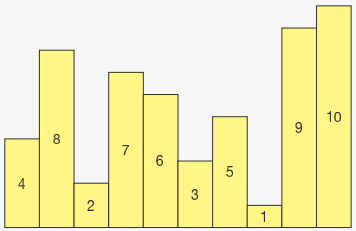
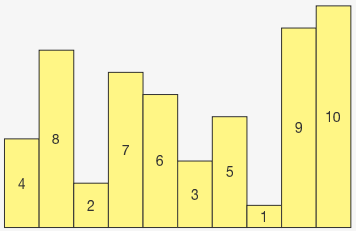
Estratégia: recursão

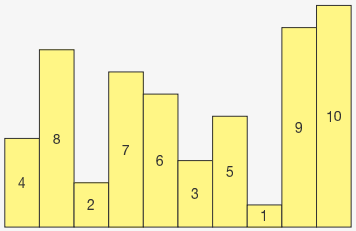


Estratégia: recursão



Como ordenar a primeira metade do vetor?

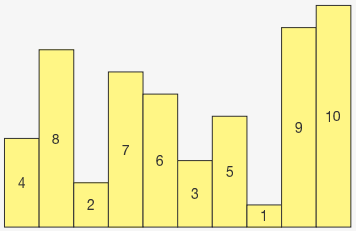
Estratégia: recursão



Como ordenar a primeira metade do vetor?

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

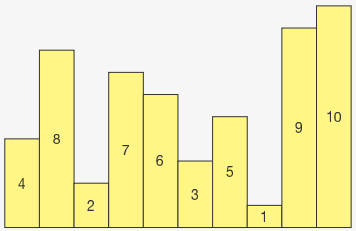
Estratégia: recursão



Como ordenar a primeira metade do vetor?

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

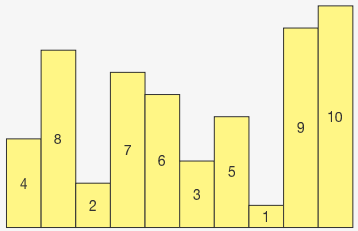
Estratégia: recursão



Como ordenar a primeira metade do vetor?

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

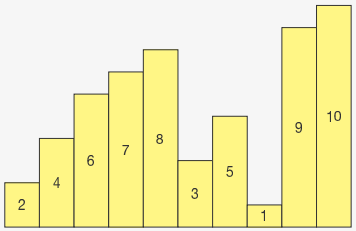
Estratégia: recursão



Como ordenar a primeira metade do vetor?

- usamos uma função `ordernar(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

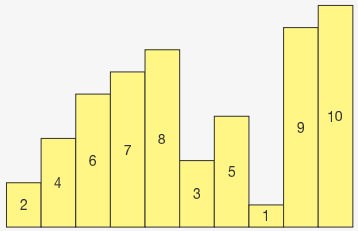


Como ordenar a primeira metade do vetor?

- usamos uma função `ordernar(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?

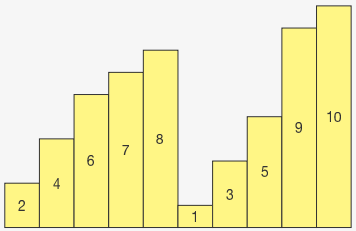
Ordenando a segunda parte



Para ordenar a segunda metade:

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

Ordenando a segunda parte



Para ordenar a segunda metade:

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

Ordenando todo o vetor

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

Ordenando todo o vetor

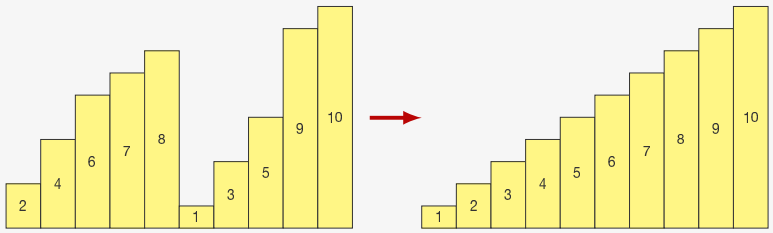
Suponha que temos um vetor com as suas duas metades já ordenadas

- Como ordernar todo o vetor?

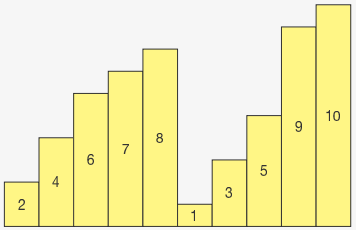
Ordenando todo o vetor

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

- Como ordernar todo o vetor?

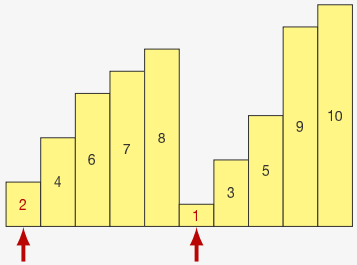


Intercalando



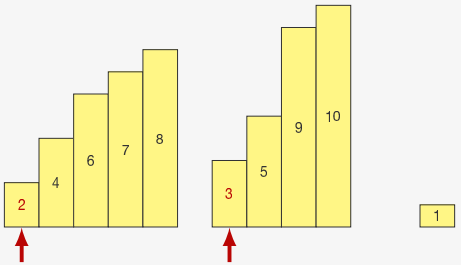
- Percorremos os dois subvetores

Intercalando



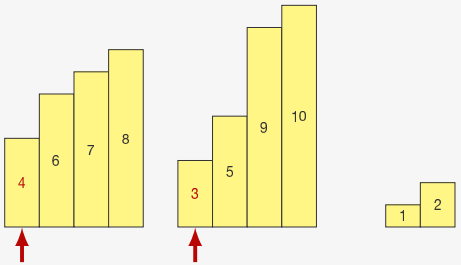
- Percorremos os dois subvetores
- Pegamos o mínimo e inserimos em um vetor auxiliar

Intercalando



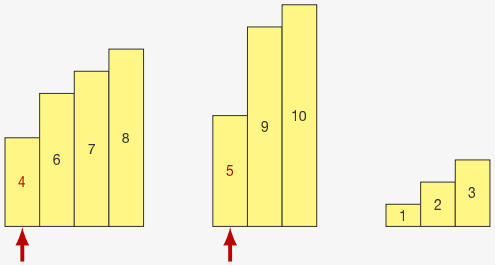
- Percorremos os dois subvetores
- Pegamos o mínimo e inserimos em um vetor auxiliar

Intercalando



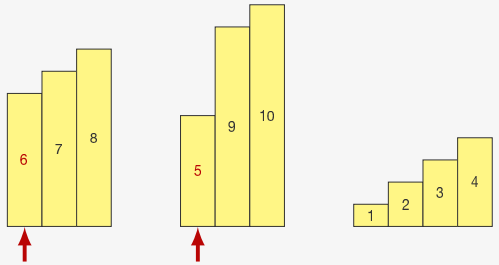
- Percorremos os dois subvetores
- Pegamos o mínimo e inserimos em um vetor auxiliar

Intercalando



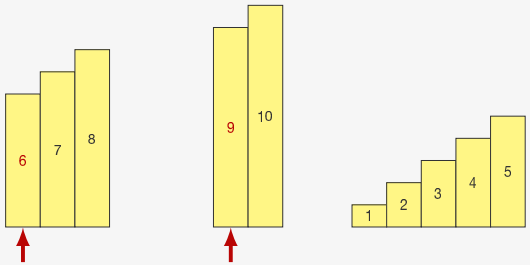
- Percorremos os dois subvetores
- Pegamos o mínimo e inserimos em um vetor auxiliar

Intercalando



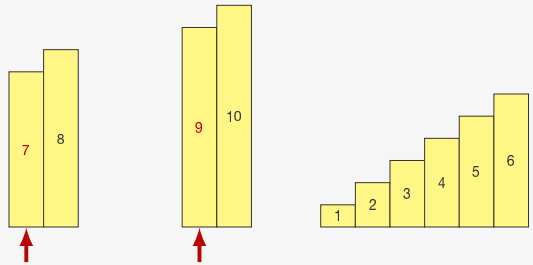
- Percorremos os dois subvetores
- Pegamos o mínimo e inserimos em um vetor auxiliar

Intercalando



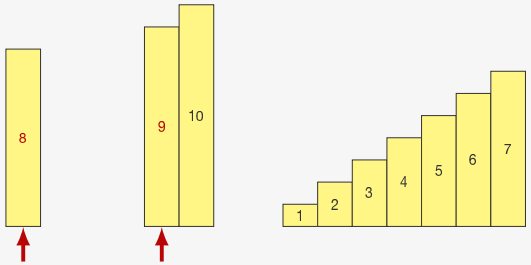
- Percorremos os dois subvetores
- Pegamos o mínimo e inserimos em um vetor auxiliar

Intercalando



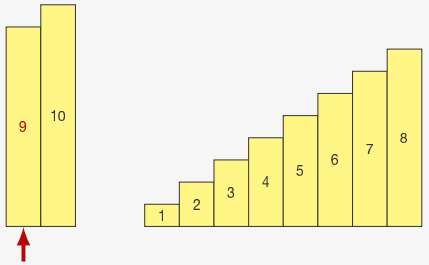
- Percorremos os dois subvetores
- Pegamos o mínimo e inserimos em um vetor auxiliar

Intercalando



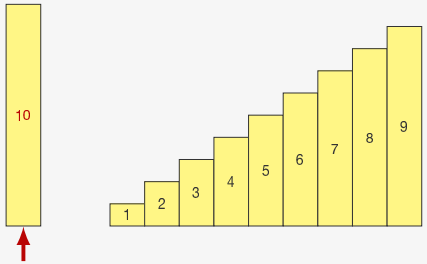
- Percorremos os dois subvetores
- Pegamos o mínimo e inserimos em um vetor auxiliar

Intercalando



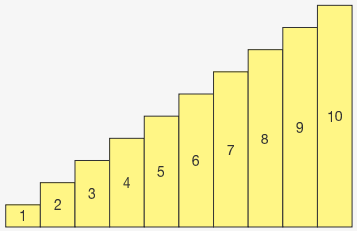
- Percorremos os dois subvetores
- Pegamos o mínimo e inserimos em um vetor auxiliar
- Depois copiamos o restante

Intercalando



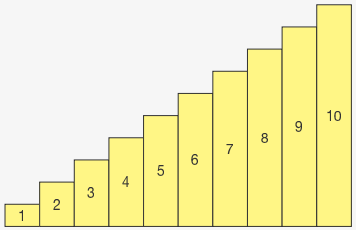
- Percorremos os dois subvetores
- Pegamos o mínimo e inserimos em um vetor auxiliar
- Depois copiamos o restante

Intercalando



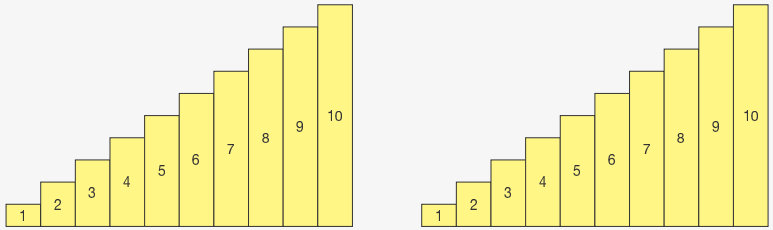
- Percorremos os dois subvetores
- Pegamos o mínimo e inserimos em um vetor auxiliar
- Depois copiamos o restante

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++];
```

9

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 }
```

9

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?

9

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[l]`

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]`

10

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 $n/2$

10

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

10

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

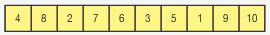
Ordenação por intercalação (*MergeSort*)

Ordenação:

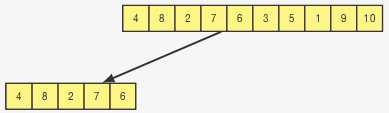
- Recebemos um `vetor` de tamanho n com limites:
  - O vetor começa na posição `vetor[l]`
  - O vetor termina na posição `vetor[r]`
- Dividimos o vetor em dois subvetores de tamanho $n/2$
- O caso base é um vetor de tamanho 0 ou 1

```
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 }
```

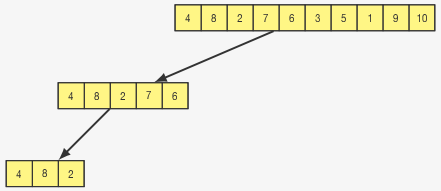

Simulação



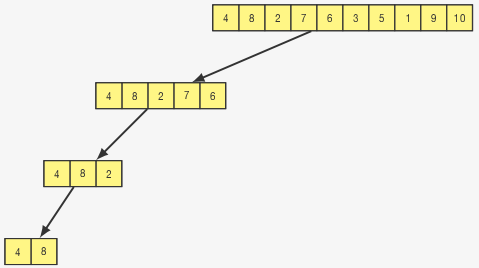
Simulação



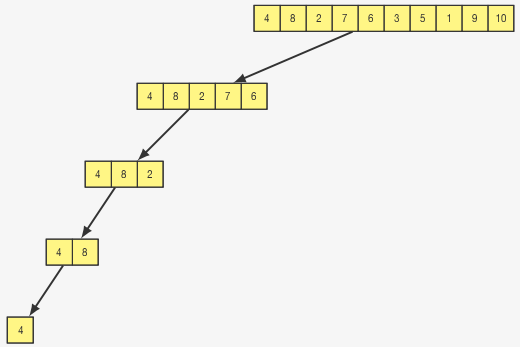
Simulação



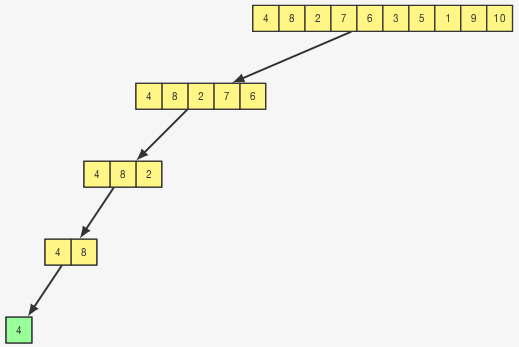
Simulação



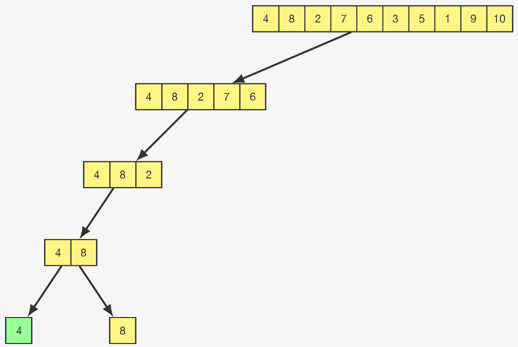
Simulação



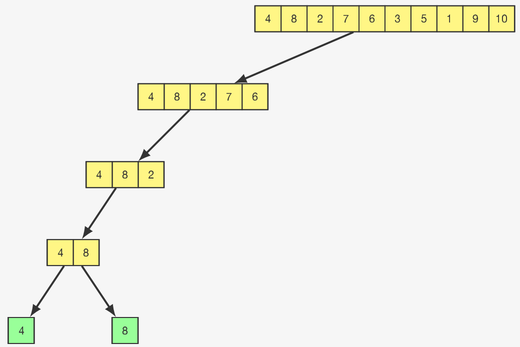
Simulação



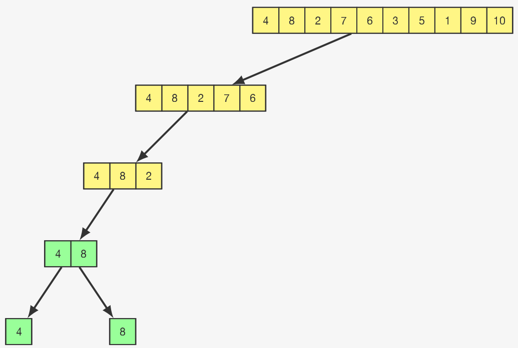
Simulação



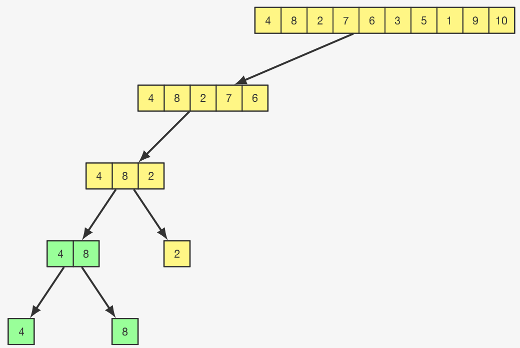
Simulação



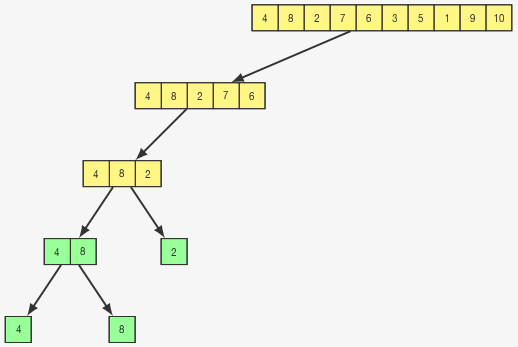
Simulação



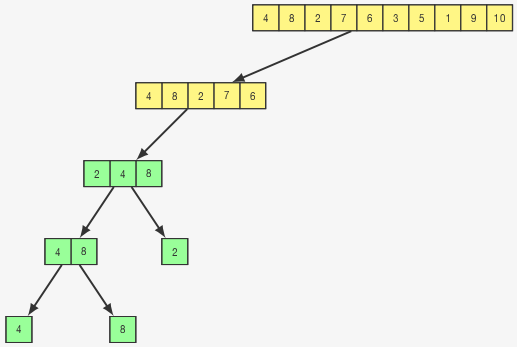
Simulação



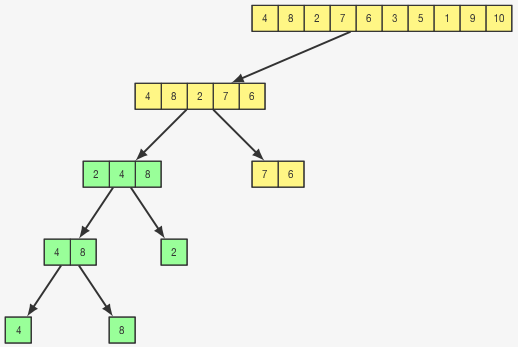
Simulação



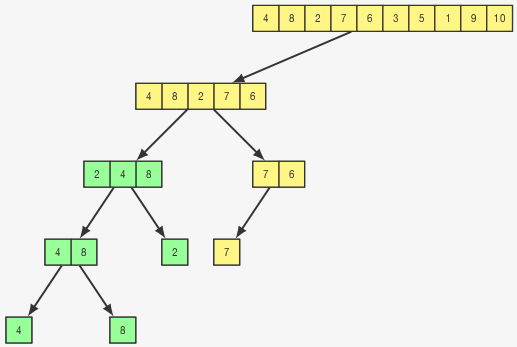
Simulação



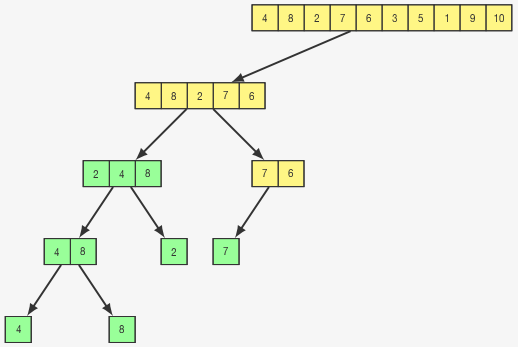
Simulação



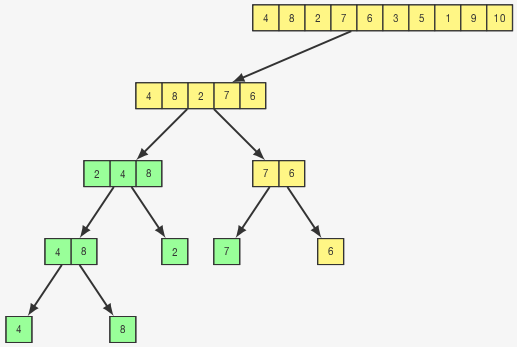
Simulação



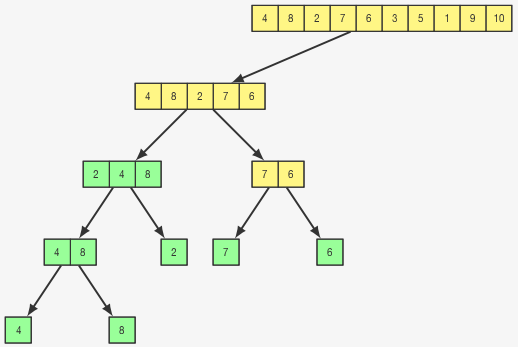
Simulação



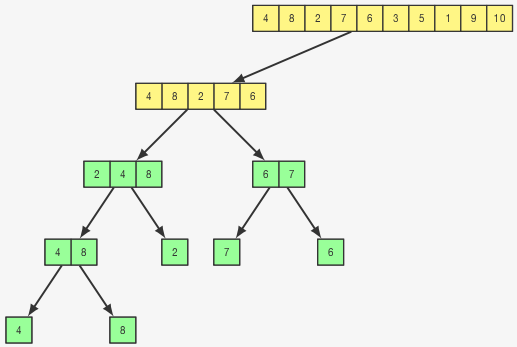
Simulação



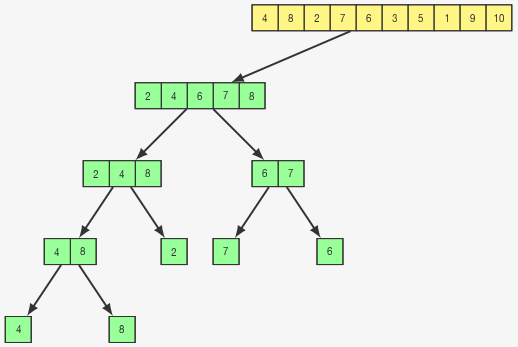
Simulação



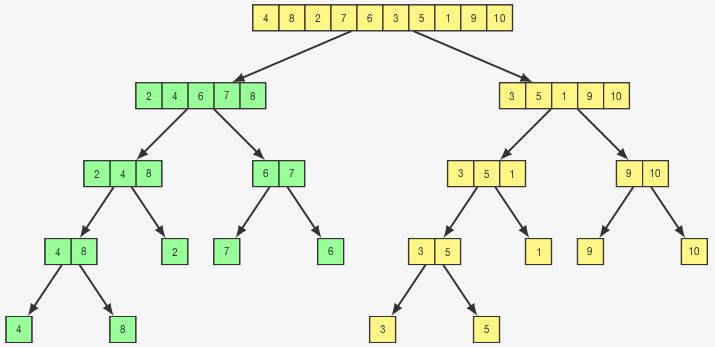
Simulação



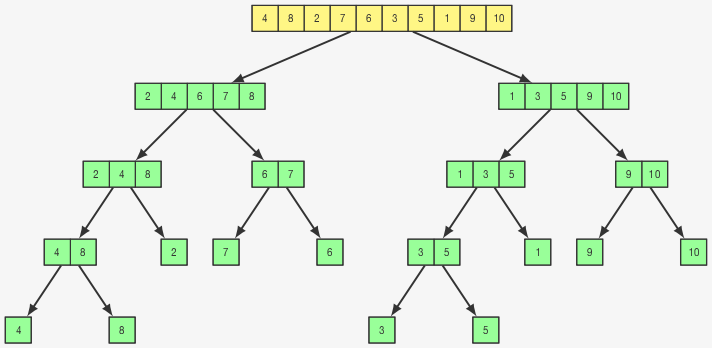
Simulação



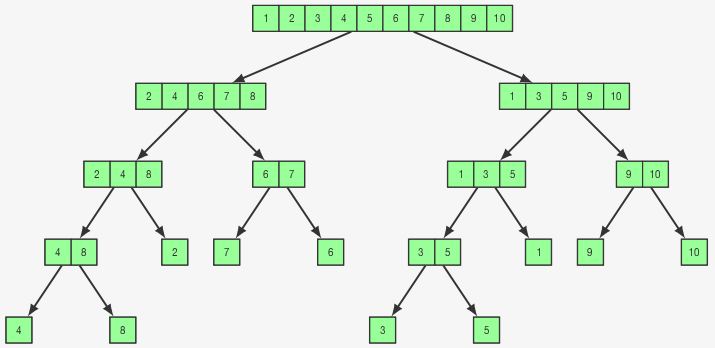
Simulação



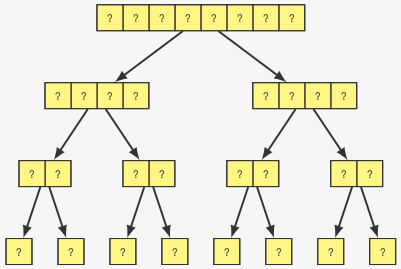
Simulação



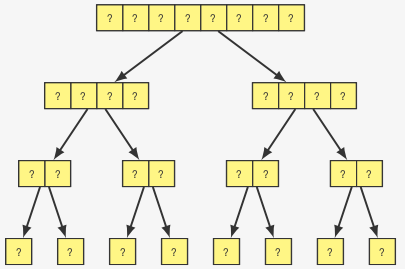
Simulação



Tempo de execução para $n = 2^l$

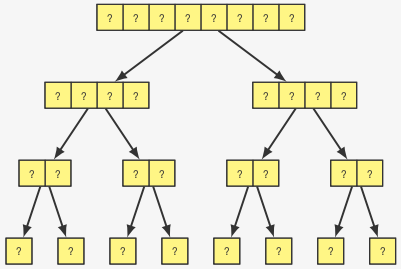


Tempo de execução para $n = 2^l$



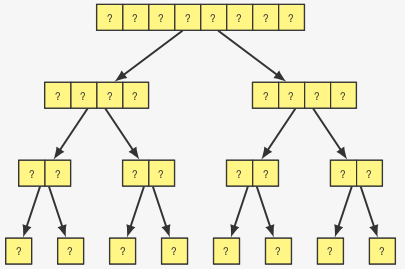
- No primeiro nível fazemos um `merge` com n elementos

Tempo de execução para $n = 2^l$



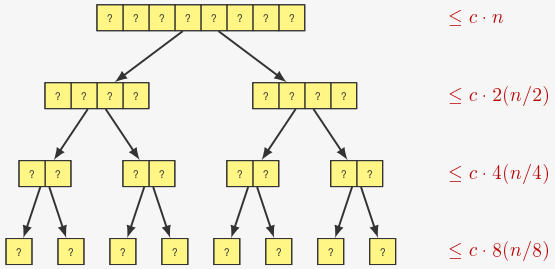
- No primeiro nível fazemos um `merge` com n elementos
- No segundo fazemos dois `merge` com $n/2$ elementos

Tempo de execução para $n = 2^l$



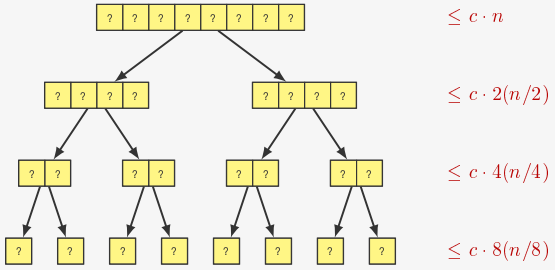
- No primeiro nível fazemos um `merge` com n elementos
- No segundo fazemos dois `merge` com $n/2$ elementos
- No k-ésimo fazemos 2^k `merge` com $n/2^k$ elementos

Tempo de execução para $n = 2^l$



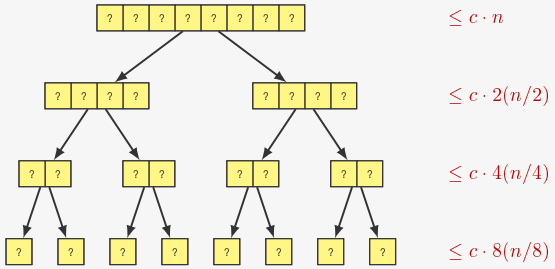
- No primeiro nível fazemos um `merge` com n elementos
- No segundo fazemos dois `merge` com $n/2$ elementos
- No k-ésimo fazemos 2^k `merge` com $n/2^k$ elementos
- No último gastamos tempo constante n vezes

Tempo de execução para $n = 2^l$



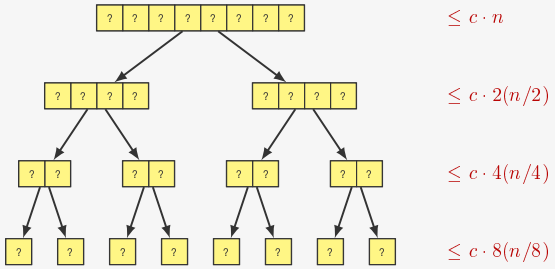
- No nível k gastamos tempo $\leq c \cdot n$

Tempo de execução para $n = 2^l$



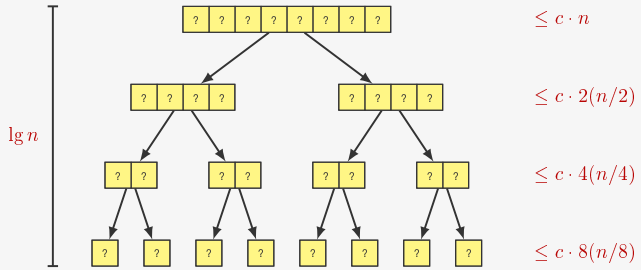
- No nível k gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?

Tempo de execução para $n = 2^l$



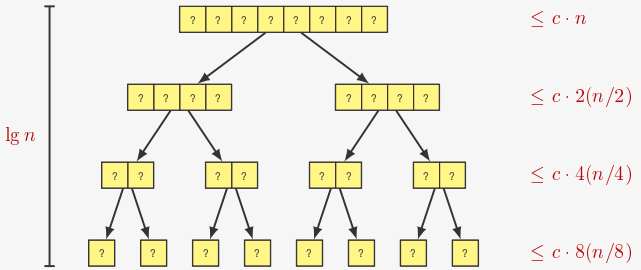
- No nível k gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?
  - Dividimos n por 2 até que fique menor igual a 1

Tempo de execução para $n = 2^l$



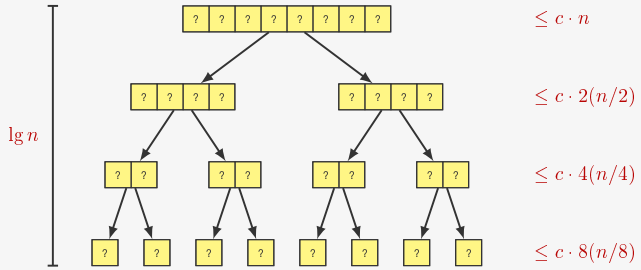
- No nível k gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?
  - Dividimos n por 2 até que fique menor igual a 1
  - Ou seja, $l = \log_2 n$

Tempo de execução para $n = 2^l$



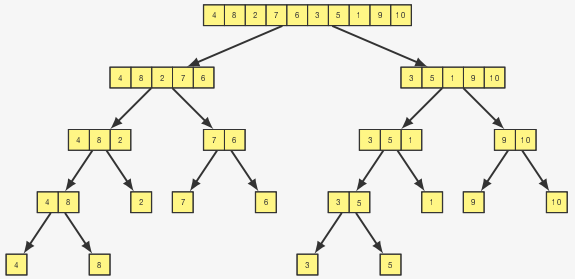
- No nível k gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?
  - Dividimos n por 2 até que fique menor igual a 1
  - Ou seja, $l = \log_2 n$
- Como $\log_2 n$ é muito comum, escrevemos $\lg n$

Tempo de execução para $n = 2^l$

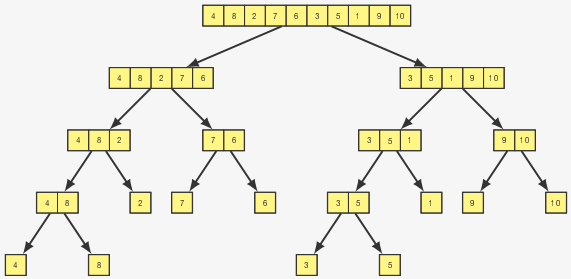


- No nível k gastamos tempo $\leq c \cdot n$
- Quantos níveis temos?
  - Dividimos n por 2 até que fique menor igual a 1
  - Ou seja, $l = \log_2 n$
- Como $\log_2 n$ é muito comum, escrevemos $\lg n$
- Tempo total: $cn \lg n = \mathrm{O}(n \lg n)$

Tempo de execução para n qualquer

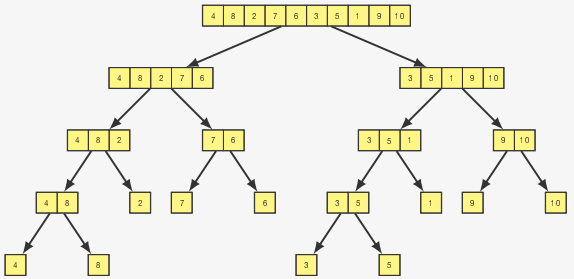


Tempo de execução para n qualquer



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

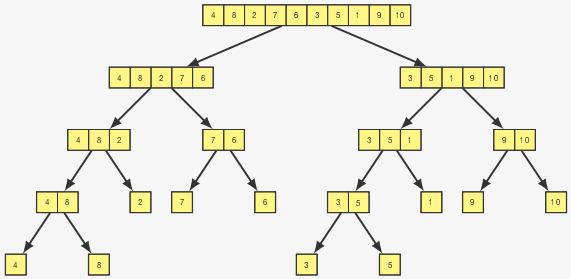
Tempo de execução para n qualquer



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

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

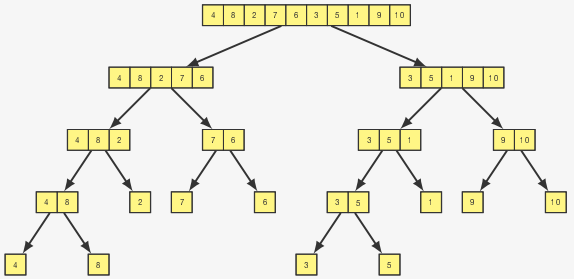
Tempo de execução para n qualquer



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

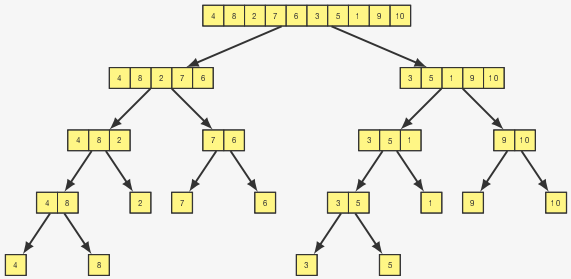
Tempo de execução para n qualquer



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 $2^{k-1} < n < 2^k$

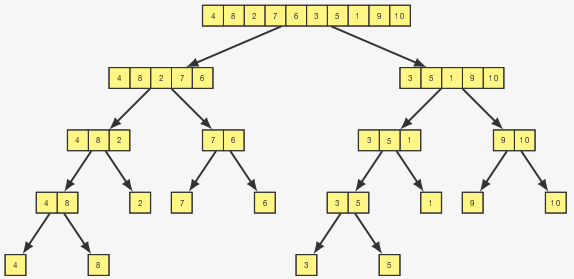
Tempo de execução para n qualquer



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, $2^k < 2n$

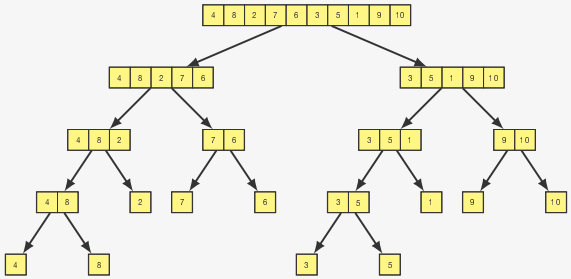
Tempo de execução para n qualquer



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

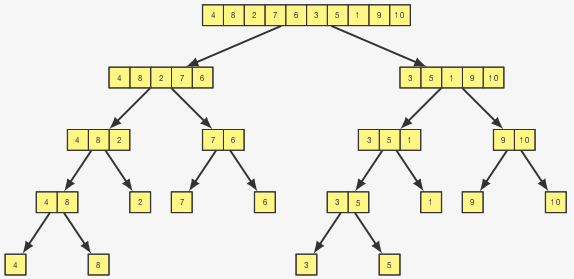


Qual o tempo de execução para n que não é potência de 2?

- 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

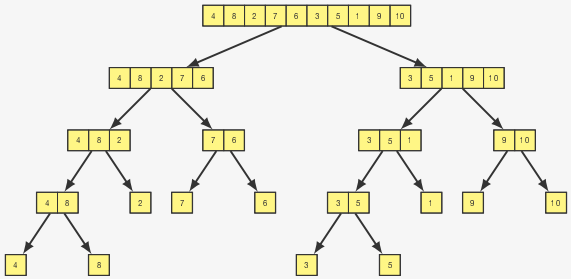


Qual o tempo de execução para n que não é potência de 2?

- 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

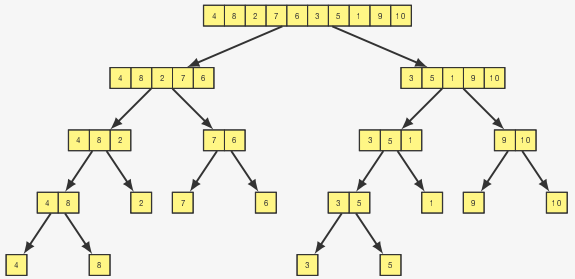


Qual o tempo de execução para n que não é potência de 2?

- 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

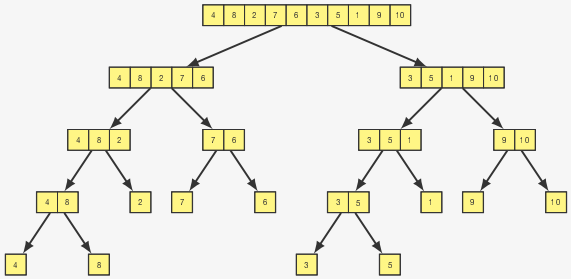


Qual o tempo de execução para n que não é potência de 2?

- 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

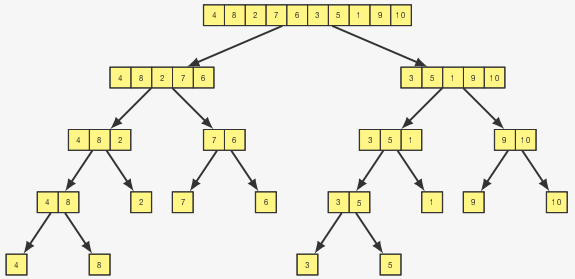


Qual o tempo de execução para n que não é potência de 2?

- 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

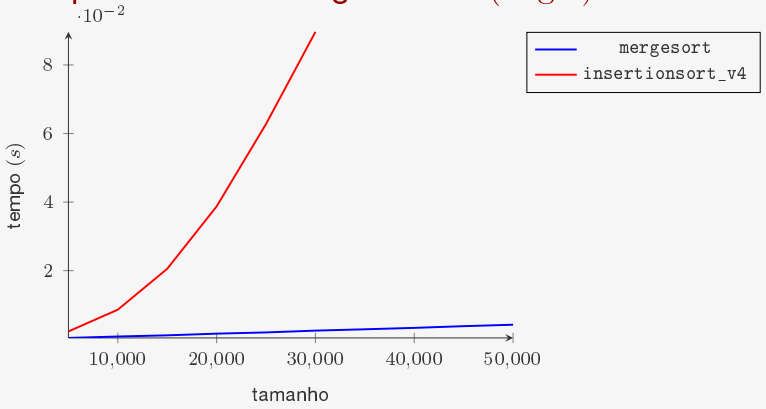


Qual o tempo de execução para n que não é potência de 2?

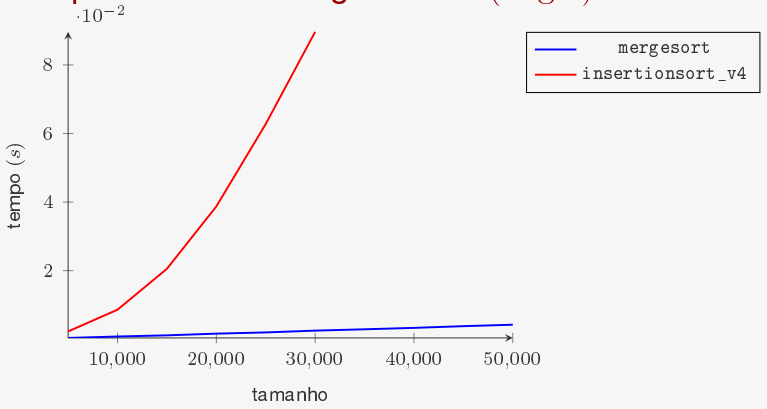
- Seja 2^k a próxima potência de 2 depois de n
  - Ex: Se $n = 3000$, a próxima potência é 4096
- Temos que $2^{k-1} < n < 2^k$
  - Ou seja, $2^k < 2n$
- O tempo de execução para n é menor do que

$$c\,2^k \lg 2^k \leq 2cn \lg(2n) = 2cn(\lg 2 + \lg n) = 2cn + 2cn \lg n = \mathrm{O}(n \lg n)$$

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

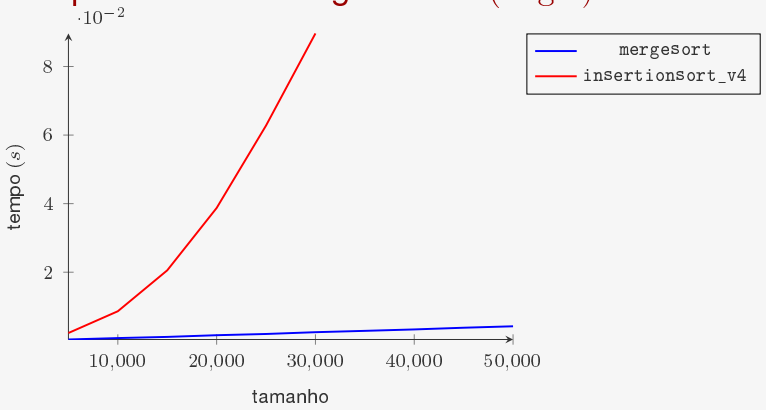


Valeu a pena fazer um algoritmo $\mathrm{O}(n \lg n)$?



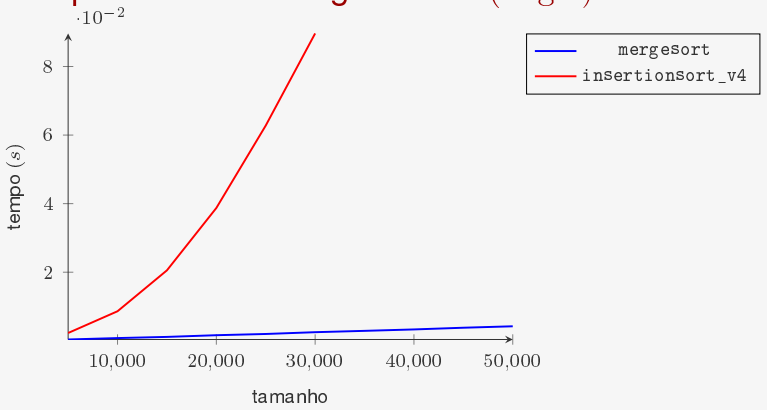
- `insertionsort_v4` ordena 30.000 números em $0.0896s$

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



- `insertionsort_v4` ordena 30.000 números em $0.0896s$
- `mergesort` ordena 800.000 números em $0.0874s$

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



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