```
int main(int argc, char** argv)
{
  char c = 0;
  char* commands = "ads pq"; // key commands: "left,right,rotate,confirm,pause,quit"
  int speed = 2; // sets max moves per row
  int moves_to_go = 2;
  int full = 0; // whether board is full
  init(); // initialize board an tetrominoes


  cur =
  state[
  state[

  system
  printf

  while
    {
      dr

    // process user action
    c = getchar();   // get new action
    if (c == commands[0] && !intersect(cur, state[0]-1, state[1])) state[0]--; // move left
    if (c == commands[1] && !intersect(cur, state[0]+1, state[1])) state[0]++; // move right
    if (c == commands[2] && !intersect(cur->rotated, state[0], state[1])) cur = cur->rotated;
    if (c == commands[3]) moves_to_go=0;

    // scroll down
    if (!moves_to_go--)
      {
        if (intersect(cur,state[0],state[1]+1)) // if tetromino intersected with sth
          {
            cramp_tetromino();
            remove_complete_lines();
            cur = &tetrominoes[rand() % NUM_POSES];
            state[0] = (WIDTH - cur->width)/2;
```

MAC122 - PRINCÍPIOS DE DESENVOLVIMENTO DE ALGORITMOS

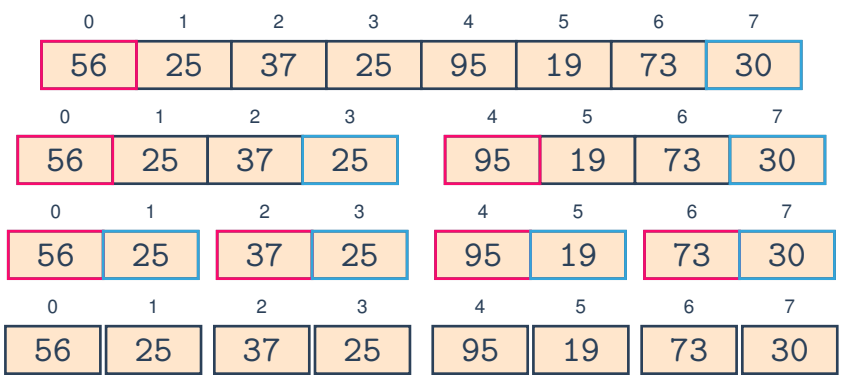
Ordenação por intercalação

QUAL A COMPLEXIDADE DO ALGORITMO ABAIXO

```
1  /* Calcula a soma dos elementos em v[e..d] */
2  int SomaR (int v[], int e, int d) {
3   int x, y, c;
4   if (e == d) return v[e];
5   else if (e > d) return 0;
6   c = (e+d)/2;
7   x = SomaR(v, e, c);
8   y = SomaR(v, c + 1, d);
9   return x + y;
10 }
11
12 int Soma(int v[], int n) {
13  return SomaR(v, 0, n-1);
14 }
```

Complexidade de espaço

```
1 int SomaR (int v[], int e, int d) {
2   int x, y, c;
3   if (e == d) return v[e];
4   else if (e > d) return 0;
5   c = (e+d)/2;
6   x = SomaR(v, e, c);
7   y = SomaR(v, c + 1, d);
8   return x + y;
9 }
```



```
1 int SomaR (int v[], int e, int d) {
2  int x, y, c;
3  if (e == d) return v[e];
4  else if (e > d) return 0;
5  c = (e+d)/2;
6  x = SomaR(v, e, c);
7  y = SomaR(v, c + 1, d);
8  return x + y;
9 }
```

- Tamanho da instância $n = d - e + 1$
- Profundidade de recursão: $\lfloor \log_2 n \rfloor$
- Espaço $O(\log n)$

Dica: assuma que $n = 2^k$

$T(2^k) =$

```
1 int SomaR (int v[], int e, int d) {
2  int x, y, c;
3  if (e == d) return v[e];
4  else if (e > d) return 0;
5  c = (e+d)/2;
6  x = SomaR(v, e, c);
7  y = SomaR(v, c + 1, d);
8  return x + y;
9 }
```

```
1 int SomaR (int v[], int e, int d) {
2  int x, y, c;
3  if (e == d) return v[e];
4  else if (e > d) return 0;
5  c = (e+d)/2;
6  x = SomaR(v, e, c);
7  y = SomaR(v, c + 1, d);
8  return x + y;
9 }
```

Dica: assuma que $n = 2^k$

$$T(2^k) = 2T(2^{k-1}) + 1$$

```
1 int SomaR (int v[], int e, int d) {
2  int x, y, c;
3  if (e == d) return v[e];
4  else if (e > d) return 0;
5  c = (e+d)/2;
6  x = SomaR(v, e, c);
7  y = SomaR(v, c + 1, d);
8  return x + y;
9 }
```

Dica: assuma que $n = 2^k$

$$T(2^k) = 2T(2^{k-1}) + 1$$
$$= 2^2 T(2^{k-2}) + 2 + 1$$

```
1 int SomaR (int v[], int e, int d) {
2  int x, y, c;
3  if (e == d) return v[e];
4  else if (e > d) return 0;
5  c = (e+d)/2;
6  x = SomaR(v, e, c);
7  y = SomaR(v, c + 1, d);
8  return x + y;
9 }
```

Dica: assuma que $n = 2^k$

$$
\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + 1 \\
&= 2^2 T(2^{k-2}) + 2 + 1 \\
&= 2^3 T(2^{k-3}) + 2^2 + 2 + 1
\end{aligned}
$$

```
1 int SomaR (int v[], int e, int d) {
2   int x, y, c;
3   if (e == d) return v[e];
4   else if (e > d) return 0;
5   c = (e+d)/2;
6   x = SomaR(v, e, c);
7   y = SomaR(v, c + 1, d);
8   return x + y;
9 }
```

Dica: assuma que $n = 2^k$

$$
\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + 1 \\
&= 2^2 T(2^{k-2}) + 2 + 1 \\
&= 2^3 T(2^{k-3}) + 2^2 + 2 + 1 \\
&\vdots \\
&= 2^k T(1) + \sum_{i=0}^{k-1} 2^i \\
&= 2^k + 2^k - 1
\end{aligned}
$$

```
1 int SomaR (int v[], int e, int d) {
2   int x, y, c;
3   if (e == d) return v[e];
4   else if (e > d) return 0;
5   c = (e+d)/2;
6   x = SomaR(v, e, c);
7   y = SomaR(v, c + 1, d);
8   return x + y;
9 }
```

Dica: assuma que $n = 2^k$

$$
\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + 1 \\
&= 2^2 T(2^{k-2}) + 2 + 1 \\
&= 2^3 T(2^{k-3}) + 2^2 + 2 + 1 \\
&\vdots \\
&= 2^k T(1) + \sum_{i=0}^{k-1} 2^i \\
&= 2^k + 2^k - 1 \in \boxed{O(n)}
\end{aligned}
$$

Correção

```
1 int SomaR (int v[], int e, int d) {
2  int x, y, c;
3  if (e == d) return v[e];
4  else if (e > d) return 0;
5  c = (e+d)/2;
6  x = SomaR(v, e, c);
7  y = SomaR(v, c + 1, d);
8  return x + y;
9 }
```

```
1 int SomaR (int v[], int e, int d) {
2  int x, y, c;
3  if (e == d) return v[e];
4  else if (e > d) return 0;
5  c = (e+d)/2;
6  x = SomaR(v, e, c);
7  y = SomaR(v, c + 1, d);
8  return x + y;
9 }
```

Prova por indução no tamanho do vetor $n = \mathtt{d} - \mathtt{e} + 1 = 2^k$

- Base: para $k = 0$, função retorna soma de 1 elemento
- Passo indutivo: se função é correta para 2^k, então função é correta para 2^{k+1}

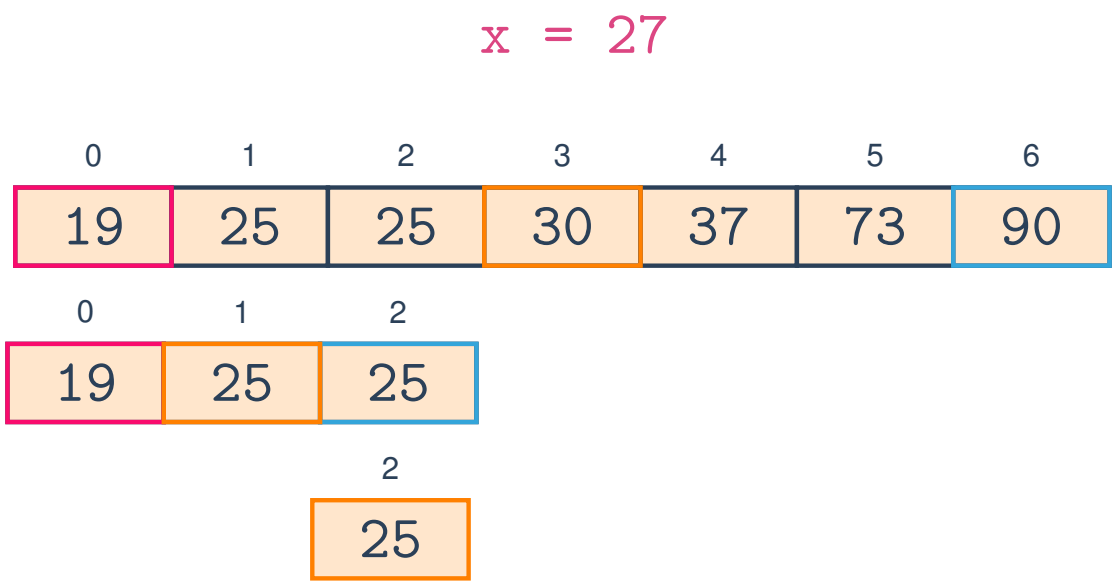
Busca binária recursiva

```
1  int BuscaBinariaR (int x, int v[], int e, int d) {
2   if (e > d) return -1;
3   c = (e+d)/2;
4   if (v[c] == x) return c;
5   else if (v[c] > x) return BuscaBinariaR(x, v, /* ? */);
6   else return BuscaBinariaR(x, v, /* ? */);
7  }
8
9  int BuscaBinaria (int x, int v[], int n) {
10  return BuscaBinariaR(x, v, /* ? */);
11 }
```

Busca binária recursiva

```
1  int BuscaBinariaR (int x, int v[], int e, int d) {
2   if (e > d) return -1;
3   c = (e+d)/2;
4   if (v[c] == x) return c;
5   else if (v[c] > x) return BuscaBinariaR(x, v, e, c-1);
6   else return BuscaBinariaR(x, v, c+1, d);
7  }
8
9  int BuscaBinaria (int x, int v[], int n) {
10  return BuscaBinariaR(x, v, 0, n);
11 }
```

Busca binária recursiva



Busca binária recursiva: Complexidade

```
1  int BuscaBinariaR (int x, int v[], int e, int d) {
2   if (e > d) return -1;
3   c = (e+d)/2;
4   if (v[c] == x) return c;
5   else if (v[c] > x) return BuscaBinariaR(x, v, /* ? */);
6   else return BuscaBinariaR(x, v, /* ? */);
7  }
8
9  int BuscaBinaria (int x, int v[], int n) {
10  return BuscaBinariaR(x, v, /* ? */);
11 }
```

Espaço:

Busca binária recursiva: Complexidade

```
1  int BuscaBinariaR (int x, int v[], int e, int d) {
2   if (e > d) return -1;
3   c = (e+d)/2;
4   if (v[c] == x) return c;
5   else if (v[c] > x) return BuscaBinariaR(x, v, /* ? */);
6   else return BuscaBinariaR(x, v, /* ? */);
7  }
8
9  int BuscaBinaria (int x, int v[], int n) {
10  return BuscaBinariaR(x, v, /* ? */);
11 }
```

Espaço: $O(\log n)$

Busca binária recursiva: Complexidade de tempo

Dica: assuma que $n = 2^k$

```
1 int BuscaBinariaR(int x, int v[],
      int e, int d) {
2  if (e > d) return -1;
3  c = (e+d)/2;
4  if (v[c] == x) return c;
5  else if (v[c] > x)
6    return BuscaBinariaR(x,v,e,c-1);
7  else
8    return BuscaBinariaR(x,v,c+1,d);
9 }
```

$$T(2^k) =$$

Busca binária recursiva: Complexidade de tempo

```
1 int BuscaBinariaR(int x, int v[],
      int e, int d) {
2  if (e > d) return -1;
3  c = (e+d)/2;
4  if (v[c] == x) return c;
5  else if (v[c] > x)
6    return BuscaBinariaR(x,v,e,c-1);
7  else
8    return BuscaBinariaR(x,v,c+1,d);
9 }
```

Dica: assuma que $n = 2^k$

$$T(2^k) = T(2^{k-1}) + 1$$

Busca binária recursiva: Complexidade de tempo

```
1 int BuscaBinariaR(int x, int v[],
      int e, int d) {
2  if (e > d) return -1;
3  c = (e+d)/2;
4  if (v[c] == x) return c;
5  else if (v[c] > x)
6    return BuscaBinariaR(x,v,e,c-1);
7  else
8    return BuscaBinariaR(x,v,c+1,d);
9 }
```

Dica: assuma que $n = 2^k$

$$T(2^k) = T(2^{k-1}) + 1$$
$$= T(2^{k-2}) + 1 + 1$$

Busca binária recursiva: Complexidade de tempo

```
1 int BuscaBinariaR(int x, int v[],
      int e, int d) {
2  if (e > d) return -1;
3  c = (e+d)/2;
4  if (v[c] == x) return c;
5  else if (v[c] > x)
6    return BuscaBinariaR(x,v,e,c-1);
7  else
8    return BuscaBinariaR(x,v,c+1,d);
9 }
```

Dica: assuma que $n = 2^k$

$$
\begin{aligned}
T(2^k) &= T(2^{k-1}) + 1 \\
&= T(2^{k-2}) + 1 + 1 \\
&= T(2^{k-3}) + 1 + 1 + 1
\end{aligned}
$$

Busca binária recursiva: Complexidade de tempo

```
1 int BuscaBinariaR(int x, int v[],
      int e, int d) {
2  if (e > d) return -1;
3  c = (e+d)/2;
4  if (v[c] == x) return c;
5  else if (v[c] > x)
6    return BuscaBinariaR(x,v,e,c-1);
7  else
8    return BuscaBinariaR(x,v,c+1,d);
9 }
```

Dica: assuma que $n = 2^k$

$$
\begin{aligned}
T(2^k) &= T(2^{k-1}) + 1 \\
&= T(2^{k-2}) + 1 + 1 \\
&= T(2^{k-3}) + 1 + 1 + 1 \\
&\vdots \\
&= T(1) + k
\end{aligned}
$$

Busca binária recursiva: Complexidade de tempo

```
1 int BuscaBinariaR(int x, int v[],
      int e, int d) {
2  if (e > d) return -1;
3  c = (e+d)/2;
4  if (v[c] == x) return c;
5  else if (v[c] > x)
6    return BuscaBinariaR(x,v,e,c-1);
7  else
8    return BuscaBinariaR(x,v,c+1,d);
9 }
```

Dica: assuma que $n = 2^k$

$$
\begin{aligned}
T(2^k) &= T(2^{k-1}) + 1 \\
&= T(2^{k-2}) + 1 + 1 \\
&= T(2^{k-3}) + 1 + 1 + 1 \\
&\vdots \\
&= T(1) + k \in \boxed{O(\log n)}
\end{aligned}
$$

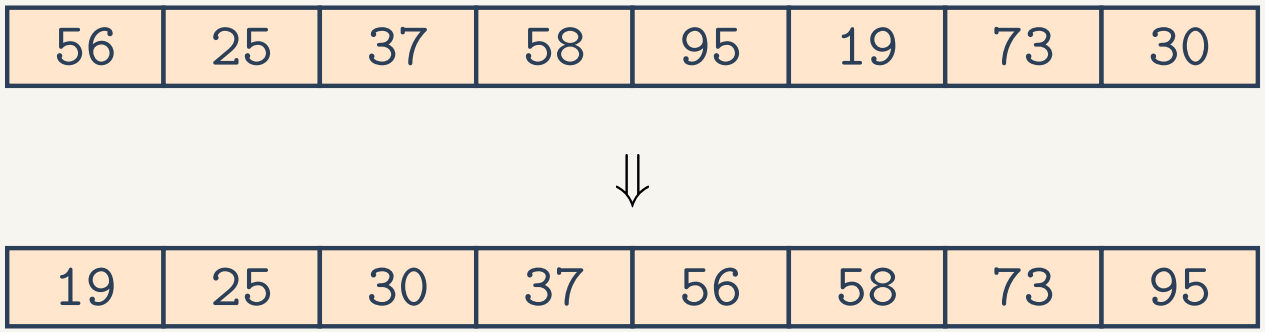
Rearranjar os elementos de uma lista $x_1, \ldots, x_n$ de tal modo que fiquem em ordem não descrescente: $x_1 \leq x_2 \leq \cdots \leq x_n$

56	25	37	58	95	19	73	30

$\Downarrow$

19	25	30	37	56	58	73	95

Rearranjar os elementos de uma lista $x_1, \ldots, x_n$ de tal modo que fiquem em ordem não descrescente: $x_1 \leq x_2 \leq \cdots \leq x_n$



- Elementos x_i são genéricos (inteiros, reais, strings, registros) porém comparáveis, isto é, $x_i \leq x_j$ é bem definido para quaisquer elementos
- Lista é representada como vetor de tamanho fixo

- Por seleção
- Por inserção

Complexidade de tempo: $O(n^2)$ no pior caso

Ordenação por intercalação (Mergesort)

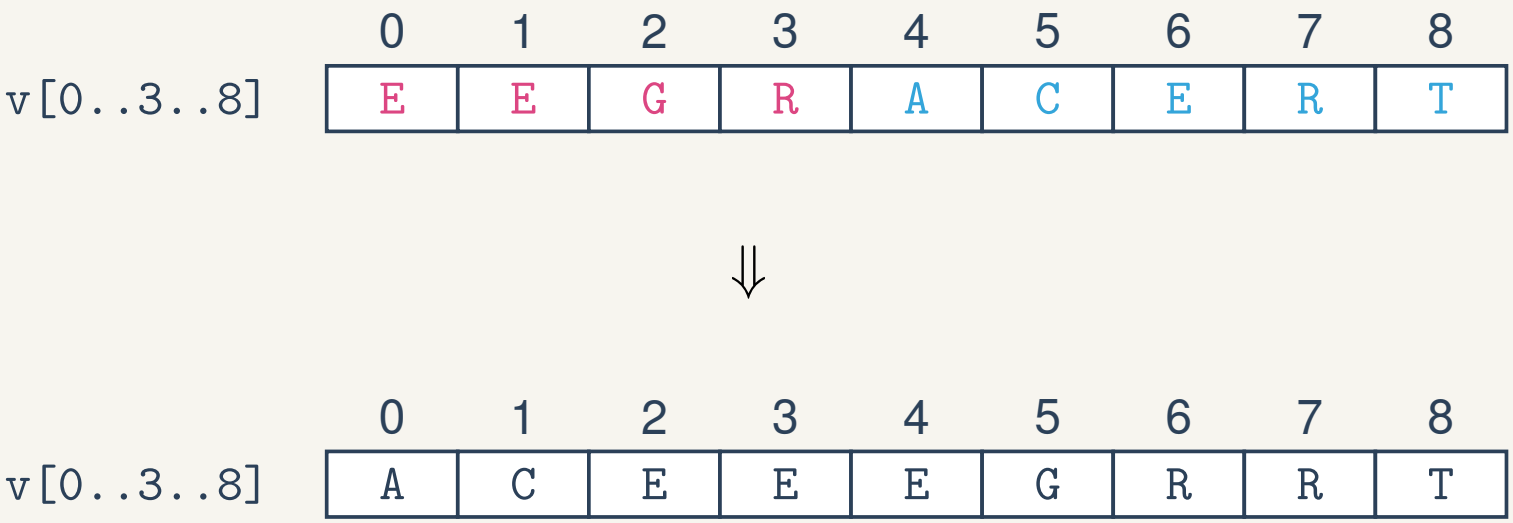
Algoritmo “divide-e-conquista”:

1. Divida vetor em duas partes contíguas de tamanhos aproximados
2. Ordene recursivamente cada parte
3. Intercale as duas metades ordenadas para formar um vetor crescente

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

INTERCALAÇÃO DE VETORES CRESCENTES

Passo principal: Dado vetor `v[e..c..d]` tal que `v[e..c]` e `v[c+1..d]` são crescentes, obter permutação crescente de `v[e..d]`



Algoritmo:

1. Copiar vetor `v[e..d]` para vetor auxiliar `aux[0..n-1]`
2. Para `k=e` a `k=d` faça:
   - 2.1 Se `aux[j] < aux[i]`, copie `aux[j]` para `v[k]` e incremente `j`
   - 2.2 Caso contrário, copie `aux[i]` para `v[k]` e incremente `i`

	0	1	2	3	4	5	6	7	8
v	E	E	G	R	A	C	E	R	T

	0	1	2	3	4	5	6	7	8
aux	E	E	G	R	A	C	E	R	T

Algoritmo:

1. Copiar vetor `v[e..d]` para vetor auxiliar `aux[0..n-1]`
2. Para `k=e` a `k=d` faça:
   - 2.1 Se `aux[j] < aux[i]`, copie `aux[j]` para `v[k]` e incremente `j`
   - 2.2 Caso contrário, copie `aux[i]` para `v[k]` e incremente `i`

	0	1	2	3	4	5	6	7	8
v	A	E	G	R	A	C	E	R	T

	0	1	2	3	4	5	6	7	8
aux	E	E	G	R	A	C	E	R	T

Algoritmo:

1. Copiar vetor `v[e..d]` para vetor auxiliar `aux[0..n-1]`
2. Para `k=e` a `k=d` faça:
   - 2.1 Se `aux[j] < aux[i]`, copie `aux[j]` para `v[k]` e incremente `j`
   - 2.2 Caso contrário, copie `aux[i]` para `v[k]` e incremente `i`

	0	1	2	3	4	5	6	7	8
v	A	C	G	R	A	C	E	R	T

	0	1	2	3	4	5	6	7	8
aux	E	E	G	R	A	C	E	R	T

Algoritmo:

1. Copiar vetor `v[e..d]` para vetor auxiliar `aux[0..n-1]`
2. Para `k=e` a `k=d` faça:
   - 2.1 Se `aux[j] < aux[i]`, copie `aux[j]` para `v[k]` e incremente `j`
   - 2.2 Caso contrário, copie `aux[i]` para `v[k]` e incremente `i`

	0	1	2	3	4	5	6	7	8
v	A	C	E	R	A	C	E	R	T

	0	1	2	3	4	5	6	7	8
aux	E	E	G	R	A	C	E	R	T

Algoritmo:

1. Copiar vetor `v[e..d]` para vetor auxiliar `aux[0..n-1]`
2. Para `k=e` a `k=d` faça:
   - 2.1 Se `aux[j] < aux[i]`, copie `aux[j]` para `v[k]` e incremente `j`
   - 2.2 Caso contrário, copie `aux[i]` para `v[k]` e incremente `i`

	0	1	2	3	4	5	6	7	8
v	A	C	E	E	A	C	E	R	T

	0	1	2	3	4	5	6	7	8
aux	E	E	G	R	A	C	E	R	T

Algoritmo:

1. Copiar vetor `v[e..d]` para vetor auxiliar `aux[0..n-1]`
2. Para `k=e` a `k=d` faça:
   - 2.1 Se `aux[j] < aux[i]`, copie `aux[j]` para `v[k]` e incremente `j`
   - 2.2 Caso contrário, copie `aux[i]` para `v[k]` e incremente `i`

	0	1	2	3	4	5	6	7	8
v	A	C	E	E	E	E	E	R	T

	0	1	2	3	4	5	6	7	8
aux	E	E	G	R	A	C	E	R	T

Algoritmo:

1. Copiar vetor `v[e..d]` para vetor auxiliar `aux[0..n-1]`
2. Para `k=e` a `k=d` faça:
   - 2.1 Se `aux[j] < aux[i]`, copie `aux[j]` para `v[k]` e incremente `j`
   - 2.2 Caso contrário, copie `aux[i]` para `v[k]` e incremente `i`

	0	1	2	3	4	5	6	7	8
v	A	C	E	E	E	G	E	R	T

	0	1	2	3	4	5	6	7	8
aux	E	E	G	R	A	C	E	R	T

Algoritmo:

1. Copiar vetor `v[e..d]` para vetor auxiliar `aux[0..n-1]`
2. Para `k=e` a `k=d` faça:
   - 2.1 Se `aux[j] < aux[i]`, copie `aux[j]` para `v[k]` e incremente `j`
   - 2.2 Caso contrário, copie `aux[i]` para `v[k]` e incremente `i`

	0	1	2	3	4	5	6	7	8
v	A	C	E	E	E	G	R	R	T

	0	1	2	3	4	5	6	7	8
aux	E	E	G	R	A	C	E	R	T

Algoritmo:

1. Copiar vetor `v[e..d]` para vetor auxiliar `aux[0..n-1]`
2. Para `k=e` a `k=d` faça:
   - 2.1 Se `aux[j] < aux[i]`, copie `aux[j]` para `v[k]` e incremente `j`
   - 2.2 Caso contrário, copie `aux[i]` para `v[k]` e incremente `i`

	0	1	2	3	4	5	6	7	8
v	A	C	E	E	E	G	R	R	T

	0	1	2	3	4	5	6	7	8
aux	E	E	G	R	A	C	E	R	T

Algoritmo:

1. Copiar vetor `v[e..d]` para vetor auxiliar `aux[0..n-1]`
2. Para `k=e` a `k=d` faça:
   - 2.1 Se `aux[j] < aux[i]`, copie `aux[j]` para `v[k]` e incremente `j`
   - 2.2 Caso contrário, copie `aux[i]` para `v[k]` e incremente `i`

	0	1	2	3	4	5	6	7	8
v	A	C	E	E	E	G	R	R	T

	0	1	2	3	4	5	6	7	8
aux	E	E	G	R	A	C	E	R	T

INTERCALAÇÃO DE VETORES CRESCENTES

```
1  /* Intercala vetores crescentes v[e..c] e v[c+1..d] */
2  void intercala(int v[], int e, int c, int d) {
3    const int n = (d-e+1);
4    /* 1. Copia para vetor auxiliar */
5    int *aux = malloc( n*sizeof(int) );
6    for (int k = 0; k < n; k++) aux[k] = v[e+k];
7    c -= e;
8    /* 2. Copia de volta para v em ordem crescente */
9    int i = 0;     /* cursor da metade esquerda */
10   int j = c + 1; /* cursor da metade direita */
11   for (int k = e; k <= d; k++) {
12     if       (i > c)           v[k] = aux[j++];/* fim esq */
13     else if (j >= n)           v[k] = aux[i++];/* fim dir */
14     else if (aux[j] < aux[i]) v[k] = aux[j++];
15     else                       v[k] = aux[i++];
16   }
17   free(aux);
18 }
```

INTERCALAÇÃO DE VETORES CRESCENTES

```
1  /* Intercala vetores crescentes v[e..c] e v[c+1..d] */
2  void intercala(int v[], int e, int c, int d) {
3    const int n = (d-e+1);
4    /* 1. Copia para vetor auxiliar */
5    int *aux = malloc( n*sizeof(int) );◀
6    for (int k = 0; k < n; k++) aux[k] = v[e+k];
7    c -= e;
8    /* 2. Copia de volta para v em ordem crescente */
9    int i = 0;     /* cursor da metade esquerda */
10   int j = c + 1; /* cursor da metade direita */
11   for (int k = e; k <= d; k++) {
12     if       (i > c)           v[k] = aux[j++];/* fim esq */
13     else if (j >= n)           v[k] = aux[i++];/* fim dir */
14     else if (aux[j] < aux[i]) v[k] = aux[j++];
15     else                       v[k] = aux[i++];
16   }
17   free(aux);◀
18 }
```

INTERCALAÇÃO DE VETORES CRESCENTES

```
1  /* Recebe vetores v e aux de mesmos tamanhos */
2  void intercala(int v[], int e, int c, int d, int aux[]) {
3    /* 1. Copia para vetor auxiliar */
4    for (int k = e; k <= d; k++) aux[k] = v[k];
5    /* 2. Copia de volta para v em ordem crescente */
6    int i = e;     /* cursor da metade esquerda */
7    int j = c + 1; /* cursor da metade direita */
8    for (int k = e; k <= d; k++) {
9      if      (i > c)           v[k] = aux[j++];/* fim esq */
10     else if (j > d)           v[k] = aux[i++];/* fim dir */
11     else if (aux[j] < aux[i]) v[k] = aux[j++];
12     else                      v[k] = aux[i++];
13   }
14 }
```

Complexidade de tempo:

INTERCALAÇÃO DE VETORES CRESCENTES

```
1  /* Recebe vetores v e aux de mesmos tamanhos */
2  void intercala(int v[], int e, int c, int d, int aux[]) {
3    /* 1. Copia para vetor auxiliar */
4    for (int k = e; k <= d; k++) aux[k] = v[k];
5    /* 2. Copia de volta para v em ordem crescente */
6    int i = e;     /* cursor da metade esquerda */
7    int j = c + 1; /* cursor da metade direita */
8    for (int k = e; k <= d; k++) {
9      if      (i > c)           v[k] = aux[j++];/* fim esq */
10     else if (j > d)           v[k] = aux[i++];/* fim dir */
11     else if (aux[j] < aux[i]) v[k] = aux[j++];
12     else                      v[k] = aux[i++];
13   }
14 }
```

Complexidade de tempo: $O(n)$

Ordenação por intercalação

```
1  void ordenaR(int v[], int e, int d, int aux[]) {
2    if (e >= d) return; /* ordenado */
3    int c = (e + d)/2;
4    ordenaR(v, e, c, aux);
5    ordenaR(v, c+1, d, aux);
6    intercala(v, e, c, d, aux);
7  }
8
9  void ordena(int v[], int n) {
10   int *aux = malloc(n*sizeof(int));
11   ordenaR(v, 0, n-1, aux);
12 }
```

Simulação

	a[]															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	M	E	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 0, 1)	E	M	R	G	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 2, 2, 3)	E	M	G	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 0, 1, 3)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 4, 5)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 6, 6, 7)	E	G	M	R	E	S	O	R	T	E	X	A	M	P	L	E
merge(a, aux, 4, 5, 7)	E	G	M	R	E	O	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 0, 3, 7)	E	E	G	M	O	R	R	S	T	E	X	A	M	P	L	E
merge(a, aux, 8, 8, 9)	E	E	G	M	O	R	R	S	E	T	X	A	M	P	L	E
merge(a, aux, 10, 10, 11)	E	E	G	M	O	R	R	S	E	T	A	X	M	P	L	E
merge(a, aux, 8, 9, 11)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 12, 12, 13)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	L	E
merge(a, aux, 14, 14, 15)	E	E	G	M	O	R	R	S	A	E	T	X	M	P	E	L
merge(a, aux, 12, 13, 15)	E	E	G	M	O	R	R	S	A	E	T	X	E	L	M	P
merge(a, aux, 8, 11, 15)	E	E	G	M	O	R	R	S	A	E	E	L	M	P	T	X
merge(a, aux, 0, 7, 15)	A	E	E	E	E	G	L	M	M	O	P	R	R	S	T	X

lo hi

result after recursive call

Visualização:

https://www.toptal.com/developers/sorting-algorithms/merge-sort

MERGESORT: COMPLEXIDADE DE TEMPO

```
1 void ordenaR(int v[], int e, int d,
2                       int aux[]) {
3   if (e >= d) return;
4   int c = (e + d)/2;
5   ordenaR(v,e,c,aux);
6   ordenaR(v,c+1,d,aux);
7   intercala(v,e,c,d,aux);/* T=n */
8 }
```

Assumindo que $n = 2^k$

$T(2^k) =$

MERGESORT: COMPLEXIDADE DE TEMPO

```
1 void ordenaR(int v[], int e, int d,
2                       int aux[]) {
3   if (e >= d) return;
4   int c = (e + d)/2;
5   ordenaR(v,e,c,aux);
6   ordenaR(v,c+1,d,aux);
7   intercala(v,e,c,d,aux);/* T=n */
8 }
```

Assumindo que $n = 2^k$

$$T(2^k) = 2T(2^{k-1}) + n$$

MERGESORT: COMPLEXIDADE DE TEMPO

```
1 void ordenaR(int v[], int e, int d,
2                       int aux[]) {
3   if (e >= d) return;
4   int c = (e + d)/2;
5   ordenaR(v,e,c,aux);
6   ordenaR(v,c+1,d,aux);
7   intercala(v,e,c,d,aux);/* T=n */
8 }
```

Assumindo que $n = 2^k$

$$T(2^k) = 2T(2^{k-1}) + n$$
$$= 2^2 T(2^{k-2}) + n + 2n/2$$

MERGESORT: COMPLEXIDADE DE TEMPO

```
1 void ordenaR(int v[], int e, int d,
2                       int aux[]) {
3   if (e >= d) return;
4   int c = (e + d)/2;
5   ordenaR(v,e,c,aux);
6   ordenaR(v,c+1,d,aux);
7   intercala(v,e,c,d,aux);/* T=n */
8 }
```

Assumindo que $n = 2^k$

$$
\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + n \\
&= 2^2 T(2^{k-2}) + n + 2n/2 \\
&= 2^3 T(2^{k-3}) + 3n
\end{aligned}
$$

MERGESORT: COMPLEXIDADE DE TEMPO

```
1 void ordenaR(int v[], int e, int d,
2                       int aux[]) {
3   if (e >= d) return;
4   int c = (e + d)/2;
5   ordenaR(v,e,c,aux);
6   ordenaR(v,c+1,d,aux);
7   intercala(v,e,c,d,aux);/* T=n */
8 }
```

Assumindo que $n = 2^k$

$$
\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + n \\
&= 2^2 T(2^{k-2}) + n + 2n/2 \\
&= 2^3 T(2^{k-3}) + 3n \\
&\vdots \\
&= 2^k T(1) + kn
\end{aligned}
$$

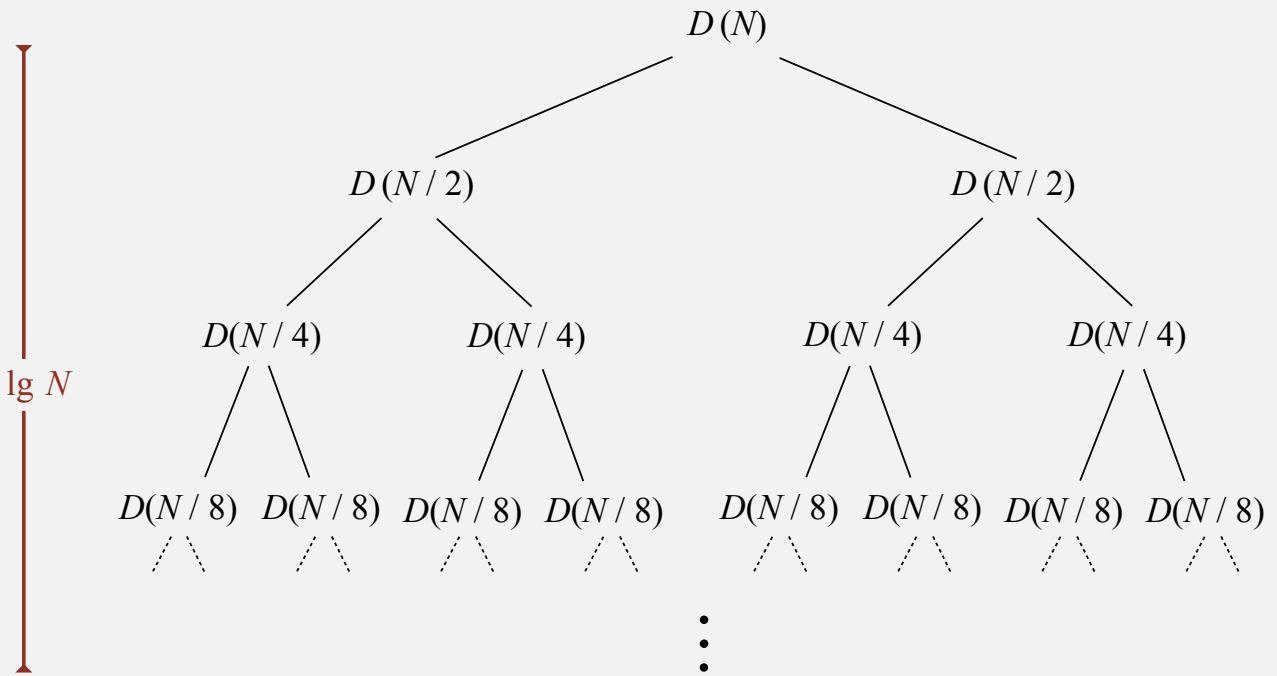
MERGESORT: COMPLEXIDADE DE TEMPO

```
1 void ordenaR(int v[], int e, int d,
2                        int aux[]) {
3   if (e >= d) return;
4   int c = (e + d)/2;
5   ordenaR(v,e,c,aux);
6   ordenaR(v,c+1,d,aux);
7   intercala(v,e,c,d,aux);/* T=n */
8 }
```

Assumindo que $n = 2^k$

$$
\begin{aligned}
T(2^k) &= 2T(2^{k-1}) + n \\
&= 2^2 T(2^{k-2}) + n + 2n/2 \\
&= 2^3 T(2^{k-3}) + 3n \\
&\vdots \\
&= 2^k T(1) + kn \in \boxed{O(n \log n)}
\end{aligned}
$$

MERGESORT: COMPLEXIDADE DE TEMPO



$$
\begin{aligned}
N &= N \\
2\,(N/2) &= N \\
4\,(N/4) &= N \\
8\,(N/8) &= N \\
&\vdots \\
\hline
T(N) &= N \lg N
\end{aligned}
$$

MERGESORT: COMPLEXIDADE DE TEMPO

```
1 void ordenaR(int v[], int e, int d, int aux[]) {
2   if (e >= d) return;
3   int c = (e + d)/2;
4   ordenaR(v,e,c,aux);
5   ordenaR(v,c+1,d,aux);
6   intercala(v,e,c,d,aux);
7 }
```

Prova por indução:

MERGESORT: COMPLEXIDADE DE TEMPO

```
1 void ordenaR(int v[], int e, int d, int aux[]) {
2   if (e >= d) return;
3   int c = (e + d)/2;
4   ordenaR(v,e,c,aux);
5   ordenaR(v,c+1,d,aux);
6   intercala(v,e,c,d,aux);
7 }
```

Prova por indução:

- Base: $T(2^0) = 1$ (constante)

MERGESORT: COMPLEXIDADE DE TEMPO

```
1 void ordenaR(int v[], int e, int d, int aux[]) {
2   if (e >= d) return;
3   int c = (e + d)/2;
4   ordenaR(v,e,c,aux);
5   ordenaR(v,c+1,d,aux);
6   intercala(v,e,c,d,aux);
7 }
```

Prova por indução:

- Base: $T(2^0) = 1$ (constante)
- Passo indutivo: Assuma que $T(2^k) = 2^k \log_2 2^k$. Então

MERGESORT: COMPLEXIDADE DE TEMPO

```
1 void ordenaR(int v[], int e, int d, int aux[]) {
2   if (e >= d) return;
3   int c = (e + d)/2;
4   ordenaR(v,e,c,aux);
5   ordenaR(v,c+1,d,aux);
6   intercala(v,e,c,d,aux);
7 }
```

Prova por indução:

- Base: $T(2^0) = 1$ (constante)
- Passo indutivo: Assuma que $T(2^k) = 2^k \log_2 2^k$. Então

$$T(2^{k+1}) = 2T(2^k) + 2 \cdot 2^k$$

MERGESORT: COMPLEXIDADE DE TEMPO

```
1 void ordenaR(int v[], int e, int d, int aux[]) {
2   if (e >= d) return;
3   int c = (e + d)/2;
4   ordenaR(v,e,c,aux);
5   ordenaR(v,c+1,d,aux);
6   intercala(v,e,c,d,aux);
7 }
```

Prova por indução:

- Base: $T(2^0) = 1$ (constante)
- Passo indutivo: Assuma que $T(2^k) = 2^k \log_2 2^k$. Então

$$\begin{aligned} T(2^{k+1}) &= 2T(2^k) + 2 \cdot 2^k \\ &= 2^{k+1}k + 2^{k+1} \end{aligned}$$

MERGESORT: COMPLEXIDADE DE TEMPO

```
1 void ordenaR(int v[], int e, int d, int aux[]) {
2   if (e >= d) return;
3   int c = (e + d)/2;
4   ordenaR(v,e,c,aux);
5   ordenaR(v,c+1,d,aux);
6   intercala(v,e,c,d,aux);
7 }
```

Prova por indução:

- Base: $T(2^0) = 1$ (constante)
- Passo indutivo: Assuma que $T(2^k) = 2^k \log_2 2^k$. Então

$$\begin{aligned} T(2^{k+1}) &= 2T(2^k) + 2 \cdot 2^k \\ &= 2^{k+1}k + 2^{k+1} \\ &= 2^{k+1}(k+1-1) + 2^{k+1} \end{aligned}$$

MERGESORT: COMPLEXIDADE DE TEMPO

```
1 void ordenaR(int v[], int e, int d, int aux[]) {
2   if (e >= d) return;
3   int c = (e + d)/2;
4   ordenaR(v,e,c,aux);
5   ordenaR(v,c+1,d,aux);
6   intercala(v,e,c,d,aux);
7 }
```

Prova por indução:

- Base: $T(2^0) = 1$ (constante)
- Passo indutivo: Assuma que $T(2^k) = 2^k \log_2 2^k$. Então

$$\begin{aligned} T(2^{k+1}) &= 2T(2^k) + 2 \cdot 2^k \\ &= 2^{k+1}k + 2^{k+1} \\ &= 2^{k+1}(k+1-1) + 2^{k+1} \\ &= 2^{k+1}(k+1) = \boxed{2n \log 2n} \end{aligned}$$ □

- Complexidade de espaço: $O(n)$
- Complexidade de tempo: $O(n \log n)$
- Melhorias:
  - Usar ordenação por inserção quando recursão for em instância pequena ($n \sim 10$), para evitar sobrecarga
  - Para recursão se `v[d] <= v[c+1]` (por que?)
- Vetor auxiliar pode ser descartado, porém algoritmo se torna excessivamente complexo (Kronrod, 1969)

- Complexidade de espaço: $O(n)$
- Complexidade de tempo: $O(n \log n)$
- Melhorias:
  - Usar ordenação por inserção quando recursão for em instância pequena ($n \sim 10$), para evitar sobrecarga
  - Para recursão se `v[d] <= v[c+1]` (por que?)
- Vetor auxiliar pode ser descartado, porém algoritmo se torna excessivamente complexo (Kronrod, 1969)

- Complexidade de espaço: $O(n)$
- Complexidade de tempo: $O(n \log n)$
- Melhorias:
  - Usar ordenação por inserção quando recursão for em instância pequena ($n \sim 10$), para evitar sobrecarga
  - Para recursão se `v[c] <= v[c+1]` (por que?)
- Vetor auxiliar pode ser descartado, porém algoritmo se torna excessivamente complexo (Kronrod, 1969)

- Complexidade de espaço: $O(n)$
- Complexidade de tempo: $O(n \log n)$
- Melhorias:
  - Usar ordenação por inserção quando recursão for em instância pequena ($n \sim 10$), para evitar sobrecarga
  - Para recursão se `v[c] <= v[c+1]` (por que?)
- Vetor auxiliar pode ser descartado, porém algoritmo se torna excessivamente complexo (Kronrod, 1969)

- Complexidade de espaço: $O(n)$
- Complexidade de tempo: $O(n \log n)$
- Melhorias:
  - Usar ordenação por inserção quando recursão for em instância pequena ($n \sim 10$), para evitar sobrecarga
  - Para recursão se `v[c] <= v[c+1]` (por que?)
- Vetor auxiliar pode ser descartado, porém algoritmo se torna excessivamente complexo (Kronrod, 1969)

- Complexidade de espaço: $O(n)$
- Complexidade de tempo: $O(n \log n)$
- Melhorias:
  - Usar ordenação por inserção quando recursão for em instância pequena ($n \sim 10$), para evitar sobrecarga
  - Para recursão se `v[c] <= v[c+1]` (por que?)
- Vetor auxiliar pode ser descartado, porém algoritmo se torna excessivamente complexo (Kronrod, 1969)

Estabilidade

```
1  void intercala(int v[], int e, int c, int d, int aux[]) {
2    for (int k = e; k <= d; k++) aux[k] = v[k];
3    int i = e, j = c + 1;
4    for (int k = e; k <= d; k++) {
5      if       (i > c)           v[k] = aux[j++];
6      else if (j > d)            v[k] = aux[i++];
7      else if (aux[j] < aux[i]) v[k] = aux[j++];
8      else                       v[k] = aux[i++];
9    }
10 }
```

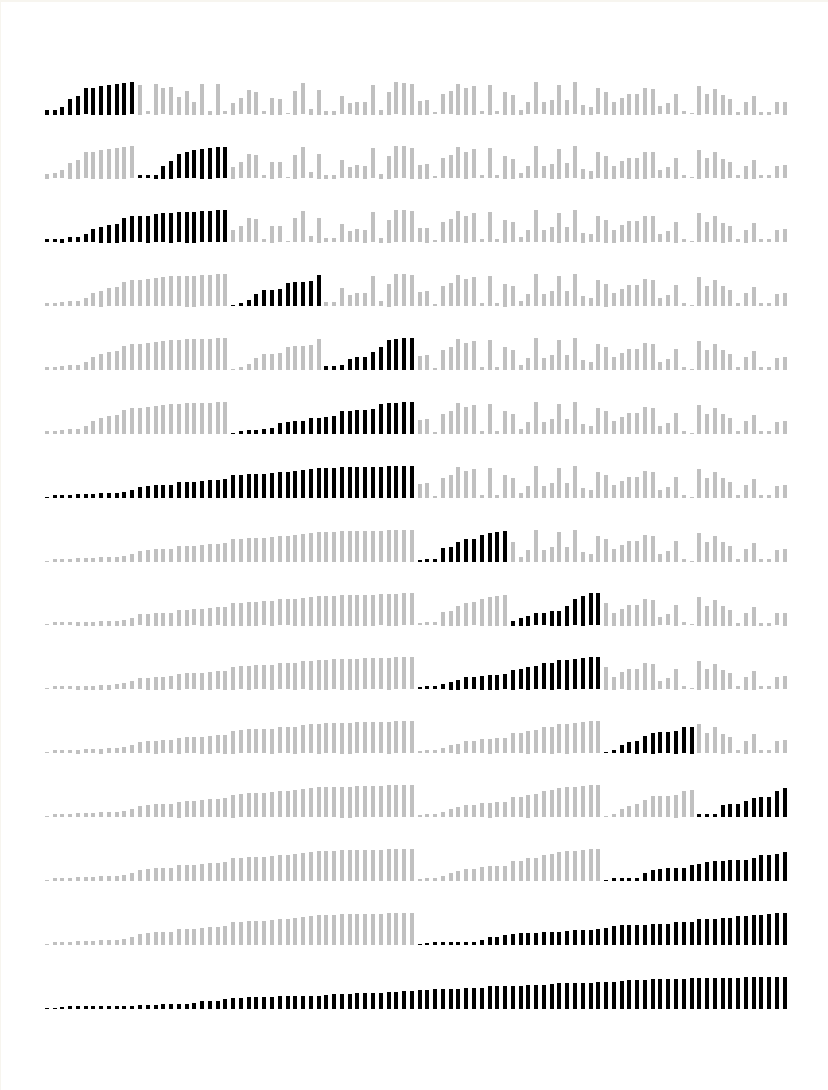
Função intercala não inverte posições relativas de números, portanto garante estabilidade.

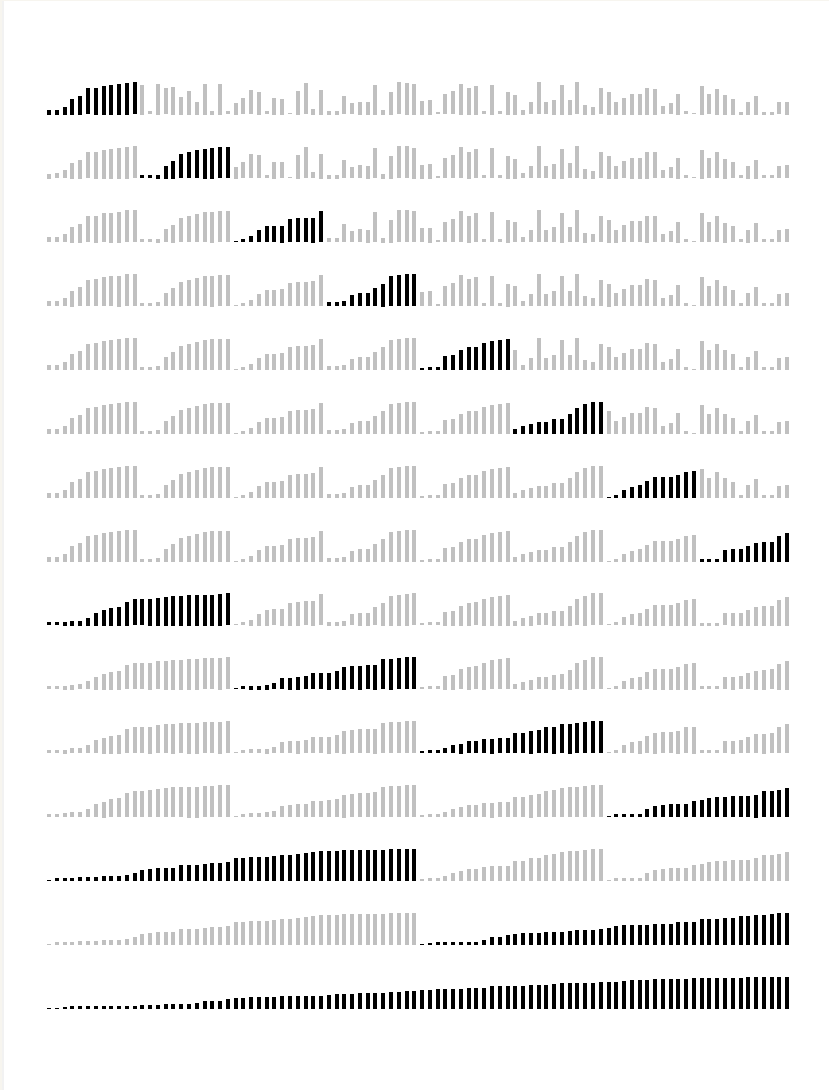
Ordenação por intercalação iterativo

```
1 void mergesort_it(int v[], int n) {
2   int *aux = malloc(n*sizeof(int));
3   for (int m = 1; m < n; m = 2*m)
4     for (int e = 0; e < n - m; e += 2*m)
5       intercala(v, e, e+m-1, min(e+2*m-1, n-1));
6 }
```

Empiricamente mais lento que versão recursiva

Ordenação por intercalação: Recursivo Vs. Iterativo





Para casa

Exercícios 12A e 12B