Ordenação com Quicksort

Projeto e Análise de Algoritmos

Bruno Prado

Departamento de Computação / UFS

Introdução

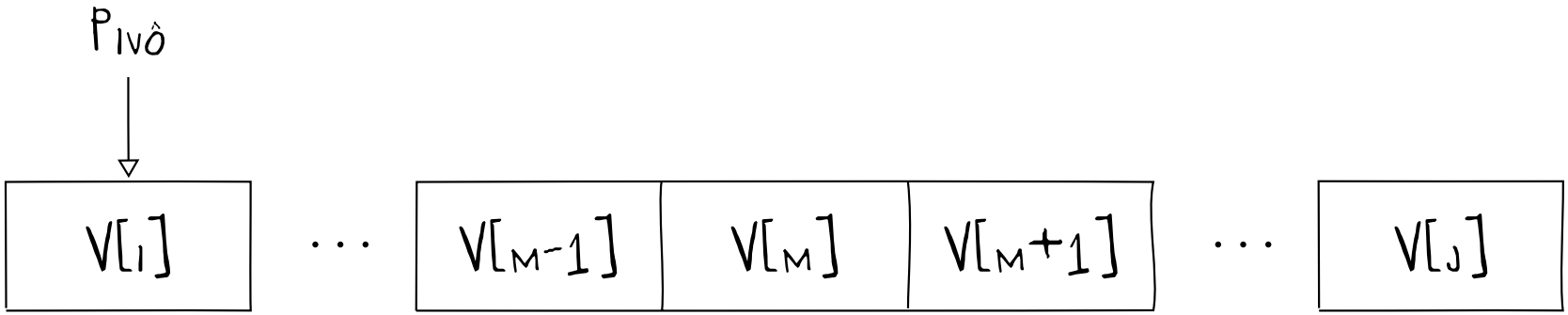
- O que é o Quicksort?
  - É um algoritmo de ordenação instável, criado em 1960 pelo cientista da computação Tony Hoare

Introdução

- O que é o Quicksort?
  - É um algoritmo de ordenação instável, criado em 1960 pelo cientista da computação Tony Hoare
  - Utiliza a estratégia de Divisão e Conquista
    1. Etapa de divisão do problema: criar subproblemas menores
    2. Resolver os subproblemas: gerar soluções mais simples
    3. Etapa de conquista: combinar os resultados parciais

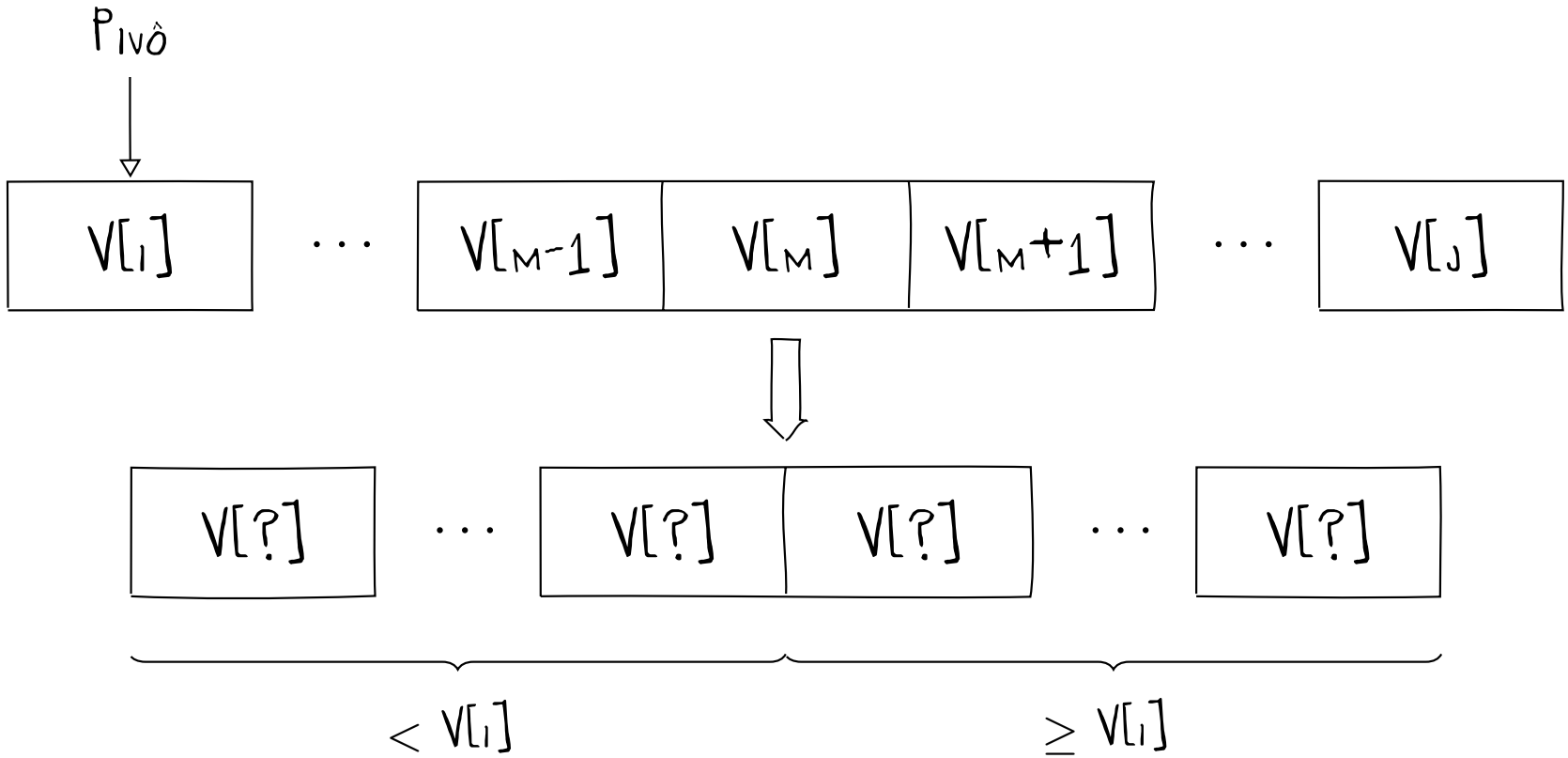
Quicksort

- Etapa de divisão
  - Particionamento do vetor por um pivô



Quicksort

- Etapa de divisão
  - Particionamento do vetor por um pivô



Quicksort

- Etapa de divisão

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
15  // Quicksort recursivo
16  void quicksort(int32_t* V, int32_t i, int32_t j) {
17      // Caso base
18      if(i < j) {
19          // Particionamento do vetor
20          int32_t p = hoare(V, i, j);
21          // Divisão em subvetores
22          quicksort(V, i, p);
23          quicksort(V, p + 1, j);
24      }
25  }
```

Quicksort

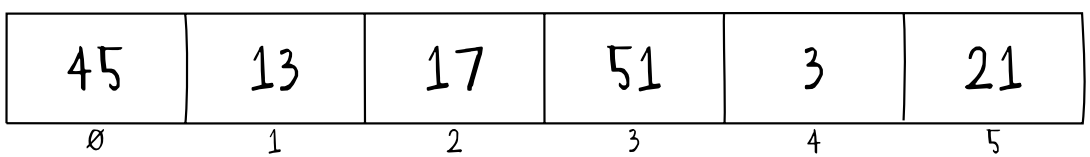
- Etapa de divisão

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
15  // Quicksort recursivo
16  void quicksort(int32_t* V, int32_t i, int32_t j) {
17      // Caso base
18      if(i < j) {
19          // Particionamento do vetor
20          int32_t p = hoare(V, i, j);
21          // Divisão em subvetores
22          quicksort(V, i, p);
23          quicksort(V, p + 1, j);
24      }
25  }
```

Quicksort

- Etapa de conquista

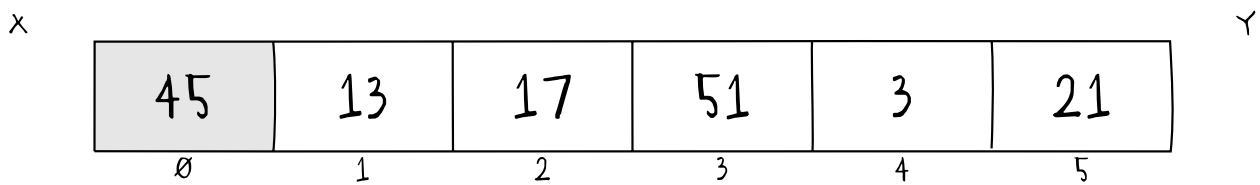
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare)
4   int32_t hoare(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[i], x = i - 1, y = j + 1;
7       // Particionando o vetor pelo pivô
8       while(1) {
9           while(V[--y] > P);
10          while(V[++x] < P);
11          if(x < y) trocar(&V[x], &V[y]);
12          else return y;
13      }
14  }
…   ...
```



Quicksort

- Etapa de conquista

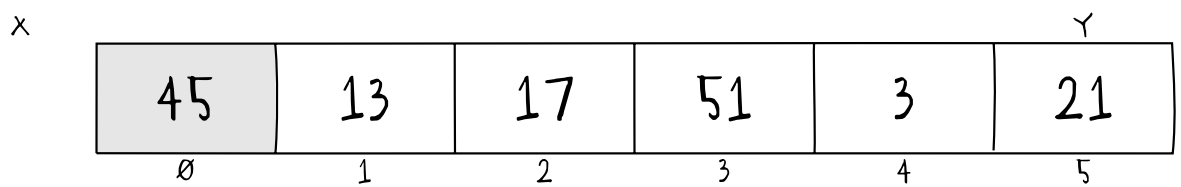
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare)
4   int32_t hoare(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[i], x = i - 1, y = j + 1;
7       // Particionando o vetor pelo pivô
8       while(1) {
9           while(V[--y] > P);
10          while(V[++x] < P);
11          if(x < y) trocar(&V[x], &V[y]);
12          else return y;
13      }
14  }
... ...
```



Quicksort

- Etapa de conquista

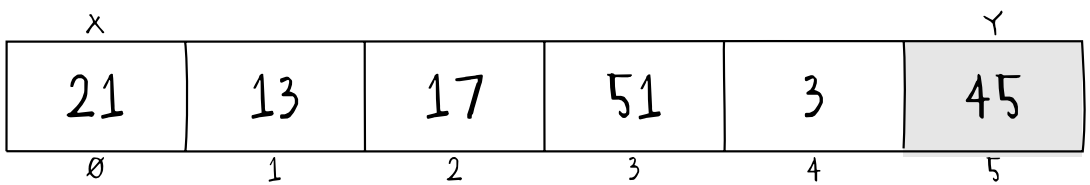
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare)
4   int32_t hoare(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[i], x = i - 1, y = j + 1;
7       // Particionando o vetor pelo pivô
8       while(1) {
9           while(V[--y] > P);
10          while(V[++x] < P);
11          if(x < y) trocar(&V[x], &V[y]);
12          else return y;
13      }
14  }
…   ...
```



Quicksort

- Etapa de conquista

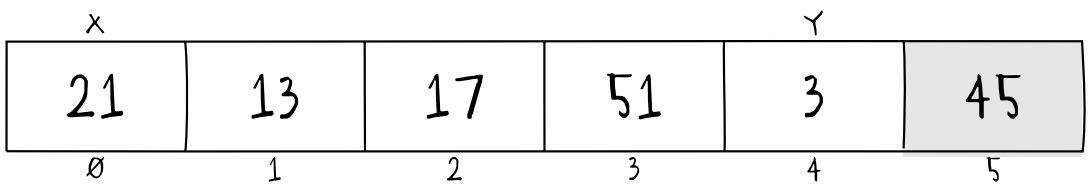
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare)
4   int32_t hoare(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[i], x = i - 1, y = j + 1;
7       // Particionando o vetor pelo pivô
8       while(1) {
9           while(V[--y] > P);
10          while(V[++x] < P);
11          if(x < y) trocar(&V[x], &V[y]);
12          else return y;
13      }
14  }
…   ...
```



Quicksort

- Etapa de conquista

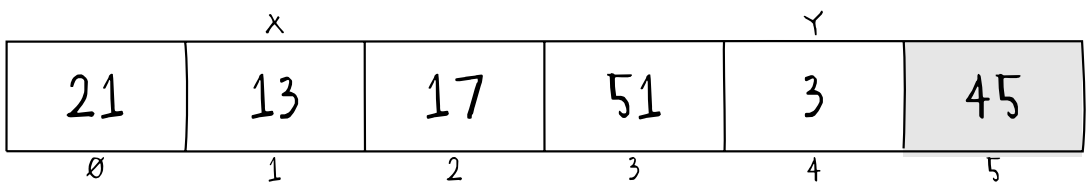
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare)
4   int32_t hoare(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[i], x = i - 1, y = j + 1;
7       // Particionando o vetor pelo pivô
8       while(1) {
9           while(V[--y] > P);
10          while(V[++x] < P);
11          if(x < y) trocar(&V[x], &V[y]);
12          else return y;
13      }
14  }
…   ...
```



Quicksort

- Etapa de conquista

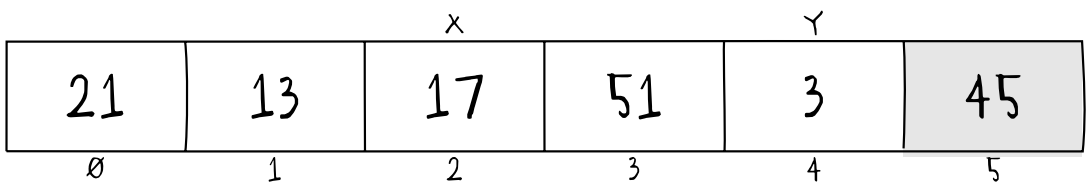
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare)
4   int32_t hoare(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[i], x = i - 1, y = j + 1;
7       // Particionando o vetor pelo pivô
8       while(1) {
9           while(V[--y] > P);
10          while(V[++x] < P);
11          if(x < y) trocar(&V[x], &V[y]);
12          else return y;
13      }
14  }
…   ...
```



Quicksort

- Etapa de conquista

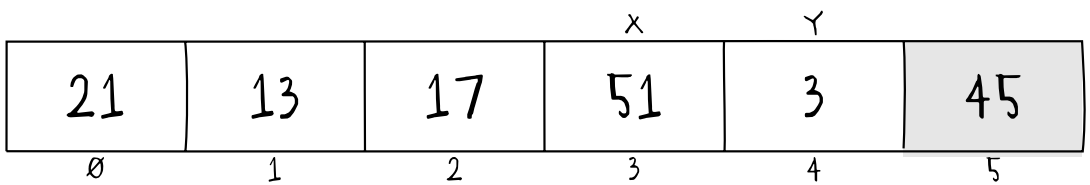
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare)
4   int32_t hoare(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[i], x = i - 1, y = j + 1;
7       // Particionando o vetor pelo pivô
8       while(1) {
9           while(V[--y] > P);
10          while(V[++x] < P);
11          if(x < y) trocar(&V[x], &V[y]);
12          else return y;
13      }
14  }
…   ...
```



Quicksort

- Etapa de conquista

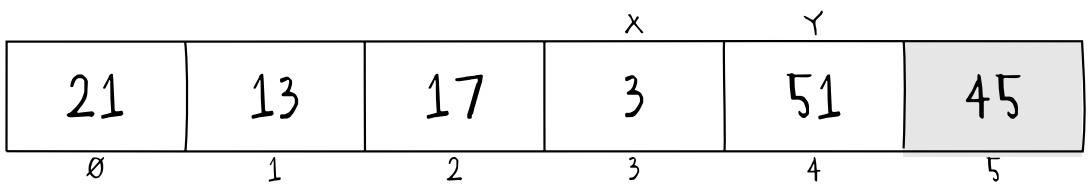
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare)
4   int32_t hoare(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[i], x = i - 1, y = j + 1;
7       // Particionando o vetor pelo pivô
8       while(1) {
9           while(V[--y] > P);
10          while(V[++x] < P);
11          if(x < y) trocar(&V[x], &V[y]);
12          else return y;
13      }
14  }
…   ...
```



Quicksort

- Etapa de conquista

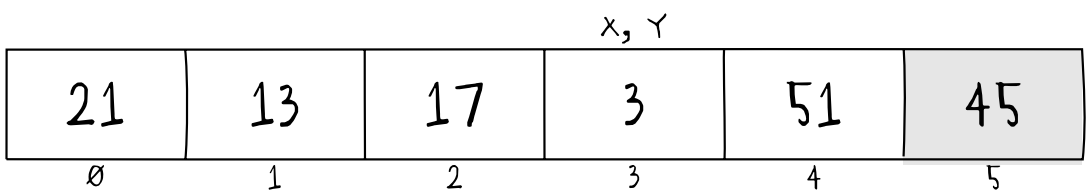
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare)
4   int32_t hoare(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[i], x = i - 1, y = j + 1;
7       // Particionando o vetor pelo pivô
8       while(1) {
9           while(V[--y] > P);
10          while(V[++x] < P);
11          if(x < y) trocar(&V[x], &V[y]);
12          else return y;
13      }
14  }
…   ...
```



Quicksort

- Etapa de conquista

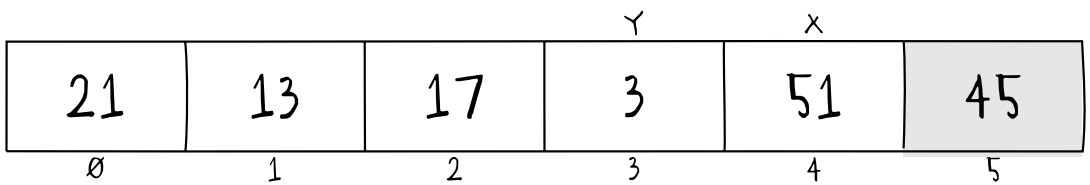
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare)
4   int32_t hoare(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[i], x = i - 1, y = j + 1;
7       // Particionando o vetor pelo pivô
8       while(1) {
9           while(V[--y] > P);
10          while(V[++x] < P);
11          if(x < y) trocar(&V[x], &V[y]);
12          else return y;
13      }
14  }
…   ...
```



Quicksort

- Etapa de conquista

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare)
4   int32_t hoare(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[i], x = i - 1, y = j + 1;
7       // Particionando o vetor pelo pivô
8       while(1) {
9           while(V[--y] > P);
10          while(V[++x] < P);
11          if(x < y) trocar(&V[x], &V[y]);
12          else return y;
13      }
14  }
…   ...
```



Quicksort

- Etapa de divisão

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
…  ...
15 // Quicksort recursivo
16 void quicksort(int32_t* V, int32_t i, int32_t j) {
17     // Caso base
18     if(i < j) {
19         // Particionamento do vetor
20         int32_t p = hoare(V, i, j);
21         // Divisão em subvetores
22         quicksort(V, i, p);
23         quicksort(V, p + 1, j);
24     }
25 }
```

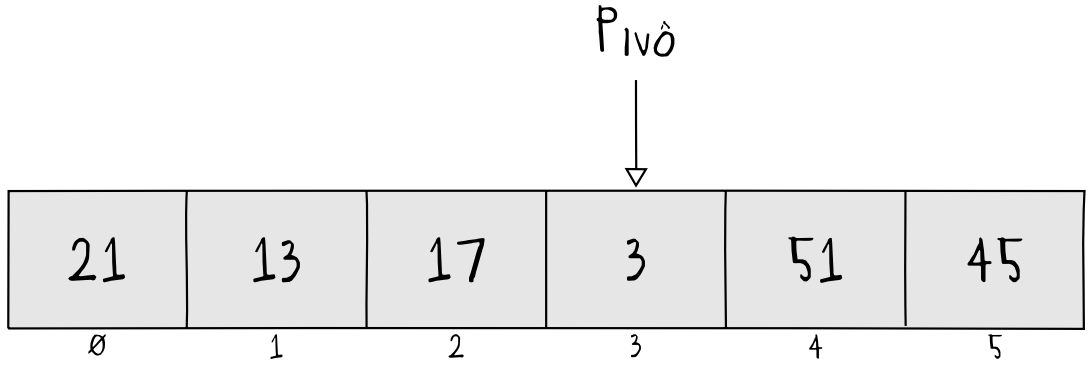
Quicksort

- Etapa de divisão

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
…  ...
15 // Quicksort recursivo
16 void quicksort(int32_t* V, int32_t i, int32_t j) {
17     // Caso base
18     if(i < j) {
19         // Particionamento do vetor
20         int32_t p = hoare(V, i, j);
21         // Divisão em subvetores
22         quicksort(V, i, p);
23         quicksort(V, p + 1, j);
24     }
25 }
```

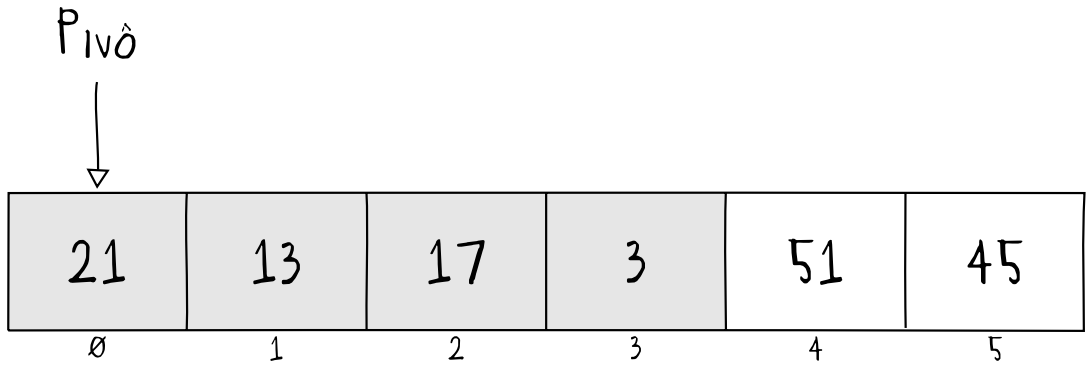
Quicksort

- Etapa de divisão



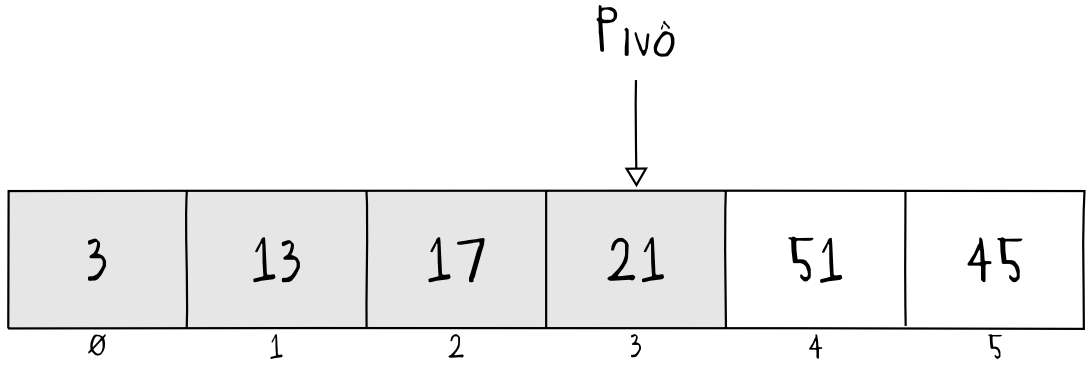
Quicksort

- Etapa de divisão



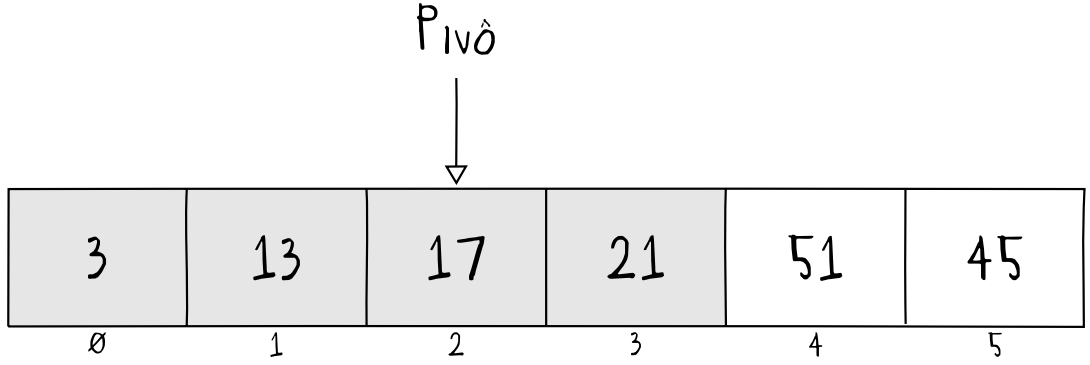
Quicksort

- Etapa de divisão



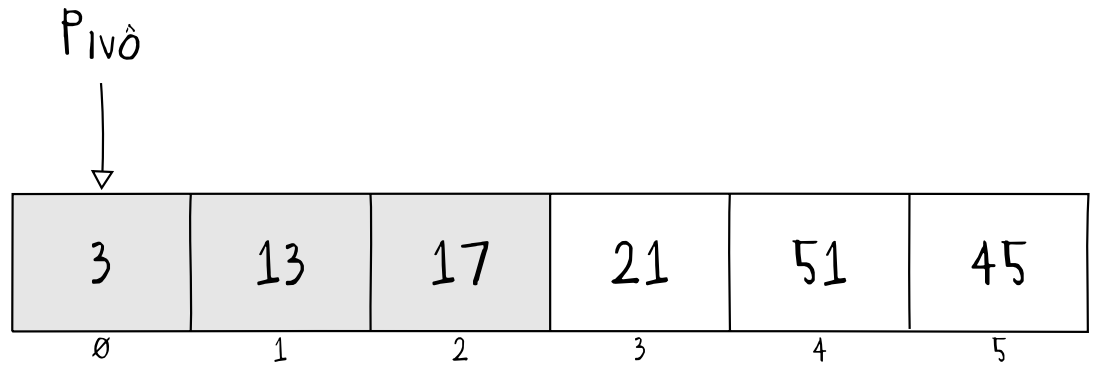
Quicksort

- Etapa de divisão



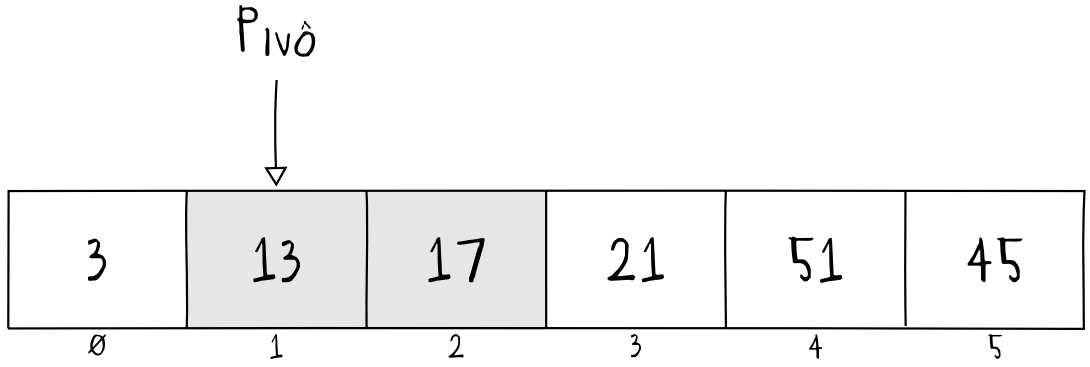
Quicksort

- Etapa de divisão



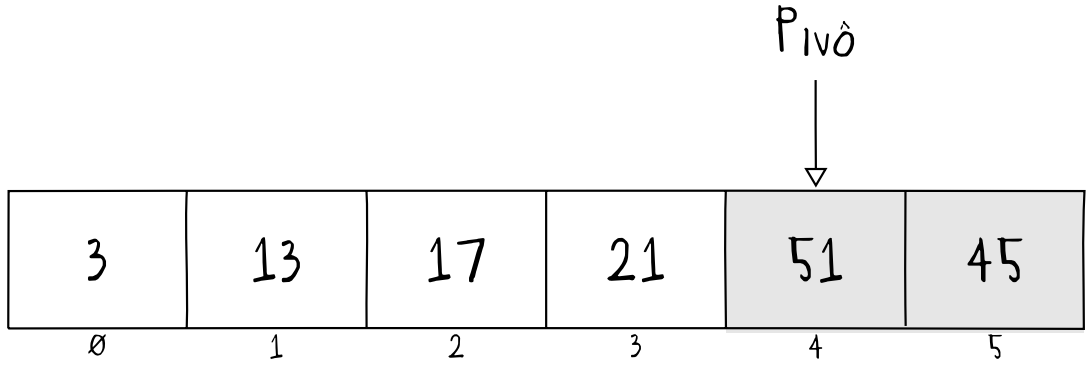
Quicksort

- Etapa de divisão



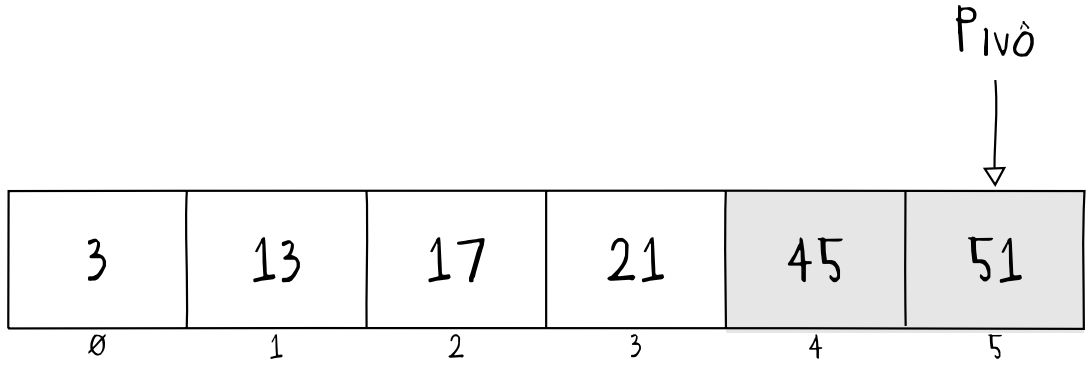
Quicksort

- Etapa de divisão



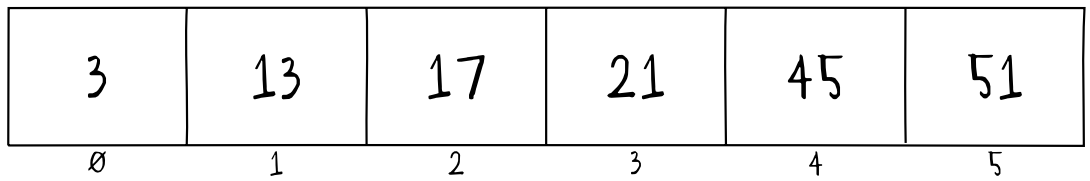
Quicksort

- Etapa de divisão



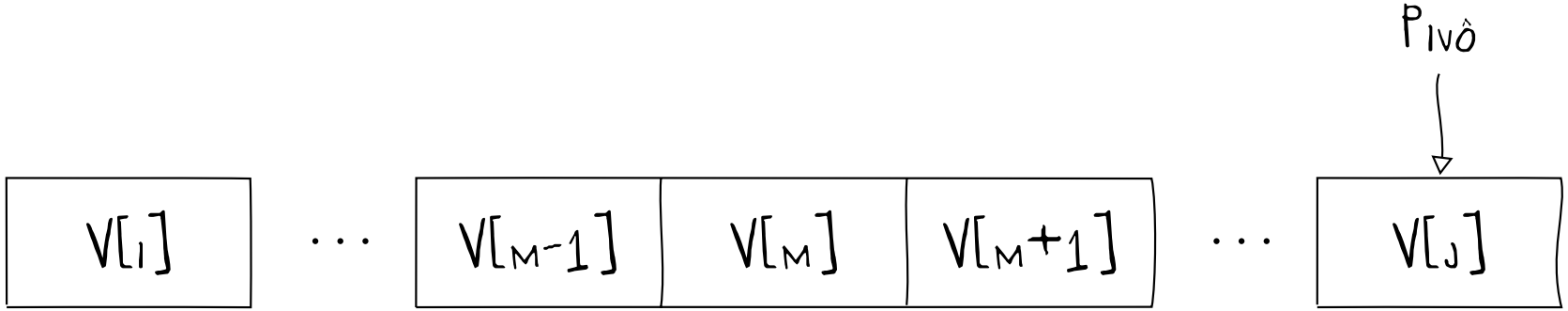
Quicksort

- Etapa de divisão



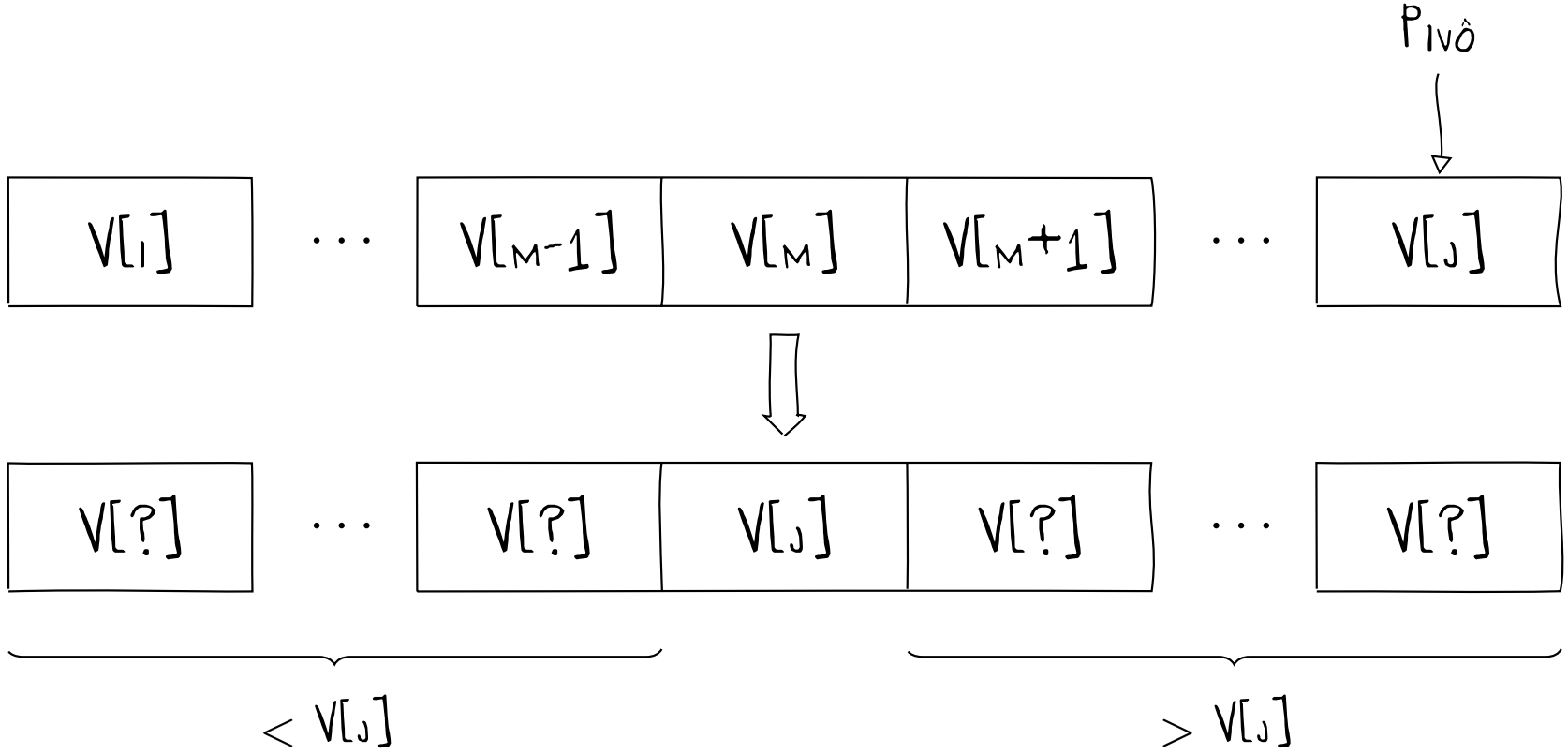
Quicksort

- Etapa de divisão
  - Particionamento do vetor por um pivô



Quicksort

- Etapa de divisão
  - Particionamento do vetor por um pivô



Quicksort

- Etapa de divisão

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
15  // Quicksort recursivo
16  void quicksort(int32_t* V, int32_t i, int32_t j) {
17      // Caso base
18      if(i < j) {
19          // Particionamento do vetor
20          int32_t p = lomuto(V, i, j);
21          // Divisão em subvetores
22          quicksort(V, i, p - 1);
23          quicksort(V, p + 1, j);
24      }
25  }
```

Quicksort

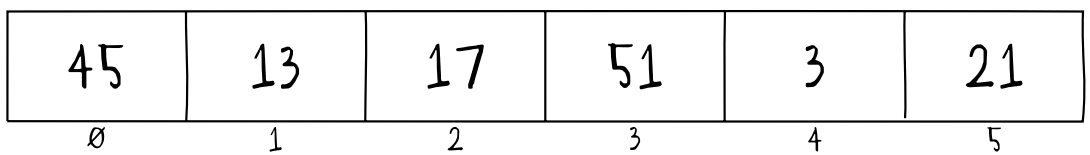
- Etapa de divisão

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
15  // Quicksort recursivo
16  void quicksort(int32_t* V, int32_t i, int32_t j) {
17      // Caso base
18      if(i < j) {
19          // Particionamento do vetor
20          int32_t p = lomuto(V, i, j);
21          // Divisão em subvetores
22          quicksort(V, i, p - 1);
23          quicksort(V, p + 1, j);
24      }
25  }
```

Quicksort

- Etapa de conquista

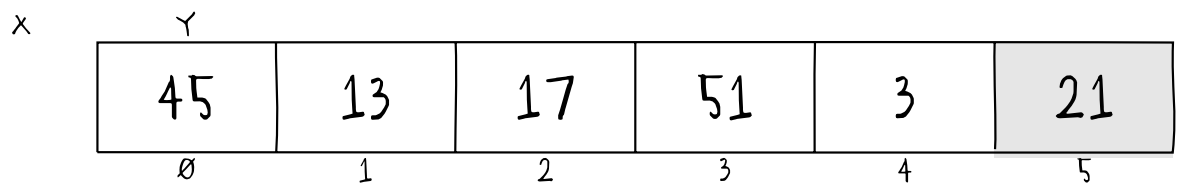
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Lomuto)
4   int32_t lomuto(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[j], x = i - 1, y = i;
7       // Particionando o vetor pelo pivô
8       for(y = i; y < j; y++)
9           if(V[y] <= P) trocar(&V[++x], &V[y]);
10      // Posicionando o pivô no vetor
11      trocar(&V[++x], &V[j]);
12      // Retornando índice do pivô
13      return x;
14  }
…   ...
```



Quicksort

- Etapa de conquista

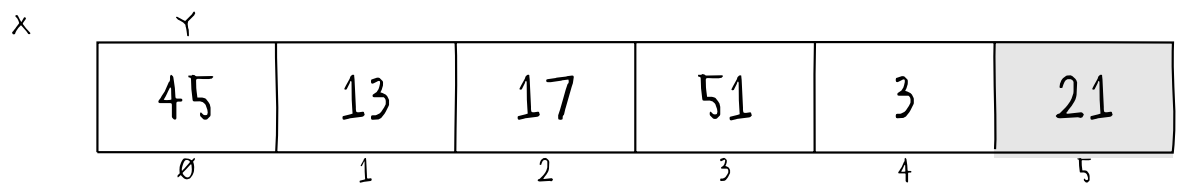
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Lomuto)
4   int32_t lomuto(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[j], x = i - 1, y = i;
7       // Particionando o vetor pelo pivô
8       for(y = i; y < j; y++)
9           if(V[y] <= P) trocar(&V[++x], &V[y]);
10      // Posicionando o pivô no vetor
11      trocar(&V[++x], &V[j]);
12      // Retornando índice do pivô
13      return x;
14  }
…   ...
```



Quicksort

- Etapa de conquista

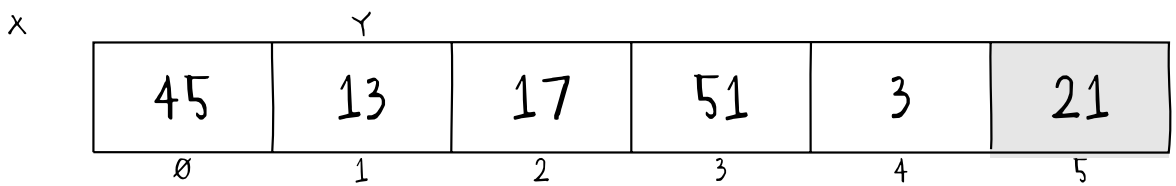
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Lomuto)
4   int32_t lomuto(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[j], x = i - 1, y = i;
7       // Particionando o vetor pelo pivô
8       for(y = i; y < j; y++)
9           if(V[y] <= P) trocar(&V[++x], &V[y]);
10      // Posicionando o pivô no vetor
11      trocar(&V[++x], &V[j]);
12      // Retornando índice do pivô
13      return x;
14  }
…   ...
```



Quicksort

- Etapa de conquista

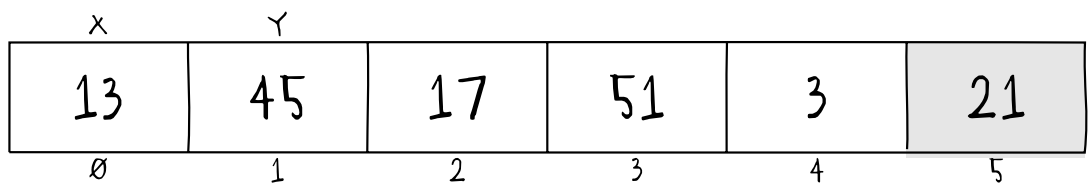
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Lomuto)
4   int32_t lomuto(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[j], x = i - 1, y = i;
7       // Particionando o vetor pelo pivô
8       for(y = i; y < j; y++)
9           if(V[y] <= P) trocar(&V[++x], &V[y]);
10      // Posicionando o pivô no vetor
11      trocar(&V[++x], &V[j]);
12      // Retornando índice do pivô
13      return x;
14  }
…   ...
```



Quicksort

- Etapa de conquista

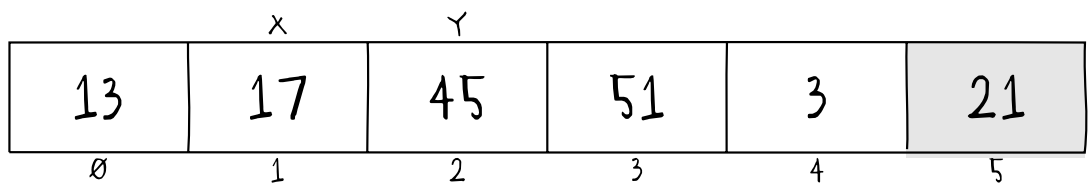
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Lomuto)
4   int32_t lomuto(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[j], x = i - 1, y = i;
7       // Particionando o vetor pelo pivô
8       for(y = i; y < j; y++)
9           if(V[y] <= P) trocar(&V[++x], &V[y]);
10      // Posicionando o pivô no vetor
11      trocar(&V[++x], &V[j]);
12      // Retornando índice do pivô
13      return x;
14  }
…   ...
```



Quicksort

- Etapa de conquista

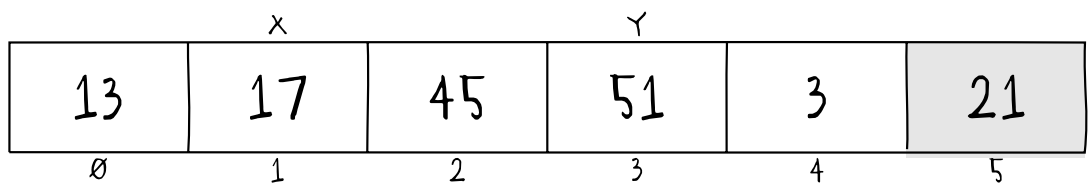
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Lomuto)
4   int32_t lomuto(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[j], x = i - 1, y = i;
7       // Particionando o vetor pelo pivô
8       for(y = i; y < j; y++)
9           if(V[y] <= P) trocar(&V[++x], &V[y]);
10      // Posicionando o pivô no vetor
11      trocar(&V[++x], &V[j]);
12      // Retornando índice do pivô
13      return x;
14  }
…   ...
```



Quicksort

- Etapa de conquista

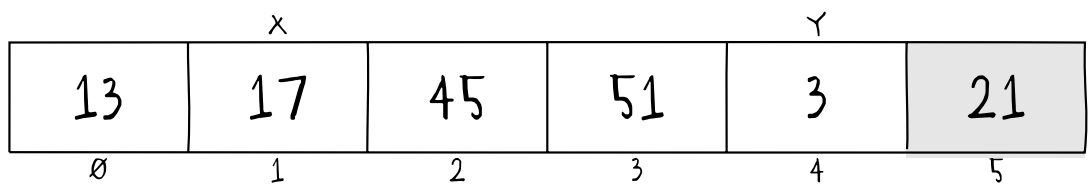
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Lomuto)
4   int32_t lomuto(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[j], x = i - 1, y = i;
7       // Particionando o vetor pelo pivô
8       for(y = i; y < j; y++)
9           if(V[y] <= P) trocar(&V[++x], &V[y]);
10      // Posicionando o pivô no vetor
11      trocar(&V[++x], &V[j]);
12      // Retornando índice do pivô
13      return x;
14  }
…   ...
```



Quicksort

- Etapa de conquista

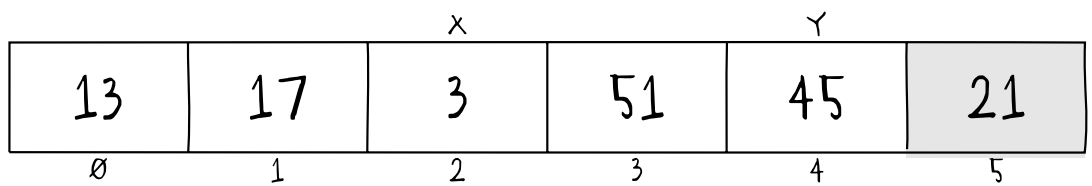
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Lomuto)
4   int32_t lomuto(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[j], x = i - 1, y = i;
7       // Particionando o vetor pelo pivô
8       for(y = i; y < j; y++)
9           if(V[y] <= P) trocar(&V[++x], &V[y]);
10      // Posicionando o pivô no vetor
11      trocar(&V[++x], &V[j]);
12      // Retornando índice do pivô
13      return x;
14  }
…   ...
```



Quicksort

- Etapa de conquista

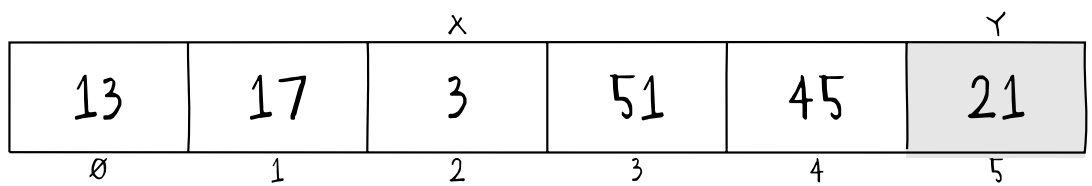
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Lomuto)
4   int32_t lomuto(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[j], x = i - 1, y = i;
7       // Particionando o vetor pelo pivô
8       for(y = i; y < j; y++)
9           if(V[y] <= P) trocar(&V[++x], &V[y]);
10      // Posicionando o pivô no vetor
11      trocar(&V[++x], &V[j]);
12      // Retornando índice do pivô
13      return x;
14  }
…   ...
```



Quicksort

- Etapa de conquista

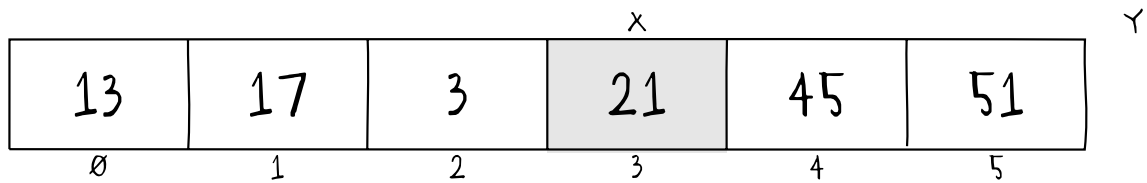
```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Lomuto)
4   int32_t lomuto(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[j], x = i - 1, y = i;
7       // Particionando o vetor pelo pivô
8       for(y = i; y < j; y++)
9           if(V[y] <= P) trocar(&V[++x], &V[y]);
10      // Posicionando o pivô no vetor
11      trocar(&V[++x], &V[j]);
12      // Retornando índice do pivô
13      return x;
14  }
…   ...
```



Quicksort

- Etapa de conquista

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Lomuto)
4   int32_t lomuto(int32_t* V, int32_t i, int32_t j) {
5       // Declaração do pivô e índices
6       int32_t P = V[j], x = i - 1, y = i;
7       // Particionando o vetor pelo pivô
8       for(y = i; y < j; y++)
9           if(V[y] <= P) trocar(&V[++x], &V[y]);
10      // Posicionando o pivô no vetor
11      trocar(&V[++x], &V[j]);
12      // Retornando índice do pivô
13      return x;
14  }
…   ...
```



Quicksort

- Etapa de divisão

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
…  ...
15 // Quicksort recursivo
16 void quicksort(int32_t* V, int32_t i, int32_t j) {
17     // Caso base
18     if(i < j) {
19         // Particionamento do vetor
20         int32_t p = lomuto(V, i, j);
21         // Divisão em subvetores
22         quicksort(V, i, p - 1);
23         quicksort(V, p + 1, j);
24     }
25 }
```

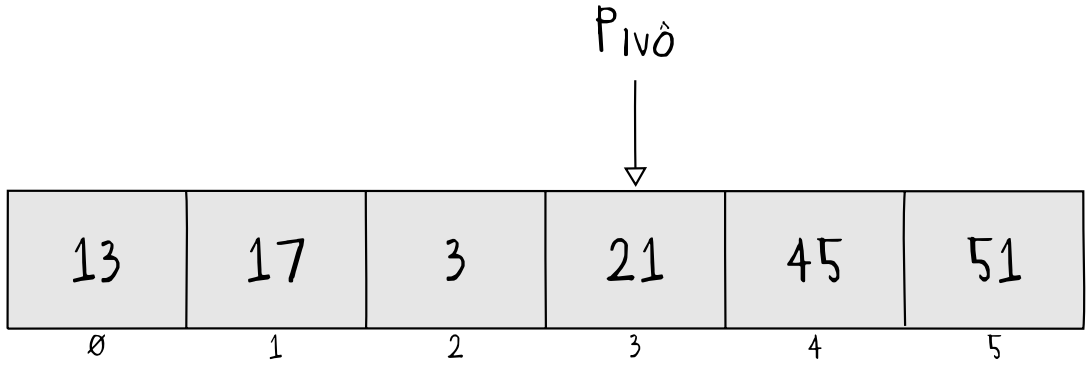
Quicksort

- Etapa de divisão

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
…   ...
15  // Quicksort recursivo
16  void quicksort(int32_t* V, int32_t i, int32_t j) {
17      // Caso base
18      if(i < j) {
19          // Particionamento do vetor
20          int32_t p = lomuto(V, i, j);
21          // Divisão em subvetores
22          quicksort(V, i, p - 1);
23          quicksort(V, p + 1, j);
24      }
25  }
```

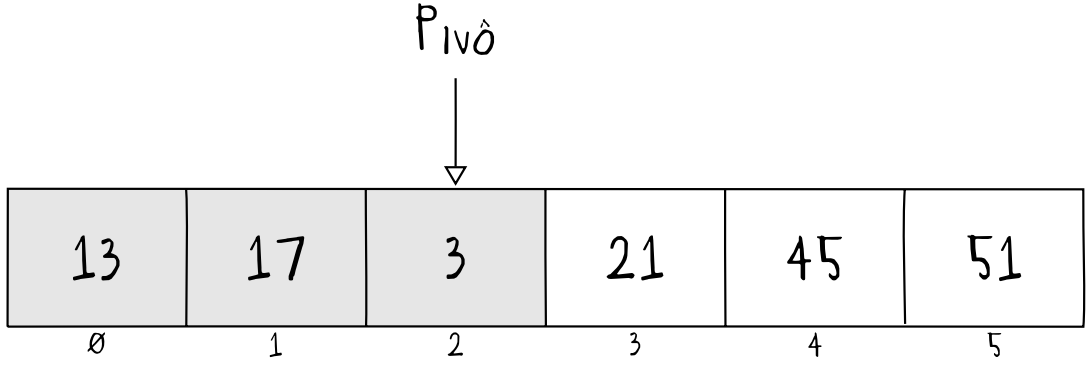
Quicksort

- Etapa de divisão



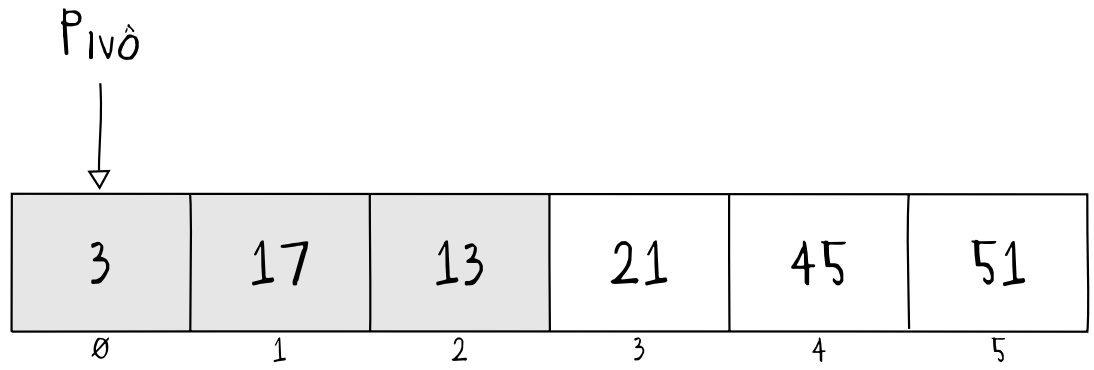
Quicksort

- Etapa de divisão



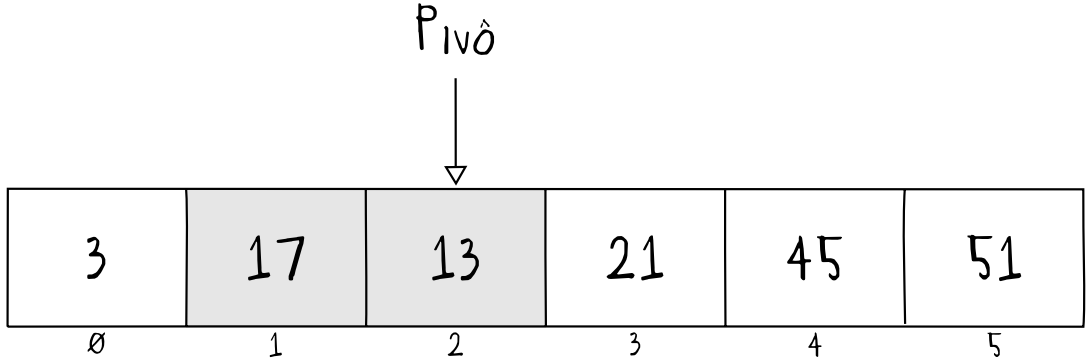
Quicksort

- Etapa de divisão



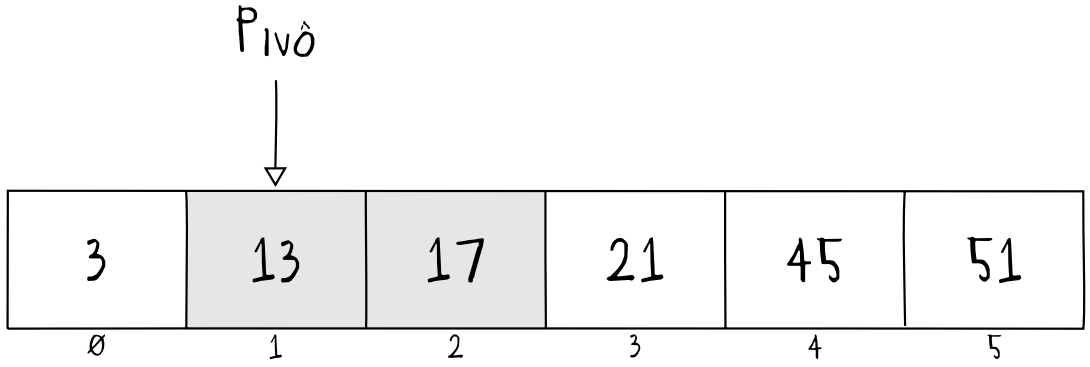
Quicksort

- Etapa de divisão



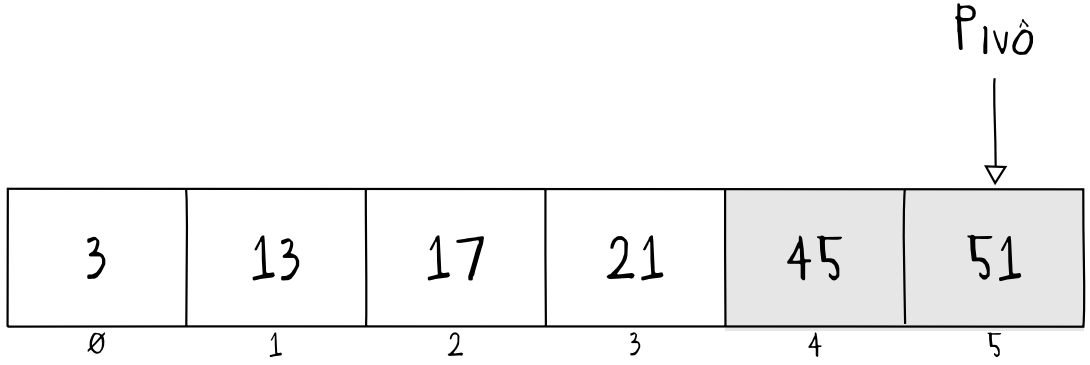
Quicksort

- Etapa de divisão



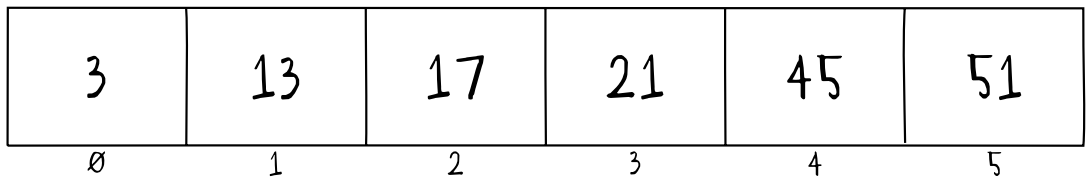
Quicksort

- Etapa de divisão



Quicksort

- Etapa de divisão



Quicksort

- Papel do particionamento no Quicksort
  - É a própria ordenação

Quicksort

- Papel do particionamento no Quicksort
  - É a própria ordenação
  - Qual a melhor forma de particionar? E qual a pior?

Quicksort

- Papel do particionamento no Quicksort
  - É a própria ordenação
  - Qual a melhor forma de particionar? E qual a pior?
  - Existem várias estratégias de particionamento

Quicksort

- Particionamento randômico

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare randômico)
4   int32_t hoare_rand(int32_t* V, int32_t i, int32_t j) {
5       // Troca do pivô por aleatório
6       trocar(&V[i], &V[i + (rand() % (j - i + 1))]);
7       // Chamada do particionamento
8       return hoare(V, i, j);
9   }
10  // Particionamento do Quicksort (Lomuto randômico)
11  int32_t lomuto_rand(int32_t* V, int32_t i, int32_t j) {
12      // Troca do pivô por aleatório
13      trocar(&V[j], &V[i + (rand() % (j - i + 1))]);
14      // Chamada do particionamento
15      return lomuto(V, i, j);
16  }
…   ...
```

Quicksort

- Particionamento randômico

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare randômico)
4   int32_t hoare_rand(int32_t* V, int32_t i, int32_t j) {
5       // Troca do pivô por aleatório
6       trocar(&V[i], &V[i + (rand() % (j - i + 1))]);
7       // Chamada do particionamento
8       return hoare(V, i, j);
9   }
10  // Particionamento do Quicksort (Lomuto randômico)
11  int32_t lomuto_rand(int32_t* V, int32_t i, int32_t j) {
12      // Troca do pivô por aleatório
13      trocar(&V[j], &V[i + (rand() % (j - i + 1))]);
14      // Chamada do particionamento
15      return lomuto(V, i, j);
16  }
…   ...
```

Quicksort

- Particionamento randômico

```
1   // Padrão de tipos por tamanho
2   #include <stdint.h>
3   // Particionamento do Quicksort (Hoare randômico)
4   int32_t hoare_rand(int32_t* V, int32_t i, int32_t j) {
5       // Troca do pivô por aleatório
6       trocar(&V[i], &V[i + (rand() % (j - i + 1))]);
7       // Chamada do particionamento
8       return hoare(V, i, j);
9   }
10  // Particionamento do Quicksort (Lomuto randômico)
11  int32_t lomuto_rand(int32_t* V, int32_t i, int32_t j) {
12      // Troca do pivô por aleatório
13      trocar(&V[j], &V[i + (rand() % (j - i + 1))]);
14      // Chamada do particionamento
15      return lomuto(V, i, j);
16  }
...
```

Quicksort

- Particionamento pela mediana de 3
  - São escolhidos três índices heuristicamente

Quicksort

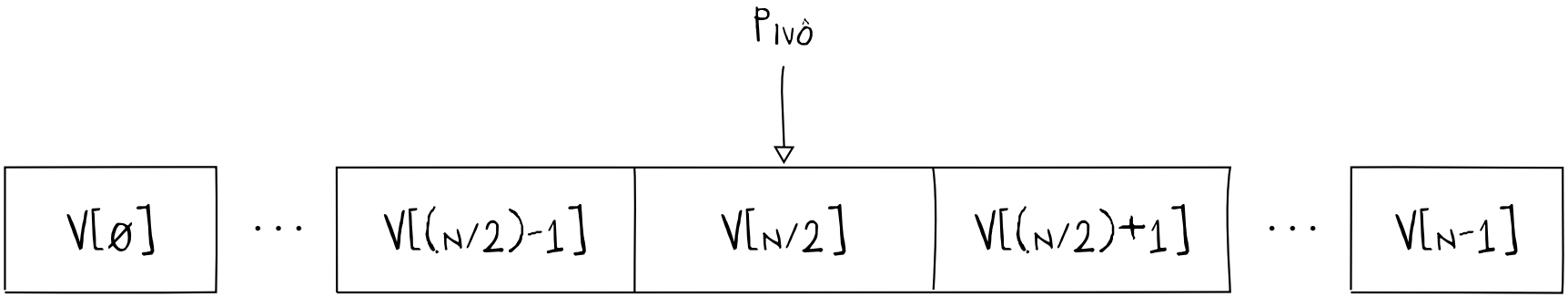
- Particionamento pela mediana de 3
  - São escolhidos três índices heuristicamente
  - O pivô é definido pela mediana destes elementos

Quicksort

- Particionamento pela mediana de 3
  - São escolhidos três índices heuristicamente
  - O pivô é definido pela mediana destes elementos
  - Esta estratégia reduz a probabilidade de escolha de um pivô que gere um cenário de pior caso

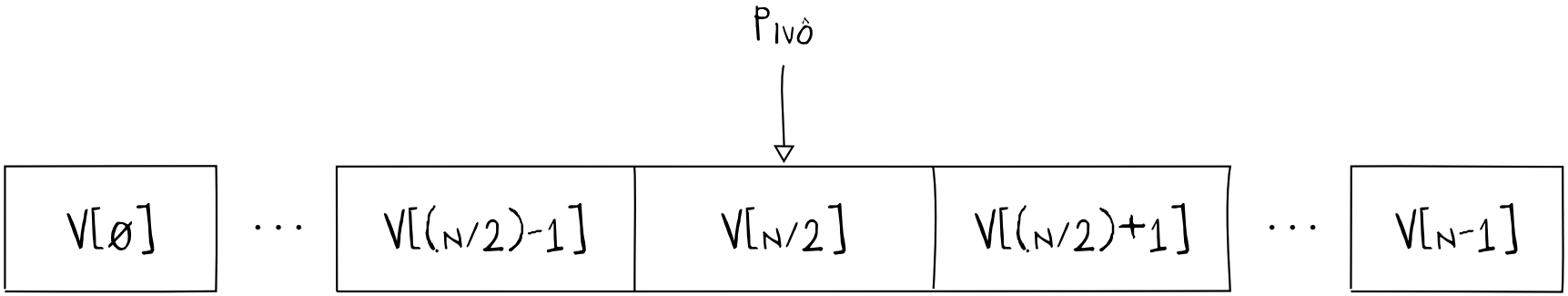
Quicksort

- Análise de complexidade
  - Melhor caso $\Omega(n \log n)$, o particionamento sempre é feito no meio do vetor, dividindo a entrada em subvetores de tamanhos próximos



Quicksort

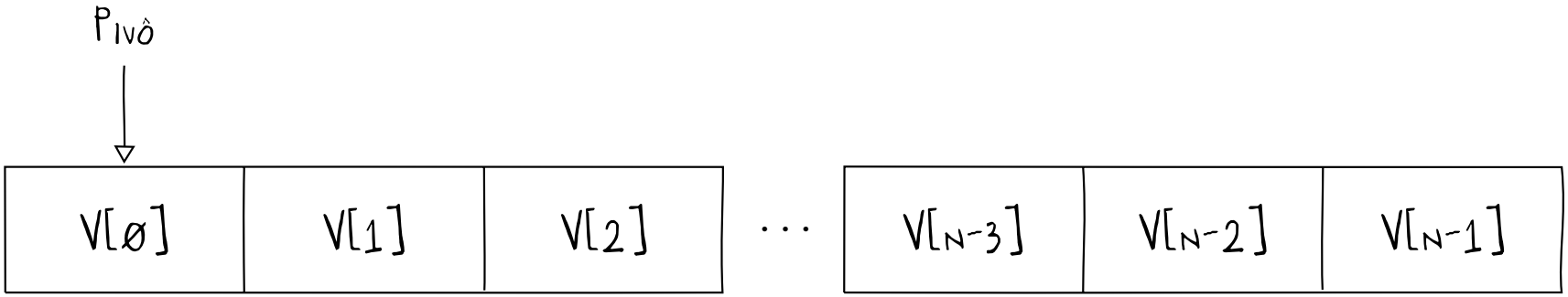
- Análise de complexidade
  - Melhor caso $\Omega(n \log n)$, o particionamento sempre é feito no meio do vetor, dividindo a entrada em subvetores de tamanhos próximos



$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

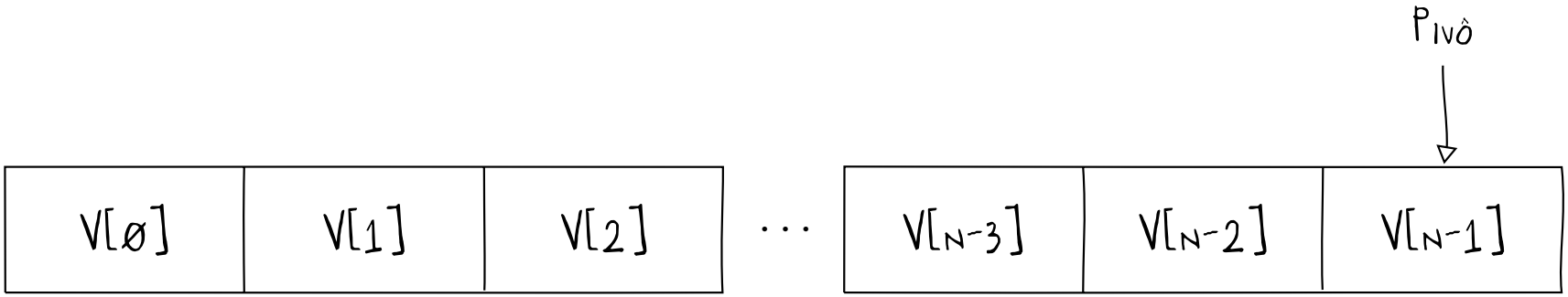
Quicksort

- Análise de complexidade
  - No pior caso $O(n^2)$, o particionamento sempre é feito nas extremidades do vetor, criando um subvetor com tamanho próximo da entrada



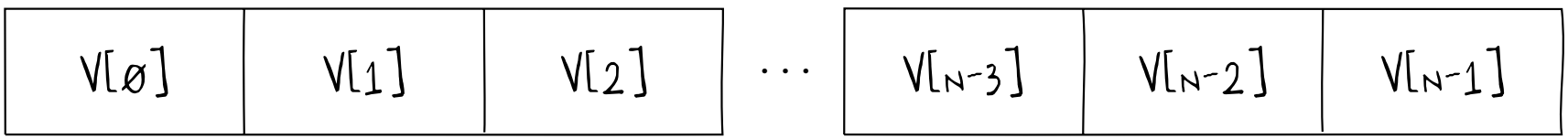
Quicksort

- Análise de complexidade
  - No pior caso $O(n^2)$, o particionamento sempre é feito nas extremidades do vetor, criando um subvetor com tamanho próximo da entrada



Quicksort

- Análise de complexidade
  - No pior caso $O(n^2)$, o particionamento sempre é feito nas extremidades do vetor, criando um subvetor com tamanho próximo da entrada



$$T(n) = T(n-1) + n$$

Quicksort

- Ordem estatística (seleção)
  - Em uma sequência de números distintos $S = x_1, x_2, \ldots, x_{n-1}, x_n$, com um inteiro k tal que $1 \leq k \leq n$, o k-ésimo elemento é maior que todos os seus antecessores $x_1, \ldots, x_{k-1}$

Quicksort

- Ordem estatística (seleção)
  - Em uma sequência de números distintos $S = x_1, x_2, \ldots, x_{n-1}, x_n$, com um inteiro k tal que $1 \leq k \leq n$, o k-ésimo elemento é maior que todos os seus antecessores $x_1, \ldots, x_{k-1}$
  - O menor elemento ($k = 1$) é o primeiro da ordem e o maior elemento ($k = n$) é o n-ésimo

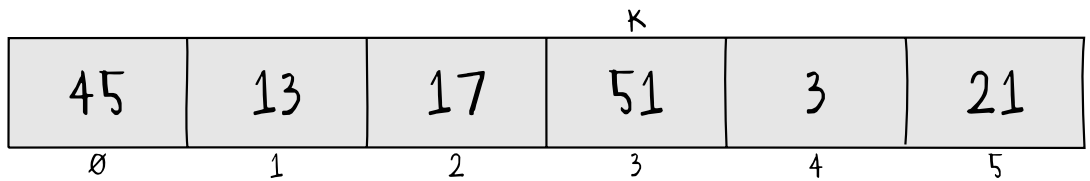
Quicksort

- Ordem estatística (seleção)
  - Em uma sequência de números distintos $S = x_1, x_2, \ldots, x_{n-1}, x_n$, com um inteiro k tal que $1 \leq k \leq n$, o k-ésimo elemento é maior que todos os seus antecessores $x_1, \ldots, x_{k-1}$
  - O menor elemento ($k = 1$) é o primeiro da ordem e o maior elemento ($k = n$) é o n-ésimo
  - Como a escolha de um pivô adequado tem impacto direto no desempenho do Quicksort, este algoritmo de seleção é capaz de encontrar a mediana com complexidade média esperada $\Omega(n)$ e pior caso improvável $O(n^2)$

Quicksort

- Ordem estatística (seleção)
  - Encontrar a mediana $k = \frac{n}{2} = 3$

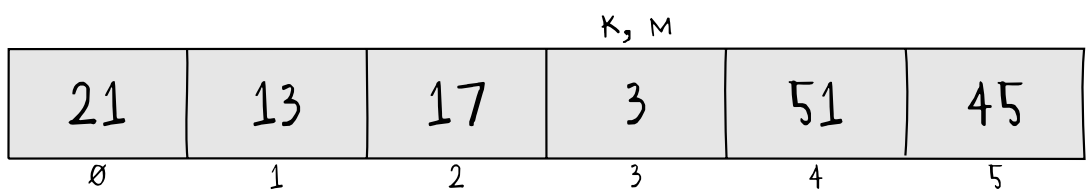
```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Algoritmo de seleção
4  int32_t selecao(int32_t* V, int32_t i, int32_t j,
       int32_t k) {
5      // Caso base
6      if(i == j) return i;
7      // Recorrência
8      else {
9          int32_t m = hoare(V, i, j);
10         if(m - i + 1 >= k) selecao(V, i, m, k);
11         else selecao(V, m + 1, j, k - (m - i + 1));
12     }
13 }
```



Quicksort

- Ordem estatística (seleção)
  - Encontrar a mediana $k = \frac{n}{2} = 3$

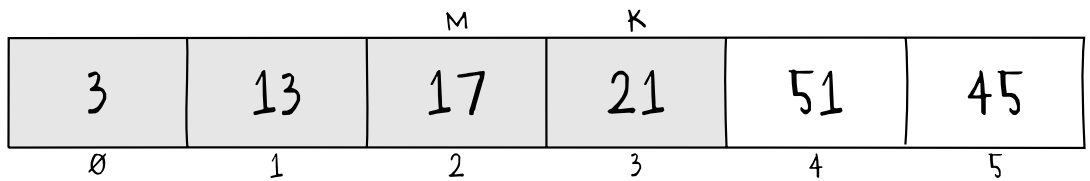
```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Algoritmo de seleção
4  int32_t selecao(int32_t* V, int32_t i, int32_t j,
       int32_t k) {
5      // Caso base
6      if(i == j) return i;
7      // Recorrência
8      else {
9          int32_t m = hoare(V, i, j);
10         if(m - i + 1 >= k) selecao(V, i, m, k);
11         else selecao(V, m + 1, j, k - (m - i + 1));
12     }
13 }
```



Quicksort

- Ordem estatística (seleção)
  - Encontrar a mediana $k = \frac{n}{2} = 3$

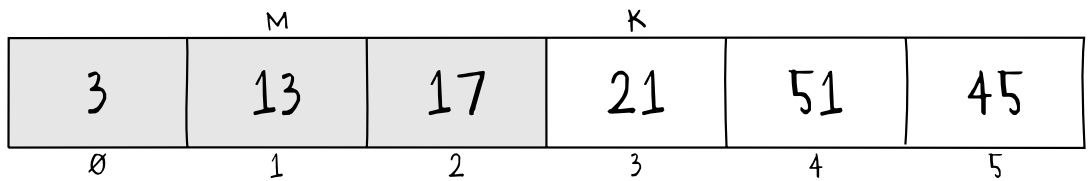
```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Algoritmo de seleção
4  int32_t selecao(int32_t* V, int32_t i, int32_t j,
       int32_t k) {
5      // Caso base
6      if(i == j) return i;
7      // Recorrência
8      else {
9          int32_t m = hoare(V, i, j);
10         if(m - i + 1 >= k) selecao(V, i, m, k);
11         else selecao(V, m + 1, j, k - (m - i + 1));
12     }
13 }
```



Quicksort

- Ordem estatística (seleção)
  - Encontrar a mediana $k = \frac{n}{2} = 3$

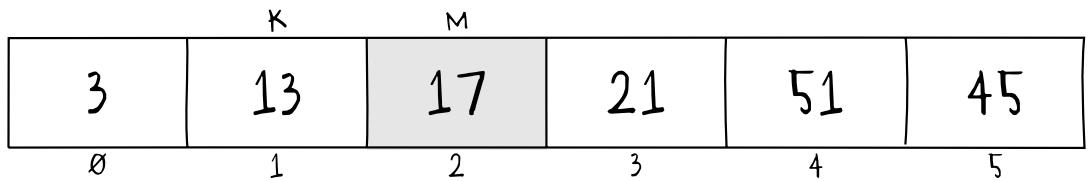
```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Algoritmo de seleção
4  int32_t selecao(int32_t* V, int32_t i, int32_t j,
       int32_t k) {
5      // Caso base
6      if(i == j) return i;
7      // Recorrência
8      else {
9          int32_t m = hoare(V, i, j);
10         if(m - i + 1 >= k) selecao(V, i, m, k);
11         else selecao(V, m + 1, j, k - (m - i + 1));
12     }
13 }
```



Quicksort

- Ordem estatística (seleção)
  - Encontrar a mediana $k = \frac{n}{2} = 3$

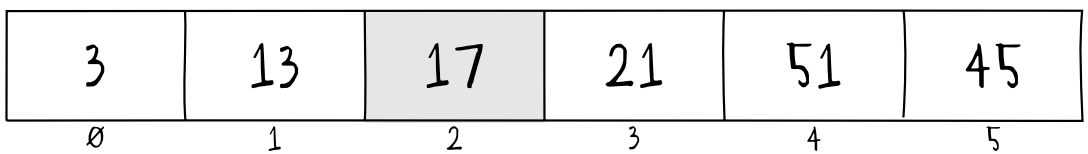
```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Algoritmo de seleção
4  int32_t selecao(int32_t* V, int32_t i, int32_t j,
       int32_t k) {
5      // Caso base
6      if(i == j) return i;
7      // Recorrência
8      else {
9          int32_t m = hoare(V, i, j);
10         if(m - i + 1 >= k) selecao(V, i, m, k);
11         else selecao(V, m + 1, j, k - (m - i + 1));
12     }
13 }
```



Quicksort

- Ordem estatística (seleção)
  - Encontrar a mediana $k = \frac{n}{2} = 3$

```
1  // Padrão de tipos por tamanho
2  #include <stdint.h>
3  // Algoritmo de seleção
4  int32_t selecao(int32_t* V, int32_t i, int32_t j,
       int32_t k) {
5      // Caso base
6      if(i == j) return i;
7      // Recorrência
8      else {
9          int32_t m = hoare(V, i, j);
10         if(m - i + 1 >= k) selecao(V, i, m, k);
11         else selecao(V, m + 1, j, k - (m - i + 1));
12     }
13 }
```



Quicksort

- Características do Quicksort
  ✓ Paralelismo: a entrada é dividida em partes que podem ser resolvidas de forma paralela

Quicksort

- Características do Quicksort
  ✓ Paralelismo: a entrada é dividida em partes que podem ser resolvidas de forma paralela
  ✓ Eficiência de espaço $\Theta(n)$ e de tempo entre $\Omega(n \log n)$ e $O(n^2)$

Quicksort

- Características do Quicksort
  ✓ Paralelismo: a entrada é dividida em partes que podem ser resolvidas de forma paralela
  ✓ Eficiência de espaço $\Theta(n)$ e de tempo entre $\Omega(n \log n)$ e $O(n^2)$
  ✓ Acesso a memória mais eficiente: conjuntos de dados menores e sequenciais cabem na cache

Quicksort

- Características do Quicksort
  ✓ Paralelismo: a entrada é dividida em partes que podem ser resolvidas de forma paralela
  ✓ Eficiência de espaço $\Theta(n)$ e de tempo entre $\Omega(n \log n)$ e $O(n^2)$
  ✓ Acesso a memória mais eficiente: conjuntos de dados menores e sequenciais cabem na cache
  ✓ *In-place*: não utiliza espaço adicional, utilizando o próprio vetor de entrada

Quicksort

- Características do Quicksort
  - ✗ Recursão: a utilização de pilha que é limitada

Quicksort

- Características do Quicksort
  - ✗ Recursão: a utilização de pilha que é limitada
  - ✗ Não é estável, ignora a ordem relativa dos elementos

Quicksort

- Características do Quicksort
  - ✗ Recursão: a utilização de pilha que é limitada
  - ✗ Não é estável, ignora a ordem relativa dos elementos
  - ✗ Escolha dos casos base: evitar processamento desnecessário de entradas pequenas e triviais

Quicksort

- Características do Quicksort
  - ✗ Recursão: a utilização de pilha que é limitada
  - ✗ Não é estável, ignora a ordem relativa dos elementos
  - ✗ Escolha dos casos base: evitar processamento desnecessário de entradas pequenas e triviais
  - ✗ Subproblemas repetidos: subvetores idênticos

Exemplo

- Considerando o algoritmo de ordenação Quicksort, ordene o vetor 23, 32, 54, 92, 74, 23, 1, 43, 63 e 12
  - Utilize o critério crescente de ordenação
  - Aplique os particionamentos de Hoare e Lomuto
  - Execute passo a passo cada etapa dos algoritmos

Exercício

- A empresa de desenvolvimento de sistemas Poxim Tech está realizando um experimento para determinar qual variante do algoritmo de ordenação crescente do Quicksort apresenta o melhor resultado para um determinado conjunto de sequências numéricas
  - Neste experimento foram utilizadas as seguintes variantes: Lomuto padrão (LP), Lomuto por mediana de 3 (LM), Lomuto por pivô aleatório (LA), Hoare padrão (HP), Hoare por mediana de 3 (HM) e Hoare por pivô aleatório (HA).
  - Técnicas de escolha do pivô
    - Mediana de 3: $V_1 = V\left[\frac{n}{4}\right], V_2 = V\left[\frac{n}{2}\right], V_3 = V\left[\frac{3n}{4}\right]$
    - Aleatório: $V_a = V\left[ini + |V\left[ini\right]| \bmod n\right]$

Exercício

- Formato de arquivo de entrada
  - [#*n total de vetores*]
  - [#*N*1 *números do vetor* 1]
  - $[E_1] \cdots [E_{N1}]$
  - ...
  - [#*Nn números do vetor n*]
  - $[E_1] \cdots [E_{Nn}]$

```
4
6
-23 10 7 -34 432 3
4
955 -32 1 9
7
834 27 39 19 3 -1 -33
10
847 38 -183 -13 94 -2 -42 54 28 100
```

Exercício

- Formato de arquivo de saída
  - Para cada vetor é impressa a quantidade total de números N e a sequência com ordenação estável contendo o número de trocas e de chamadas

```
1 [6]:LP(15),HP(16),LM(19),HM(19),HA(20),LA(22)
2 [4]:LP(10),HP(10),LM(11),LA(11),HM(12),HA(12)
3 [7]:HP(17),LM(18),LP(23),HM(26),HA(27),LA(30)
4 [10]:LM(28),HP(28),LP(33),HA(35),HM(37),LA(38)
```