MAC122 - PRINCÍPIOS DE DESENVOLVIMENTO DE ALGORITMOS

Quicksort

```
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;
```

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 $x_1, \dots, x_n$ 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

$$\Downarrow$$

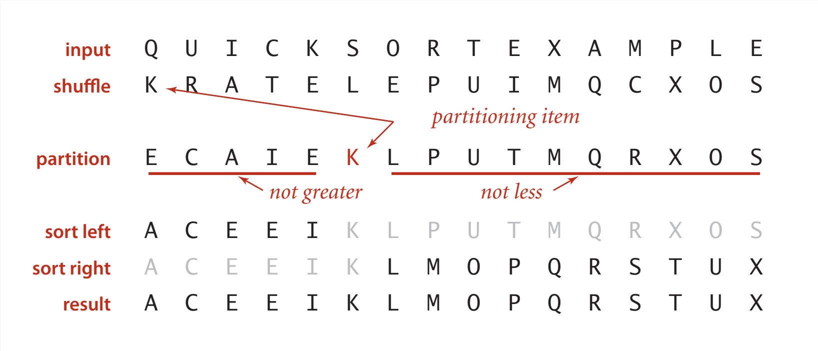
19	25	30	37	56	58	73	95

- 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: tempo $O(n^2)$ em qualquer caso, espaço $O(1)$
- Por inserção: tempo $O(n)$ no melhor caso, $O(n^2)$ em casos médio e pior caso, espaço $O(1)$
- Por intercalação: tempo $O(n \log n)$ em qualquer caso, espaço $O(n)$

Algoritmo “divide-e-conquista”:

1. Divida vetor em duas partes contíguas (de tamanhos potencialmente distintos) tais que `v[0..j-1] <= v[j] < v[j+1..n-1]` para algum elemento `v[j]`
2. Ordene recursivamente cada parte



Passo principal: Dado vetor `v[e..d]` rearranjar elementos para que `v[e..j-1] <= v[j] < v[j+1..d]` para pivô `v[j]` (em geral, primeiro elemento antes de rearranjo)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v[0..15]	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

$$\Downarrow$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v[0..15]	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

Algoritmo:

1. Defina `i=e+1` e `j=d` ◀
2. Repita até que `i > j`:
   2.1 Incremente `i` enquanto `v[i] <= v[e]`
   2.2 Decremente `j` enquanto `v[j] > v[e]`
   2.3 Troque `v[i]` e `v[j]`
3. Troque `v[e]` e `v[j]`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v[0..15]	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

PARTIÇÃO

Algoritmo:

1. Defina `i=e+1` e `j=d`
2. Repita até que `i > j`: ◀
   2.1 Incremente `i` enquanto `v[i] <= v[e]` ◀
   2.2 Decremente `j` enquanto `v[j] > v[e]` ◀
   2.3 Troque `v[i]` e `v[j]`
3. Troque `v[e]` e `v[j]`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v[0..15]	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

Algoritmo:

1. Defina `i=e+1` e `j=d`
2. Repita até que `i > j`: ◀
   2.1 Incremente `i` enquanto `v[i] <= v[e]`
   2.2 Decremente `j` enquanto `v[j] > v[e]`
   2.3 Troque `v[i]` e `v[j]` ◀
3. Troque `v[e]` e `v[j]`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v[0..15]	K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S

Algoritmo:

1. Defina `i=e+1` e `j=d`
2. Repita até que `i > j`: ◀
   2.1 Incremente `i` enquanto `v[i] <= v[e]` ◀
   2.2 Decremente `j` enquanto `v[j] > v[e]` ◀
   2.3 Troque `v[i]` e `v[j]`
3. Troque `v[e]` e `v[j]`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v[0..15]	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S

Algoritmo:

1. Defina `i=e+1` e `j=d`
2. Repita até que `i > j`: ◀
   - 2.1 Incremente `i` enquanto `v[i] <= v[e]`
   - 2.2 Decremente `j` enquanto `v[j] > v[e]`
   - 2.3 Troque `v[i]` e `v[j]` ◀
3. Troque `v[e]` e `v[j]`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v[0..15]	K	C	A	T	E	L	E	P	U	I	M	Q	R	X	O	S

Algoritmo:

1. Defina `i=e+1` e `j=d`
2. Repita até que `i > j`: ◀
   - 2.1 Incremente `i` enquanto `v[i] <= v[e]` ◀
   - 2.2 Decremente `j` enquanto `v[j] > v[e]` ◀
   - 2.3 Troque `v[i]` e `v[j]`
3. Troque `v[e]` e `v[j]`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v[0..15]	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S

Algoritmo:

1. Defina `i=e+1` e `j=d`
2. Repita até que `i > j`: ◀
   2.1 Incremente `i` enquanto `v[i] <= v[e]`
   2.2 Decremente `j` enquanto `v[j] > v[e]`
   2.3 Troque `v[i]` e `v[j]` ◀
3. Troque `v[e]` e `v[j]`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v[0..15]	K	C	A	I	E	L	E	P	U	T	M	Q	R	X	O	S

Algoritmo:

1. Defina `i=e+1` e `j=d`
2. Repita até que `i > j`: ◀
   2.1 Incremente `i` enquanto `v[i] <= v[e]`
   2.2 Decremente `j` enquanto `v[j] > v[e]`
   2.3 Troque `v[i]` e `v[j]`
3. Troque `v[e]` e `v[j]`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v[0..15]	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S

Algoritmo:

1. Defina `i=e+1` e `j=d`
2. Repita até que `i > j`:
   2.1 Incremente `i` enquanto `v[i] <= v[e]`
   2.2 Decremente `j` enquanto `v[j] > v[e]`
   2.3 Troque `v[i]` e `v[j]`
3. Troque `v[e]` e `v[j]` ◀

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v[0..15]	K	C	A	I	E	E	L	P	U	T	M	Q	R	X	O	S

Algoritmo:

1. Defina `i=e+1` e `j=d`
2. Repita até que `i > j`:
   - 2.1 Incremente `i` enquanto `v[i] <= v[e]`
   - 2.2 Decremente `j` enquanto `v[j] > v[e]`
   - 2.3 Troque `v[i]` e `v[j]`
3. Troque `v[e]` e `v[j]`

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
v[0..15]	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S

```
/* Rearranja v[e..d] tal que
  v[e+1..j] <= v[e] < v[j+1..d] e devolve j */
int separa(int v[], int e, int d) {
  int x;
  int i = e + 1; /* cursor da metade esquerda */
  int j = d;    /* cursor da metade direita */
  while (i <= j) {
    while (v[i] <= v[e]) i++;
    while (v[j] > v[e]) j--;
    /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
    if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x;    }
  }
  x = v[e]; v[e] = v[j]; v[j] = x;
  return j;
}
```

Partição

```
/* Rearranja v[e..d] tal que
  v[e+1..j] <= v[e] < v[j+1..d] e devolve j */
int separa(int v[], int e, int d) {
  int x;
  int i = e + 1; /* cursor da metade esquerda */
  int j = d;    /* cursor da metade direita */
  while (i <= j) { ▶ Invariante? ◀
    while (v[i] <= v[e]) i++;
    while (v[j] > v[e]) j--;
    /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
    if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x;    }
  }
  x = v[e]; v[e] = v[j]; v[j] = x;
  return j;
}
```

PARTIÇÃO

```
/* Rearranja v[e..d] tal que
  v[e+1..j] <= v[e] < v[j+1..d] e devolve j */
int separa(int v[], int e, int d) {
  int x;
  int i = e + 1; /* cursor da metade esquerda */
  int j = d;     /* cursor da metade direita */
  while (i <= j) {  /* v[e..i-1] <= v[e] < v[j+1..d] */
    while (v[i] <= v[e]) i++;
    while (v[j] > v[e]) j--;
    /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
    if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x;    }
  }
  x = v[e]; v[e] = v[j]; v[j] = x;
  return j;
}
```

v[e]	$\leq$ v[e]	v[i]		v[j]	$\geq$ v[e]

Partição: Correção

```
while (i <= j) { /* v[e..i-1] <= v[e] < v[j+1..d] */
  while (v[i] <= v[e]) i++;
  while (v[j] > v[e]) j--;
  /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
  if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x; }
}
```

Queremos provar que:

1. Programa termina
2. Invariante é verdadeira em cada iteração
3. Após última iteração, `v[e+1..j] <= v[e] < v[j+1..d]`

Partição: Correção

```
while (i <= j) { /* v[e..i-1] <= v[e] < v[j+1..d] */
  while (v[i] <= v[e]) i++;
  while (v[j] > v[e]) j--;
  /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
  if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x; }
}
```

Queremos provar que:

1. Programa termina: Cada iteração incrementa `i` e/ou decrementa `j`
2. Invariante é verdadeira em cada iteração
3. Após última iteração, `v[e+1..j] <= v[e] < v[j+1..d]`

Partição: Correção

```
while (i <= j) { /* v[e..i-1] <= v[e] < v[j+1..d] */
  while (v[i] <= v[e]) i++;
  while (v[j] > v[e]) j--;
  /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
  if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x; }
}
```

Queremos provar que:

1. Programa termina
2. Invariante é verdadeira em cada iteração: indução matemática
3. Após última iteração, `v[e+1..j] <= v[e] < v[j+1..d]`

Partição: Correção

```
while (i <= j) { /* v[e..i-1] <= v[e] < v[j+1..d] */
  while (v[i] <= v[e]) i++;
  while (v[j] > v[e]) j--;
  /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
  if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x; }
}
```

Base: Para `i=e` e `j=d`, resultado é trivialmente verdade.

v[e]	v[i]		v[j]

Partição: Correção

```
1   while (i <= j) { /* v[e..i-1] <= v[e] < v[j+1..d] */
2     while (v[i] <= v[e]) i++;
3     while (v[j] > v[e]) j--;
4     /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
5     if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x; }
6   }
7 }
```

Passo indutivo: Assuma que invariante é válida. Após executar linhas 2 e 3, ela permanece válida. A linha 5 não altera `i` e `j` nem o valor de `v[e..i-1]` ou de `v[j+1..d]`; portanto a invariante segue válida.

v[e]	$\leq$ v[e]	v[i]		v[j]	$\geq$ v[e]

Partição: Correção II

```c
while (i <= j) { /* v[e..i-1] <= v[e] < v[j+1..d] */
  while (v[i] <= v[e]) i++;
  while (v[j] > v[e]) j--;
  /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
  if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x; }
}
```

Queremos provar que:

1. Programa termina
2. Invariante é verdadeira em cada iteração
3. Após última iteração, `v[e+1..j] <= v[e] < v[j+1..d]`

```
1   while (i <= j) { /* v[e+1..i-1] <= v[e] <= v[j+1..d] */
2     while (v[i] <= v[e]) i++;
3     while (v[j] > v[e]) j--;
4     /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
5     if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x; }
6   }
7 }
```

Término: Vamos assumir que vetor `v[e..d]` contém pelo menos 2 valores distintos. Na última iteração, após executar as linhas 2 e 3, temos que `v[e+1..i-1] <= v[e] < v[j+1..d]` e `i = j + 1` $\Rightarrow$ `v[j] <= v[e]`. Assim, `v[e..j] <= v[e] < v[j+1..d]`.

v[e]	$\leq$ v[e]	v[j]	v[i]	$\geq$ v[e]

```
/* Rearranja v[e..d] tal que
  v[e+1..j] <= v[e] < v[j+1..d] e devolve j */
int separa(int v[], int e, int d) {
  int x;
  int i = e + 1; /* cursor da metade esquerda */
  int j = d;    /* cursor da metade direita */
  while (i <= j) {
   while (v[i] <= v[e]) i++;
    while (v[j] > v[e]) j--;
    /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
    if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x;    }
  }
  x = v[e]; v[e] = v[j]; v[j] = x;
  return j;
}
```

Complexidade de espaço:

```c
/* Rearranja v[e..d] tal que
  v[e+1..j] <= v[e] < v[j+1..d] e devolve j */
int separa(int v[], int e, int d) {
  int x;
  int i = e + 1; /* cursor da metade esquerda */
  int j = d;    /* cursor da metade direita */
  while (i <= j) {
   while (v[i] <= v[e]) i++;
    while (v[j] > v[e]) j--;
    /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
    if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x;     }
  }
  x = v[e]; v[e] = v[j]; v[j] = x;
  return j;
}
```

Complexidade de espaço: $O(1)$

```
/* Rearranja v[e..d] tal que
  v[e+1..j] <= v[e] <= v[j+1..d] e devolve j */
int separa(int v[], int e, int d) {
  int x;
  int i = e + 1; /* cursor da metade esquerda */
  int j = d;    /* cursor da metade direita */
  while (i <= j) {
   while (v[i] <= v[e]) i++;
    while (v[j] > v[e]) j--;
    /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
    if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x;     }
  }
  x = v[e]; v[e] = v[j]; v[j] = x;
  return j;
}
```

Complexidade de tempo:

Partição: Eficiência

```
/* Rearranja v[e..d] tal que
  v[e+1..j] <= v[e] <= v[j+1..d] e devolve j */
int separa(int v[], int e, int d) {
  int x;
  int i = e + 1; /* cursor da metade esquerda */
  int j = d;    /* cursor da metade direita */
  while (i <= j) {
   while (v[i] <= v[e]) i++;
    while (v[j] > v[e]) j--;
    /* i >= j || (v[i] > v[e] && v[j] <= v[e]) */
    if (i < j) {    x = v[i]; v[i] = v[j]; v[j] = x;    }
  }
  x = v[e]; v[e] = v[j]; v[j] = x;
  return j;
}
```

Complexidade de tempo: $O(n)$

QUICKSORT

```
void quicksort(int v[], int e, int d) {
  if (e >= d) return; /* ordenado */
  int j = separa(v, e, d);
  /* v[j] está em posição final */
  quicksort(v, e, j-1);
  quicksort(v, j+1, d);
}

void ordena(int v[], int n) {
  embaralha(v, n); /* importante para eficiência */
  quicksort(v, 0, n-1);
}
```

EMBARALHAR[1]

```c
/* Permuta aleatoriamente os elementos de v,
 pelo algoritmo de Fisher-Yates */
void embaralha(int v[], int n) {
 int x;
 for (int i = 0; i < n - 1; i++) {
   j = i + rand() % (n-i);
   x = v[i]; v[i] = v[j]; v[j] = x;
 }
}
```

Complexidade de tempo: $O(n)$

[1] O algoritmo apresenta um pequeno viés devido aos restos da divisão por `n-i` não serem uniformemente distribuídos

Simulação

	lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
initial values				Q	U	I	C	K	S	O	R	T	E	X	A	M	P	L	E
random shuffle				K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
	0	5	15	E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
	0	3	4	E	C	A	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	2	2	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	0	0	1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	1		1	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	4		4	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
	6	6	15	A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
no partition for subarrays of size 1	7	9	15	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	7	7	8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	8		8	A	C	E	E	I	K	L	M	O	P	T	Q	R	X	U	S
	10	13	15	A	C	E	E	I	K	L	M	O	P	S	Q	R	T	U	X
	10	12	12	A	C	E	E	I	K	L	M	O	P	R	Q	S	T	U	X
	10	11	11	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	10		10	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	14	14	15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
	15		15	A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
result				A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X

Quicksort trace (array contents after each partition)

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

QUICKSORT

```c
void quicksort(int v[], int e, int d) {
  if (e >= d) return; /* ordenado */
  int j = separa(v, e, d);
  quicksort(v, e, j-1);
  quicksort(v, j+1, d);
}

void ordena(int v[], int n) {
  embaralha(v, n); /* importante para eficiência */
  quicksort(v, 0, n-1);
}
```

Complexidade de tempo: $T(n) = T(j) + T(n-j) + n$

```
void quicksort(int v[], int e, int d) {
  if (e >= d) return; /* ordenado */
  int j = separa(v, e, d);
  quicksort(v, e, j-1);
  quicksort(v, j+1, d);
}

void ordena(int v[], int n) {
  embaralha(v, n); /* importante para eficiência */
  quicksort(v, 0, n-1);
}
```

Complexidade de tempo: $T(n) = T(j) + T(n-j) + n$

$\Rightarrow$ Depende de balanceamento do pivô j

Pivô é mediana

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
random shuffle			H	A	C	B	F	E	G	D	L	I	K	J	N	M	O
0	7	14	D	A	C	B	F	E	G	H	L	I	K	J	N	M	O
0	3	6	B	A	C	D	F	E	G	H	L	I	K	J	N	M	O
0	1	2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
0		0	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
2		2	A	B	C	D	F	E	G	H	L	I	K	J	N	M	O
4	5	6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
4		4	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
6		6	A	B	C	D	E	F	G	H	L	I	K	J	N	M	O
8	11	14	A	B	C	D	E	F	G	H	J	I	K	L	N	M	O
8	9	10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
8		8	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
10		10	A	B	C	D	E	F	G	H	I	J	K	L	N	M	O
12	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12		12	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort: Complexidade de melhor caso

```c
void quicksort(int v[], int e, int d) {
  if (e >= d) return; /* ordenado */
  int j = separa(v, e, d);
  quicksort(v, e, j-1);
  quicksort(v, j+1, d);
}

void ordena(int v[], int n) {
  embaralha(v, n); /* importante para eficiência */
  quicksort(v, 0, n-1);
}
```

Complexidade de tempo: $T(n) = 2T(n/2) + n$

QUICKSORT: COMPLEXIDADE DE MELHOR CASO

```c
void quicksort(int v[], int e, int d) {
  if (e >= d) return; /* ordenado */
  int j = separa(v, e, d);
  quicksort(v, e, j-1);
  quicksort(v, j+1, d);
}

void ordena(int v[], int n) {
  embaralha(v, n); /* importante para eficiência */
  quicksort(v, 0, n-1);
}
```

Complexidade de tempo: $T(n) = 2T(n/2) + n$
$\Rightarrow T(n) \in O(n \log n)$

Pivô é mínimo/máximo

lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
initial values			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
random shuffle			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0	0	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	1	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
2	2	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	3	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
4	4	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	5	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
6	6	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
7	7	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
8	8	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9	9	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
10	10	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
11	11	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
12	12	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
13	13	14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
14		14	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
			A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Quicksort: Complexidade de pior caso

```c
void quicksort(int v[], int e, int d) {
  if (e >= d) return; /* ordenado */
  int j = separa(v, e, d);
  quicksort(v, e, j-1);
  quicksort(v, j+1, d);
}

void ordena(int v[], int n) {
  embaralha(v, n); /* importante para eficiência */
  quicksort(v, 0, n-1);
}
```

Complexidade de tempo: $T(n) = 2T(n-1) + T(1) + n$

```c
void quicksort(int v[], int e, int d) {
  if (e >= d) return; /* ordenado */
  int j = separa(v, e, d);
  quicksort(v, e, j-1);
  quicksort(v, j+1, d);
}

void ordena(int v[], int n) {
  embaralha(v, n); /* importante para eficiência */
  quicksort(v, 0, n-1);
}
```

Complexidade de tempo: $T(n) = 2T(n-1) + T(1) + n$
$\Rightarrow T(n) \in O(n^2)$

Assumindo que partições são equiprováveis,

$$
\begin{aligned}
T(n) &= (n+1) + \left(\frac{T(1) + T(n-1)}{n}\right) + \cdots + \left(\frac{T(n-1) + T(1)}{n}\right) \\
&= (n+1) + \frac{2}{n}\left(T(1) + T(2) + \cdots + T(n-1)\right) .
\end{aligned}
$$

Assumindo que partições são equiprováveis,

$$
\begin{aligned}
T(n) &= (n+1) + \left(\frac{T(1)+T(n-1)}{n}\right) + \cdots + \left(\frac{T(n-1)+T(1)}{n}\right) \\
&= (n+1) + \frac{2}{n}\left(T(1)+T(2)+\cdots+T(n-1)\right) .
\end{aligned}
$$

Multiplicando ambos os lados por n e subtraindo da mesma equação para $n-1$:

$$
nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n \, .
$$

Assumindo que partições são equiprováveis,

$$
\begin{aligned}
T(n) &= (n+1) + \left(\frac{T(1)+T(n-1)}{n}\right) + \cdots + \left(\frac{T(n-1)+T(1)}{n}\right) \\
&= (n+1) + \frac{2}{n}\left(T(1)+T(2)+\cdots+T(n-1)\right) .
\end{aligned}
$$

Multiplicando ambos os lados por n e subtraindo da mesma equação para $n-1$:

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n\,.$$

Rearranjando os termos e dividindo por $n(n+1)$:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}$$

$$
\begin{aligned}
\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2}{n+1} \\
&= \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\vdots \\
&= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \cdots + \frac{2}{n+1}
\end{aligned}
$$

$$
\begin{aligned}
\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2}{n+1} \\
&= \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\vdots \\
&= \frac{2}{3} + \frac{2}{4} + \frac{2}{5} + \cdots + \frac{2}{n+1}
\end{aligned}
$$

Portanto:

$$
\begin{aligned}
T(n) &= 2(n+1)\left(\frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{n+1}\right) \quad < 2(n+1)\int_1^{n+1} \frac{1}{x}dx \\
&= 2(n+1)\ln n \in O(n \log n)
\end{aligned}
$$

Partição faz inversões de valores duplicados ⇒ algoritmo não é estável.

0	1	2	3
B	C	C	A

0	1	2	3
B	A	C	C

0	1	2	3
A	B	C	C

Partição faz inversões de valores duplicados $\Rightarrow$ algoritmo não é estável.

0	1	2	3
B	C	C	A

0	1	2	3
B	A	C	C

0	1	2	3
A	B	C	C

É possível tornar algoritmo estável usando um vetor auxiliar para fazer partição (mas isso o torna menos eficiente que o MergeSort)

- **Complexidade de espaço**: Profundidade da recursão, $O(\log n)$ a $O(n)$
- **Complexidade de tempo**: $O(n \log n)$ a $O(n^2)$
- **Melhorias**:
  - Recorrer primeiro no vetor menor (faz com que segunda recursão seja de cauda, garantindo complexidade de espaço logarítimica)
  - Parar recursão quando instância é pequena (e usar ordenação por inserção)
  - Escolher com pivô a mediana de 3 elementos quaisquer (ex.: `v[e], v[d], v[(e+d)/2]`)
  - Particionar vetor em `v[e..j-1] < v[j..k] < v[k+1..d]`, reduz tempo em muitas instâncias

Quicksort: Resumo

- Complexidade de espaço: Profundidade da recursão, $O(\log n)$ a $O(n)$
- Complexidade de tempo: $O(n \log n)$ a $O(n^2)$
- Melhorias:
  - Recorrer primeiro no vetor menor (faz com que segunda recursão seja de cauda, garantindo complexidade de espaço logarítimica)
  - Parar recursão quando instância é pequena (e usar ordenação por inserção)
  - Escolher com pivô a mediana de 3 elementos quaisquer (ex.: `v[e], v[d], v[(e+d)/2]`)
  - Particionar vetor em `v[e..j-1] < v[j..k] < v[k+1..d]`, reduz tempo em muitas instâncias

- Complexidade de espaço: Profundidade da recursão, $O(\log n)$ a $O(n)$
- Complexidade de tempo: $O(n \log n)$ a $O(n^2)$
- Melhorias:
  - Recorrer primeiro no vetor menor (faz com que segunda recursão seja de cauda, garantindo complexidade de espaço logarítimica)
  - Parar recursão quando instância é pequena (e usar ordenação por inserção)
  - Escolher com pivô a mediana de 3 elementos quaisquer (ex.: `v[e], v[d], v[(e+d)/2]`)
  - Particionar vetor em `v[e..j-1] < v[j..k] < v[k+1..d]`, reduz tempo em muitas instâncias

- Complexidade de espaço: Profundidade da recursão, $O(\log n)$ a $O(n)$
- Complexidade de tempo: $O(n \log n)$ a $O(n^2)$
- Melhorias:
  - Recorrer primeiro no vetor menor (faz com que segunda recursão seja de cauda, garantindo complexidade de espaço logarítimica)
  - Parar recursão quando instância é pequena (e usar ordenação por inserção)
  - Escolher com pivô a mediana de 3 elementos quaisquer (ex.: `v[e], v[d], v[(e+d)/2]`)
  - Particionar vetor em `v[e..j-1] < v[j..k] < v[k+1..d]`, reduz tempo em muitas instâncias

- Complexidade de espaço: Profundidade da recursão, $O(\log n)$ a $O(n)$
- Complexidade de tempo: $O(n \log n)$ a $O(n^2)$
- Melhorias:
  - Recorrer primeiro no vetor menor (faz com que segunda recursão seja de cauda, garantindo complexidade de espaço logarítimica)
  - Parar recursão quando instância é pequena (e usar ordenação por inserção)
  - Escolher com pivô a mediana de 3 elementos quaisquer (ex.: `v[e], v[d], v[(e+d)/2]`)
  - Particionar vetor em `v[e..j-1] < v[j..k] < v[k+1..d]`, reduz tempo em muitas instâncias

- Complexidade de espaço: Profundidade da recursão, $O(\log n)$ a $O(n)$
- Complexidade de tempo: $O(n \log n)$ a $O(n^2)$
- Melhorias:
  - Recorrer primeiro no vetor menor (faz com que segunda recursão seja de cauda, garantindo complexidade de espaço logarítimica)
  - Parar recursão quando instância é pequena (e usar ordenação por inserção)
  - Escolher com pivô a mediana de 3 elementos quaisquer (ex.: `v[e], v[d], v[(e+d)/2]`)
  - Particionar vetor em `v[e..j-1] < v[j..k] < v[k+1..d]`, reduz tempo em muitas instâncias

- Complexidade de espaço: Profundidade da recursão, $O(\log n)$ a $O(n)$
- Complexidade de tempo: $O(n \log n)$ a $O(n^2)$
- Melhorias:
  - Recorrer primeiro no vetor menor (faz com que segunda recursão seja de cauda, garantindo complexidade de espaço logarítimica)
  - Parar recursão quando instância é pequena (e usar ordenação por inserção)
  - Escolher com pivô a mediana de 3 elementos quaisquer (ex.: `v[e], v[d], v[(e+d)/2]`)
  - Particionar vetor em `v[e..j-1] < v[j..k] < v[k+1..d]`, reduz tempo em muitas instâncias

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	$\frac{1}{2} N^2$	N exchanges
insertion	✔	✔	N	$\frac{1}{4} N^2$	$\frac{1}{2} N^2$	use for small N or partially ordered
shell	✔		$N \log_3 N$	?	$c N^{3/2}$	tight code; subquadratic
merge		✔	$\frac{1}{2} N \lg N$	$N \lg N$	$N \lg N$	$N \log N$ guarantee; stable
timsort		✔	N	$N \lg N$	$N \lg N$	improves mergesort when preexisting order
quick	✔		$N \lg N$	$2 N \ln N$	$\frac{1}{2} N^2$	$N \log N$ probabilistic guarantee; fastest in practice
3-way quick	✔		N	$2 N \ln N$	$\frac{1}{2} N^2$	improves quicksort when duplicate keys
?	✔	✔	N	$N \lg N$	$N \lg N$	holy sorting grail