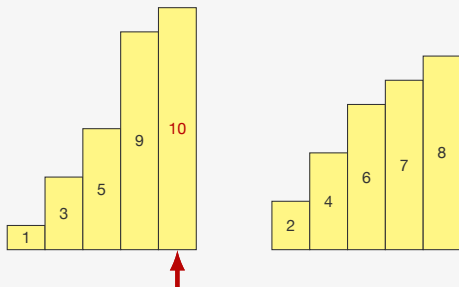
- Copiamos a primeira metade do vetor para o vetor auxiliar

Otimizando a intercalação



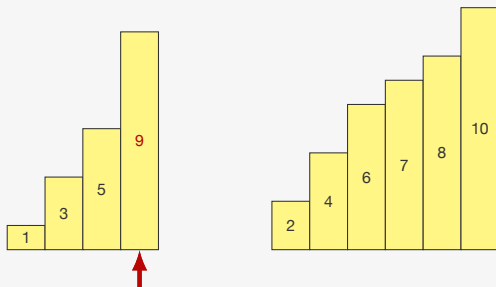
- Copiamos a primeira metade do vetor para o vetor auxiliar

Otimizando a intercalação



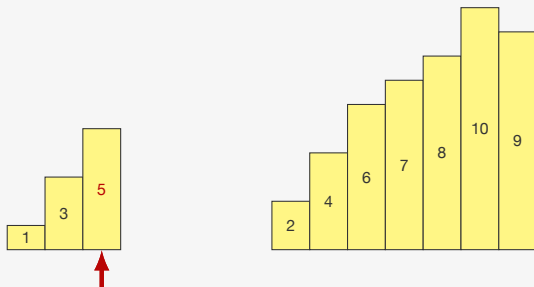
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar

Otimizando a intercalação



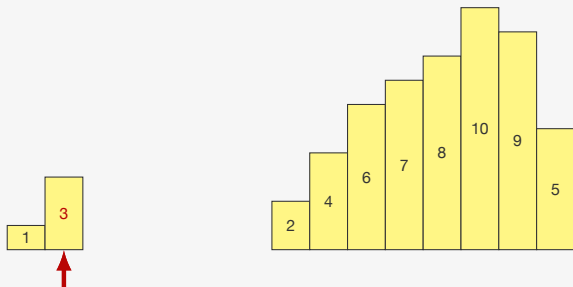
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar

Otimizando a intercalação



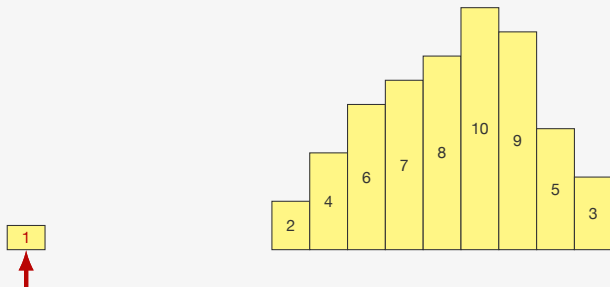
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar

Otimizando a intercalação



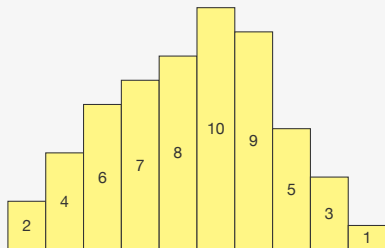
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar

Otimizando a intercalação



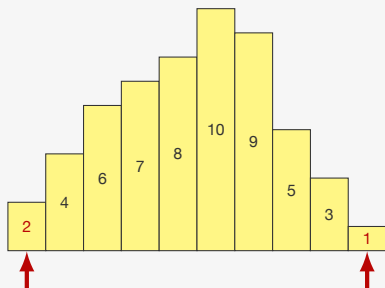
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar

Otimizando a intercalação



- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar

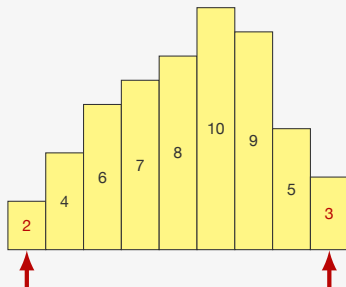
Otimizando a intercalação



- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar
- Basta ir comparando
 - da esquerda para a direita
 - e da direita para a esquerda

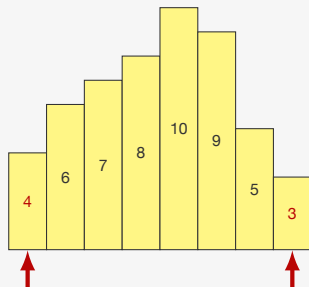
Otimizando a intercalação

1



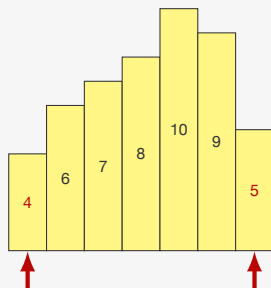
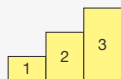
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar
- Basta ir comparando
 - da esquerda para a direita
 - e da direita para a esquerda

Otimizando a intercalação



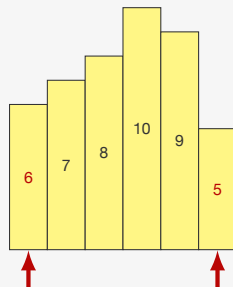
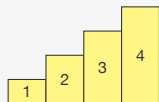
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar
- Basta ir comparando
 - da esquerda para a direita
 - e da direita para a esquerda

Otimizando a intercalação



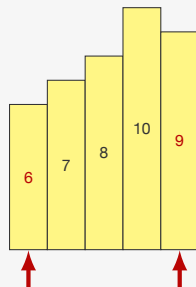
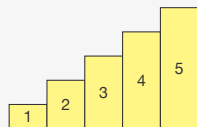
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar
- Basta ir comparando
 - da esquerda para a direita
 - e da direita para a esquerda

Otimizando a intercalação



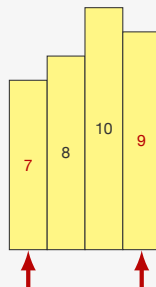
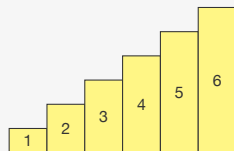
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar
- Basta ir comparando
 - da esquerda para a direita
 - e da direita para a esquerda

Otimizando a intercalação



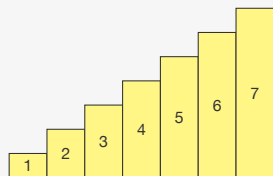
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar
- Basta ir comparando
 - da esquerda para a direita
 - e da direita para a esquerda

Otimizando a intercalação



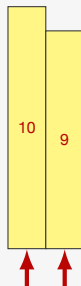
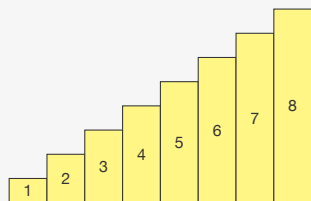
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar
- Basta ir comparando
 - da esquerda para a direita
 - e da direita para a esquerda

Otimizando a intercalação



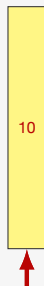
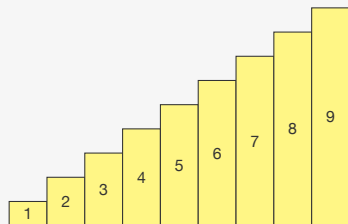
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar
- Basta ir comparando
 - da esquerda para a direita
 - e da direita para a esquerda

Otimizando a intercalação



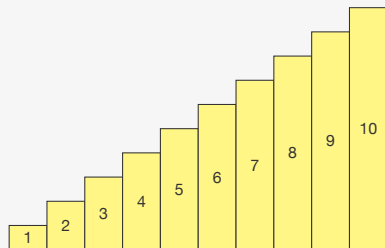
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar
- Basta ir comparando
 - da esquerda para a direita
 - e da direita para a esquerda

Otimizando a intercalação



- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar
- Basta ir comparando
 - da esquerda para a direita
 - e da direita para a esquerda

Otimizando a intercalação



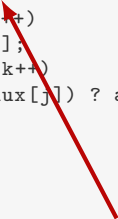
- Copiamos a primeira metade do vetor para o vetor auxiliar
- Copiamos a segunda metade invertida no vetor auxiliar
- Basta ir comparando
 - da esquerda para a direita
 - e da direita para a esquerda

Otimizando a intercalação

```
1 void merge_v2(int *v, int l, int m, int r) {
2     int i, j, k;
3     int aux[MAX];
4     for (i = m+1; i > l; i--)
5         aux[i-1] = v[i-1];
6     for (j = m; j < r; j++)
7         aux[r+m-j] = v[j+1];
8     for (k = l; k <= r; k++)
9         v[k] = (aux[i] < aux[j]) ? aux[i++] : aux[j--];
10 }
```

Otimizando a intercalação

```
1 void merge_v2(int *v, int l, int m, int r) {
2     int i, j, k;
3     int aux[MAX];
4     for (i = m+1; i > l; i--)
5         aux[i-1] = v[i-1];
6     for (j = m; j < r; j++)
7         aux[r+m-j] = v[j+1];
8     for (k = l; k <= r; k++)
9         v[k] = (aux[i] < aux[j]) ? aux[i++] : aux[j--];
10 }
```



copia a primeira metade para **aux**

Otimizando a intercalação

```
1 void merge_v2(int *v, int l, int m, int r) {
2     int i, j, k;
3     int aux[MAX];
4     for (i = m+1; i > l; i--)
5         aux[i-1] = v[i-1];
6     for (j = m; j < r; j++)
7         aux[r+m-j] = v[j+1];
8     for (k = l; k <= r; k++)
9         v[k] = (aux[i] < aux[j]) ? aux[i++] : aux[j--];
10 }
```

quando o `for` acaba, `i == l`

Otimizando a intercalação

```
1 void merge_v2(int *v, int l, int m, int r) {
2     int i, j, k;
3     int aux[MAX];
4     for (i = m+1; i > l; i--)
5         aux[i-1] = v[i-1];
6     for (j = m; j < r; j++)
7         aux[r+m-j] = v[j+1];
8     for (k = l; k <= r; k++)
9         v[k] = (aux[i] < aux[j]) ? aux[i++] : aux[j--];
10 }
```



copia a segunda metade invertido para **aux**

Otimizando a intercalação

```
1 void merge_v2(int *v, int l, int m, int r) {
2     int i, j, k;
3     int aux[MAX];
4     for (i = m+1; i > l; i--)
5         aux[i-1] = v[i-1];
6     for (j = m; j < r; j++)
7         aux[r+m-j] = v[j+1];
8     for (k = l; k <= r; k++)
9         v[k] = (aux[i] < aux[j]) ? aux[i++] : aux[j--];
10 }
```



quando o `for` acaba, `j == r`

Otimizando a intercalação

```
1 void merge_v2(int *v, int l, int m, int r) {
2     int i, j, k;
3     int aux[MAX];
4     for (i = m+1; i > l; i--)
5         aux[i-1] = v[i-1];
6     for (j = m; j < r; j++)
7         aux[r+m-j] = v[j+1];
8     for (k = l; k <= r; k++)
9         v[k] = (aux[i] < aux[j]) ? aux[i++] : aux[j--];
10 }
```

se `aux[i] < aux[j]`



Otimizando a intercalação

```
1 void merge_v2(int *v, int l, int m, int r) {
2     int i, j, k;
3     int aux[MAX];
4     for (i = m+1; i > l; i--)
5         aux[i-1] = v[i-1];
6     for (j = m; j < r; j++)
7         aux[r+m-j] = v[j+1];
8     for (k = l; k <= r; k++)
9         v[k] = (aux[i] < aux[j]) ? aux[i++] : aux[j--];
10 }
```

copiamos `aux[i]` para `v[k]`



Otimizando a intercalação

```
1 void merge_v2(int *v, int l, int m, int r) {
2     int i, j, k;
3     int aux[MAX];
4     for (i = m+1; i > l; i--)
5         aux[i-1] = v[i-1];
6     for (j = m; j < r; j++)
7         aux[r+m-j] = v[j+1];
8     for (k = l; k <= r; k++)
9         v[k] = (aux[i] < aux[j]) ? aux[i++] : aux[j--];
10 }
```

aumentamos `i`



Otimizando a intercalação

```
1 void merge_v2(int *v, int l, int m, int r) {
2     int i, j, k;
3     int aux[MAX];
4     for (i = m+1; i > l; i--)
5         aux[i-1] = v[i-1];
6     for (j = m; j < r; j++)
7         aux[r+m-j] = v[j+1];
8     for (k = l; k <= r; k++)
9         v[k] = (aux[i] < aux[j]) ? aux[i++] : aux[j--];
10 }
```

senão



Otimizando a intercalação

```
1 void merge_v2(int *v, int l, int m, int r) {
2     int i, j, k;
3     int aux[MAX];
4     for (i = m+1; i > l; i--)
5         aux[i-1] = v[i-1];
6     for (j = m; j < r; j++)
7         aux[r+m-j] = v[j+1];
8     for (k = l; k <= r; k++)
9         v[k] = (aux[i] < aux[j]) ? aux[i++] : aux[j--];
10 }
```


copiamos `aux[j]` para `v[k]`



Otimizando a intercalação

```
1 void merge_v2(int *v, int l, int m, int r) {
2     int i, j, k;
3     int aux[MAX];
4     for (i = m+1; i > l; i--)
5         aux[i-1] = v[i-1];
6     for (j = m; j < r; j++)
7         aux[r+m-j] = v[j+1];
8     for (k = l; k <= r; k++)
9         v[k] = (aux[i] < aux[j]) ? aux[i++] : aux[j--];
10 }
```

diminuímos **j**



MergeSort de baixo para cima (*bottom-up*)

O MergeSort que vimos é de cima para baixo (*top-down*)

MergeSort de baixo para cima (*bottom-up*)

O MergeSort que vimos é de cima para baixo (*top-down*)

- Mas podemos fazer também de baixo para cima

MergeSort de baixo para cima (*bottom-up*)

O MergeSort que vimos é de cima para baixo (*top-down*)

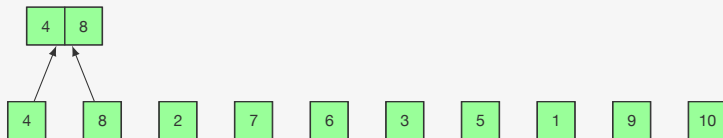
- Mas podemos fazer também de baixo para cima



MergeSort de baixo para cima (*bottom-up*)

O MergeSort que vimos é de cima para baixo (*top-down*)

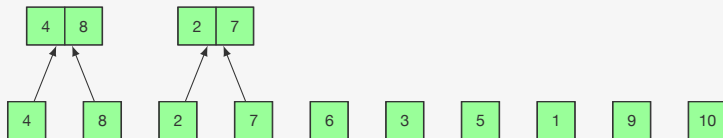
- Mas podemos fazer também de baixo para cima



MergeSort de baixo para cima (*bottom-up*)

O MergeSort que vimos é de cima para baixo (*top-down*)

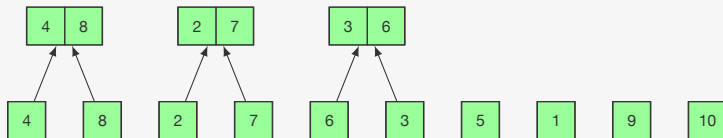
- Mas podemos fazer também de baixo para cima



MergeSort de baixo para cima (*bottom-up*)

O MergeSort que vimos é de cima para baixo (*top-down*)

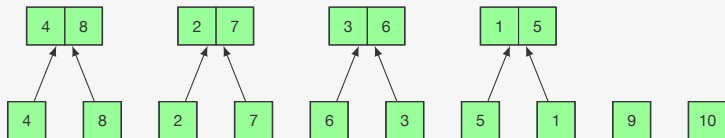
- Mas podemos fazer também de baixo para cima



MergeSort de baixo para cima (*bottom-up*)

O MergeSort que vimos é de cima para baixo (*top-down*)

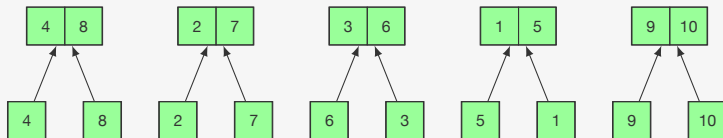
- Mas podemos fazer também de baixo para cima



MergeSort de baixo para cima (*bottom-up*)

O MergeSort que vimos é de cima para baixo (*top-down*)

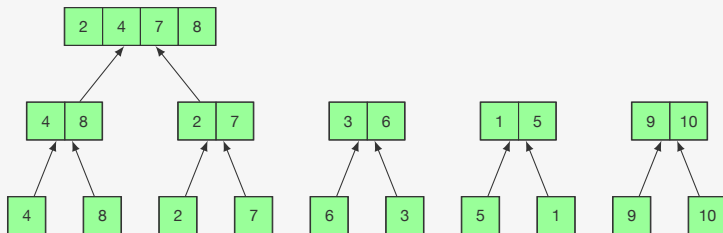
- Mas podemos fazer também de baixo para cima



MergeSort de baixo para cima (*bottom-up*)

O MergeSort que vimos é de cima para baixo (*top-down*)

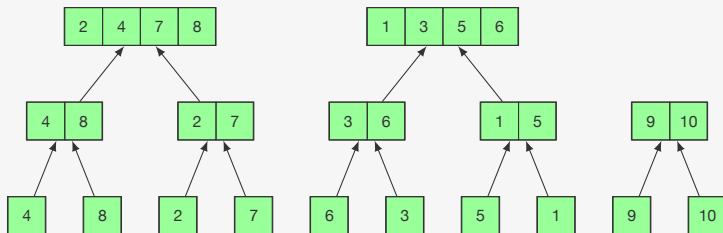
- Mas podemos fazer também de baixo para cima



MergeSort de baixo para cima (*bottom-up*)

O MergeSort que vimos é de cima para baixo (*top-down*)

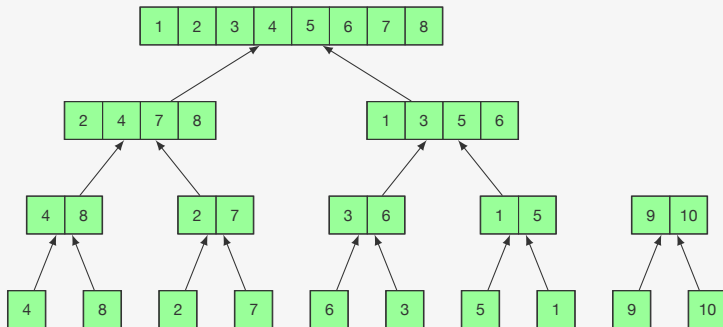
- Mas podemos fazer também de baixo para cima



MergeSort de baixo para cima (*bottom-up*)

O MergeSort que vimos é de cima para baixo (*top-down*)

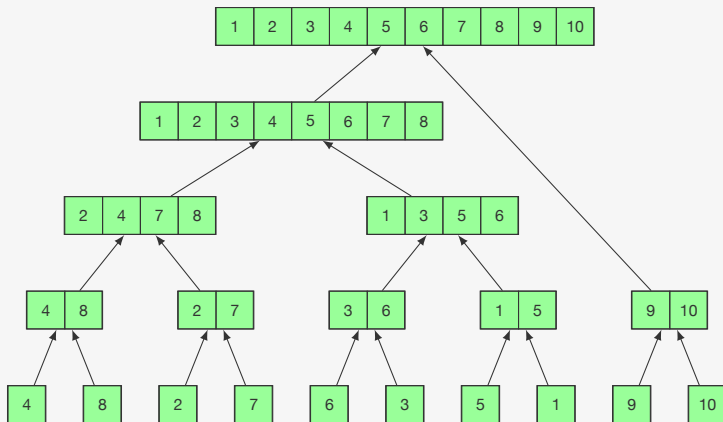
- Mas podemos fazer também de baixo para cima



MergeSort de baixo para cima (*bottom-up*)

O MergeSort que vimos é de cima para baixo (*top-down*)

- Mas podemos fazer também de baixo para cima

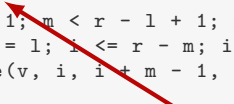


MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

MergeSort de baixo para cima (*bottom-up*)

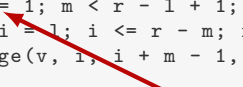
```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```



m é o tamanho dos vetores que faremos merge

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```



no começo $m = 1$ (merge de dois vetores de tamanho 1)

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

no passo seguinte, $m = 2$


MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

e no seguinte, $m = 4$

MergeSort de baixo para cima (*bottom-up*)

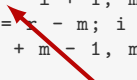
```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```



e assim por diante

MergeSort de baixo para cima (*bottom-up*)

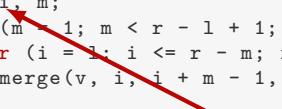
```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```



enquanto **m** for menor que o número de elementos

MergeSort de baixo para cima (*bottom-up*)

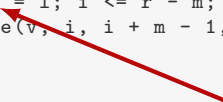
```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = 1; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```



i indica a primeira posição do primeiro vetor que faremos merge

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = 1; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```



i = 1 no começo

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

$i = l + 2*m$ no passo seguinte

MergeSort de baixo para cima (*bottom-up*)


```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```



e assim por diante

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```




enquanto ainda couber dois vetores, um tamanho m e outro com tamanho m ou menor

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

o primeiro vetor tem tamanho **m**




MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

portanto, vai de $v[i]$ a $v[i+m-1]$

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```



o segundo vetor acaba ou em $i + 2*m - 1$ ou em r

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

 m i i+m-1 min(i+2*m-1, r) intercalação

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

m	i	i+m-1	min(i+2*m-1, r)	intercalação
1	0	0	1	v[0] com v[1]

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

m	i	i+m-1	min(i+2*m-1, r)	intercalação
1	0	0	1	v[0] com v[1]
1	2	2	3	v[2] com v[3]

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

m	i	i+m-1	min(i+2*m-1, r)	intercalação
1	0	0	1	v[0] com v[1]
1	2	2	3	v[2] com v[3]
1	4	4	5	v[4] com v[5]

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

m	i	i+m-1	min(i+2*m-1, r)	intercalação
1	0	0	1	v[0] com v[1]
1	2	2	3	v[2] com v[3]
1	4	4	5	v[4] com v[5]
1	6	6	7	v[6] com v[7]

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

m	i	i+m-1	min(i+2*m-1, r)	intercalação
1	0	0	1	v[0] com v[1]
1	2	2	3	v[2] com v[3]
1	4	4	5	v[4] com v[5]
1	6	6	7	v[6] com v[7]
1	8	8	9	v[8] com v[9]

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

m	i	i+m-1	min(i+2*m-1, r)	intercalação
1	0	0	1	v[0] com v[1]
1	2	2	3	v[2] com v[3]
1	4	4	5	v[4] com v[5]
1	6	6	7	v[6] com v[7]
1	8	8	9	v[8] com v[9]
2	0	1	3	v[0], v[1] com v[2], v[3]

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

m	i	i+m-1	min(i+2*m-1, r)	intercalação
1	0	0	1	v[0] com v[1]
1	2	2	3	v[2] com v[3]
1	4	4	5	v[4] com v[5]
1	6	6	7	v[6] com v[7]
1	8	8	9	v[8] com v[9]
2	0	1	3	v[0], v[1] com v[2], v[3]
2	4	5	7	v[4], v[5] com v[6], v[7]

MergeSort de baixo para cima (*bottom-up*)

```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

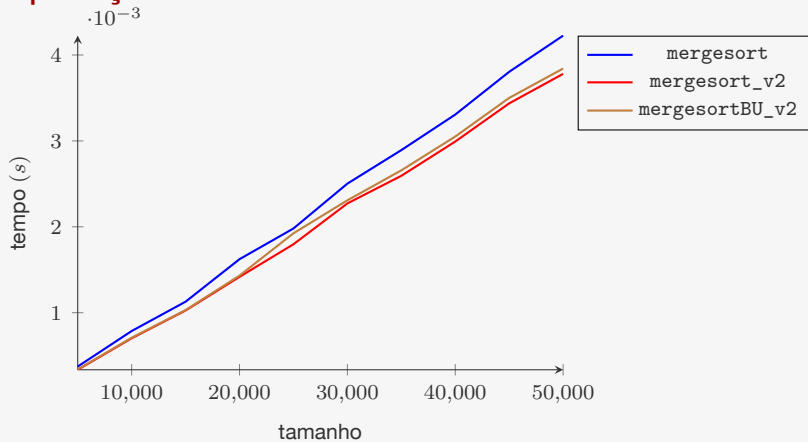
m	i	i+m-1	min(i+2*m-1, r)	intercalação
1	0	0	1	v[0] com v[1]
1	2	2	3	v[2] com v[3]
1	4	4	5	v[4] com v[5]
1	6	6	7	v[6] com v[7]
1	8	8	9	v[8] com v[9]
2	0	1	3	v[0], v[1] com v[2], v[3]
2	4	5	7	v[4], v[5] com v[6], v[7]
4	0	3	7	v[0], ..., v[3] com v[4], ..., v[7]

MergeSort de baixo para cima (*bottom-up*)

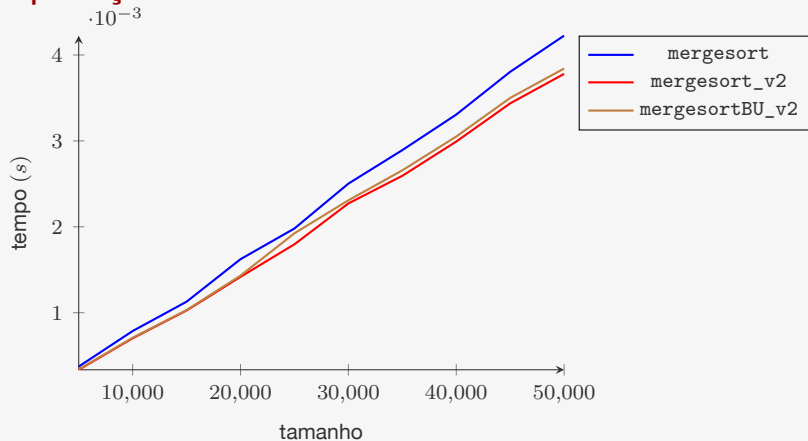
```
1 #define min(A, B) (A < B) ? A : B
2
3 void mergesortBU(int *v, int l, int r) {
4     int i, m;
5     for (m = 1; m < r - l + 1; m = 2*m)
6         for (i = l; i <= r - m; i += 2*m)
7             merge(v, i, i + m - 1, min(i + 2*m - 1, r));
8 }
```

m	i	i+m-1	min(i+2*m-1, r)	intercalação
1	0	0	1	v[0] com v[1]
1	2	2	3	v[2] com v[3]
1	4	4	5	v[4] com v[5]
1	6	6	7	v[6] com v[7]
1	8	8	9	v[8] com v[9]
2	0	1	3	v[0], v[1] com v[2], v[3]
2	4	5	7	v[4], v[5] com v[6], v[7]
4	0	3	7	v[0], ..., v[3] com v[4], ... v[7]
8	0	7	9	v[0], ..., v[7] com v[8], v[9]

Comparação

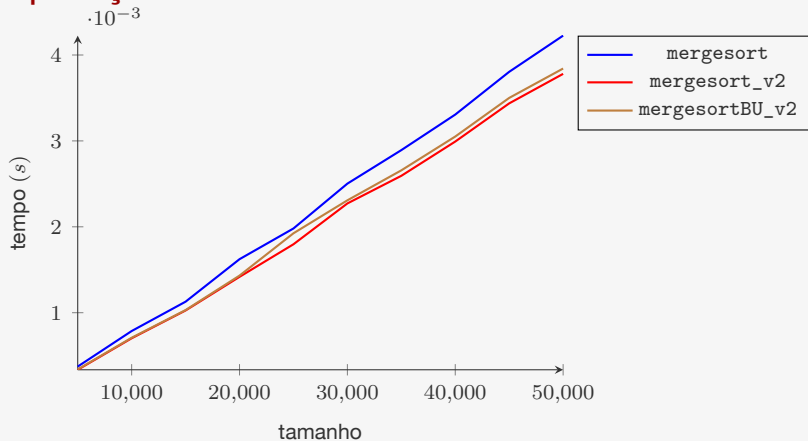


Comparação



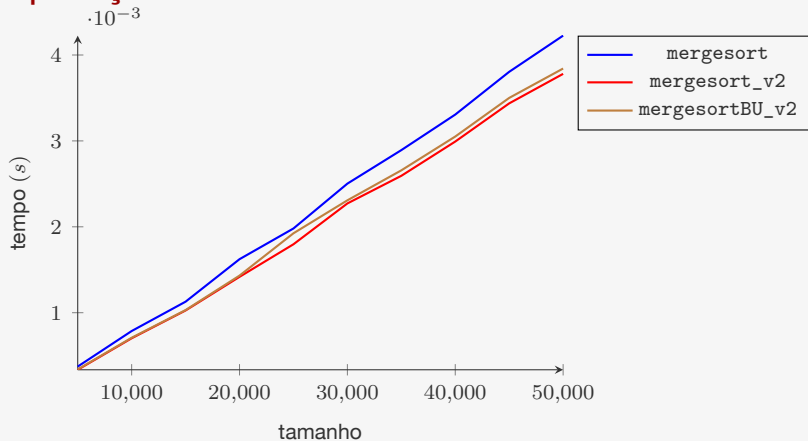
- `mergesortBU_v2` é mais lento que `mergesort_v2`

Comparação



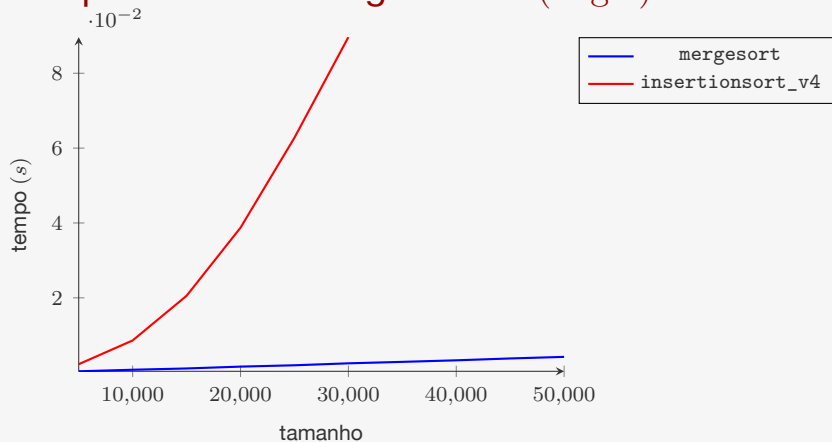
- mergesortBU_v2 é mais lento que mergesort_v2
- as árvores do dois são diferentes

Comparação

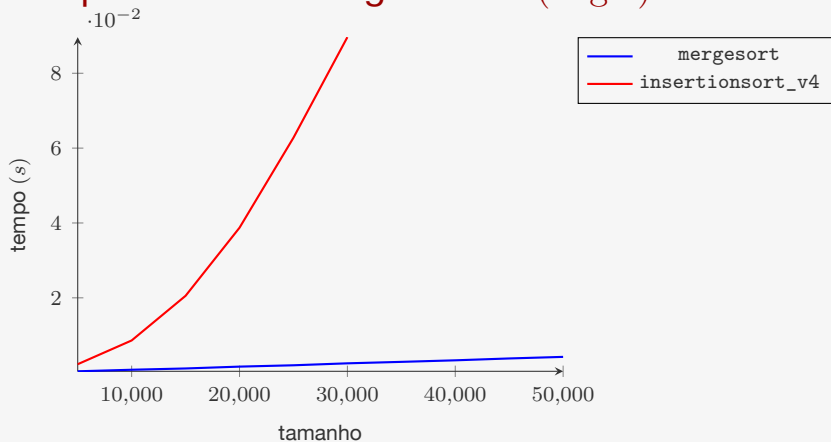


- mergesortBU_v2 é mais lento que mergesort_v2
- as árvores do dois são diferentes
- o algoritmo top-down usa a memória cache melhor

Valeu a pena fazer um algoritmo $O(n \lg n)$?

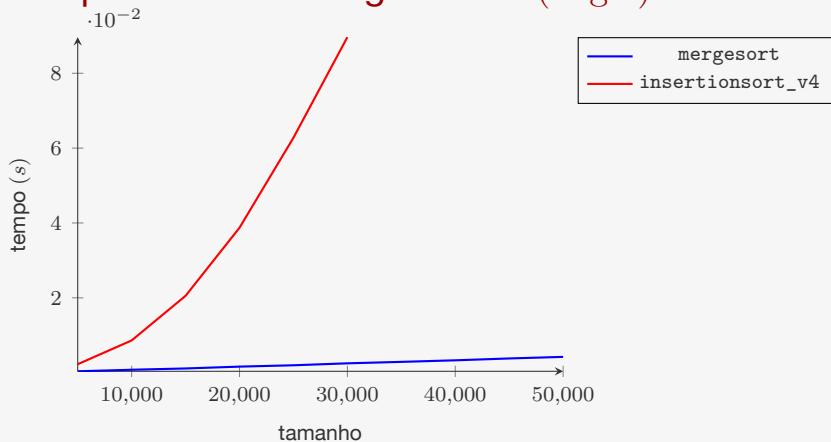


Valeu a pena fazer um algoritmo $O(n \lg n)$?



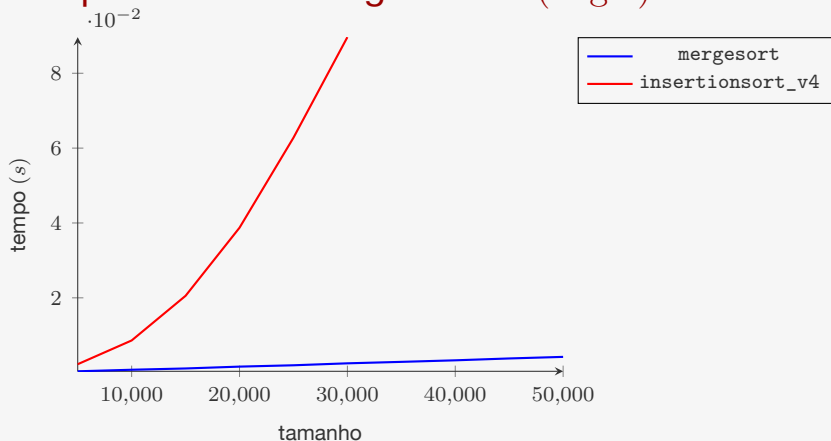
- `insertionsort_v4` ordena 30.000 números em 0.0896s

Valeu a pena fazer um algoritmo $O(n \lg n)$?



- **insertionsort_v4** ordena **30.000** números em **0.0896s**
- **mergesort** ordena **800.000** números em **0.0874s**

Valeu a pena fazer um algoritmo $O(n \lg n)$?



- `insertionsort_v4` ordena 30.000 números em 0.0896s
- `mergesort` ordena 800.000 números em 0.0874s
- É a diferença entre um algoritmo $O(n^2)$ e um $O(n \lg n)$

Exercício

Implemente a função

```
void mergeAB(int *v, int *a, int n, int *b, int m)
```

que dados vetores `a` e `b` de tamanho `n` e `m` faz a intercalação de `a` e `b` e armazena no vetor `v`. Suponha que `v` já está alocado e que tem tamanho maior ou igual a `n+m`.

Implemente uma versão do `mergesort` que evita a cópia para o vetor auxiliar no `merge`.

- Suponha que desejamos ordenar `a` e temos um vetor `b` com o mesmo conteúdo.
- Podemos ordenar as duas metades de `b` e então usar a função `mergeAB` para fazer a intercalação das duas metades de `b`, armazenando em `a`.