

Problema da separação e quickSort

Projeto de algoritmos por divisão e conquista

- **Dividir**: o problema é dividido em subproblemas menores do mesmo tipo.
- **Conquistar**: os subproblemas são resolvidos recursivamente, sendo que os subproblemas pequenos são casos base.
- **Combinar**: as soluções dos subproblemas são combinadas numa solução do problema original.

Ideia e exemplo

- Separar o vetor entre os elementos **maiores** e **menores**.
 - Então, ordenar recursivamente cada subvetor resultante da separação
- Como exemplo, considere o seguinte vetor:

7	5	2	3	9	8
---	---	---	---	---	---

- Separar entre elementos menores e maiores.



- Ordenar cada subvetor recursivamente (lembrar dos casos base).



- a simples concatenação dos subvetores corresponde à solução.

Dificuldade: Como definir os maiores e os menores?

- Num algoritmo de ordenação baseado em comparações
 - só podemos falar de menor ou maior relativo a outro elemento.
- Por isso, usaremos um elemento do vetor como referência,
 - que chamaremos de **pivô**.
- Depois veremos como escolher esse elemento adequadamente.

Código quickSort recursivo:

```
// p indica a primeira posição e r a última
void quickSortR(int v[], int p, int r) {
    int i;
    if (p < r) { // se vetor corrente tem mais de um elemento
        i = separa2(v, p, r); // i é posição do pivô após separação
        quickSortR(v, [p, i - 1]); // 1º subvetor
        quickSortR(v, [i + 1, r]); // 2º subvetor
    }
}
```

Diagram illustrating the partitioning process: A vector v is shown with indices p , i , and r marked. The vector is divided into two parts: $v[p \dots i-1] < c$ and $v[i+1 \dots r] > c$, where c is the pivot value. The pivot element is at index i .

Curiosidade: note que a maior parte do trabalho é feita pela **função de separação**,

- na fase de divisão, que ocorre antes das chamadas recursivas.

Isso é complementar ao algoritmo mergeSort,

- que realiza a maior parte do trabalho na fase de combinação das soluções,
 - através da função de intercalação.

Por isso, podemos dizer que o mergeSort ordena o vetor “de baixo para cima”,

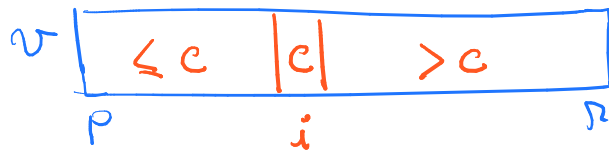
- enquanto o quickSort o ordena “de cima para baixo”.
- Isso fica mais claro se considerarmos a execução dos algoritmos
 - ilustrada numa árvore de recursão.

Assim como o algoritmo para o problema da intercalação é central no mergeSort,

- o algoritmo para o **problema da separação** é central no quickSort.
- Vamos entender melhor esse problema e projetar algoritmos para ele.

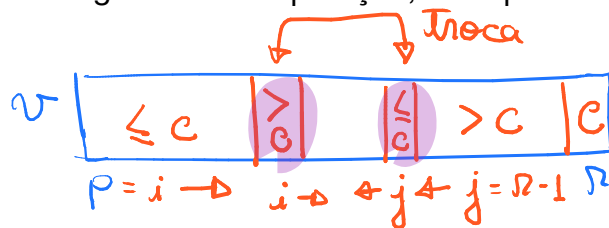
Problema da separação

- Receba como entrada um vetor $v[p \dots r]$,
 - e um pivô c , que é elemento de $v[p \dots r]$.
- O objetivo é separar os elementos do vetor de modo que
 - o prefixo deste tenha os elementos $\leq c$,
 - e o sufixo tenha os elementos $> c$.



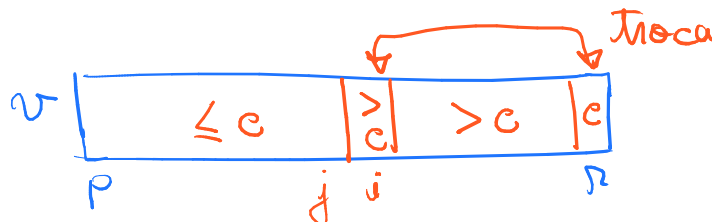
- Isto é, c deve terminar numa posição i tal que $v[p \dots i-1] \leq c = v[i] > v[i+1 \dots r]$
- Note que, c termina na posição que ele deve ocupar no vetor ordenado.

Ideia para um algoritmo de separação, exemplificado na seguinte figura,



consiste de:

- Escolher o pivô $c = v[r]$.
- Começar com um índice i em p e ir incrementando-o enquanto $v[i] \leq c$.
- Começar com outro índice j em $r - 1$ e ir decrementando-o enquanto $v[j] > c$.
- Quando ambos os índices param de avançar, temos
 - $v[i] > c$ e $v[j] \leq c$.
- Neste caso, troca $v[i]$ com $v[j]$ e volta a avançar os índices.
- Para o processo quando $i \geq j$,
 - caso em que fazer a troca não tem mais sentido.

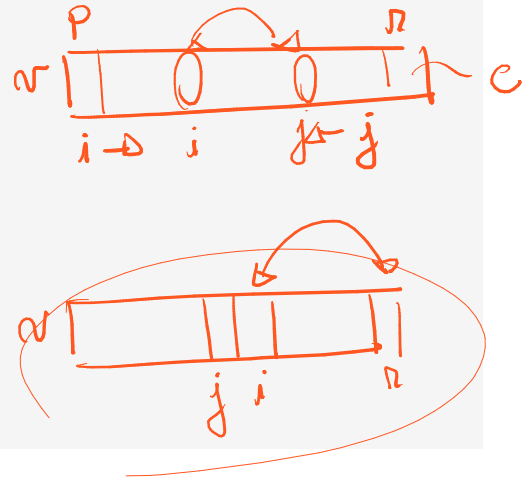


- Então troca $v[i]$ com $v[r]$
 - e devolve i .

Código do primeiro algoritmo da separação:

// separa $v[p \dots r]$ e devolve a posição do pivô

```
int separa1(int v[], int p, int r) {  
    int i = p, j = r - 1, c = v[r]; // c é o pivô  
    while (1) {  
        while (i < r && v[i] <= c) i++;  
        while (j > i && v[j] > c) j--;  
        if (i >= j) break;  
        troca(&v[i], &v[j]);  
        // i++; j--;  
    }  
    troca(&v[i], &v[r]);  
    return i;  
}
```



Invariantes e corretude do separa1:

- No início de cada iteração do laço temos
 - a) $v[p \dots r]$ é uma permutação do vetor original,
 - b) $v[p \dots i - 1] \leq c$, — prefixo tem os menores elementos
 - c) $v[j + 1 \dots r - 1] > c$, — "sufixo" tem os maiores "
 - d) $c = v[r]$.
- Note que, quando o algoritmo sai do laço principal temos $i \geq j$.
 - Portanto, todo o vetor está separado, exceto por c na posição r,
 - i.e., $v[p \dots i - 1] \leq c = v[r] < v[i \dots r - 1]$
 - de modo que $v[i]$ é o elemento mais à esquerda que é maior do que c.
- Assim, trocando $v[i]$ com $v[r]$ chegamos à solução.

Eficiência de tempo do separa1:

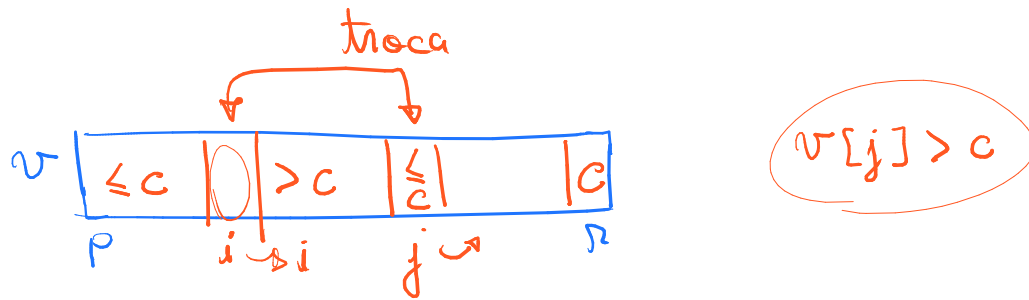
- O número de operações é linear no tamanho do subvetor, i.e., $O(r - p)$,
 - apesar dos laços aninhados sugerirem comportamento quadrático.
- Para verificar isso, note que no início $i = p$ e $j = r - 1$,
 - e em cada iteração dos laços internos
 - i é incrementado ou j é decrementado.
 - Assim, o número de iterações de ambos esse laços é $(i - p) + (r - 1 - j)$
- Note também que o laço principal termina quando $i \geq j$.
 - Assim, considerando que o laço terminou com $i = j + 1$ temos
 - # de iterações = $(i - p) + (r - 1 - j) = j + 1 - p + r - 1 - j = r - p$

Eficiência de espaço do separa1:

- $O(1)$, pois o número e tamanho das variáveis auxiliares é constante
 - em relação ao tamanho do vetor de entrada.

Curiosidade: Temos uma outra maneira de resolver o problema da separação,

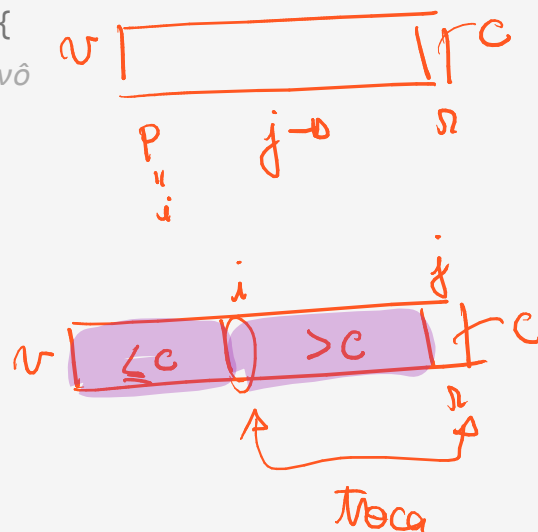
- que é exemplificada na seguinte figura



Código do segundo algoritmo da separação:

// separa $v[p \dots r]$ e devolve a posição do pivô

```
int separa2(int v[], int p, int r) {
    int i, j, c = v[r]; // c é o pivô
    i = p;
    for (j = p; j < r; j++) {
        if (v[j] <= c) {
            troca(&v[i], &v[j]);
            i++;
        }
    }
    troca(&v[i], &v[r]);
    return i;
}
```



Invariantes e corretude do separa2:

- No início de cada iteração do laço temos
 - $v[p \dots r]$ é uma permutação do vetor original,
 - $v[p \dots i - 1] \leq c$,
 - $c < v[i \dots j - 1]$,
 - $v[r] = c$,
 - $p \leq i \leq j \leq r$.
- Note que, como ao fim da última iteração $j = r$,
 - os invariantes implicam que a separação é realizada corretamente,
 - i.e., $v[p \dots i - 1] \leq c = v[r] < v[i \dots r - 1]$
- faltando apenas trocar o elemento em $v[i]$ com $v[r]$ e devolver i .

Eficiência de tempo do separa2:

- O número de operações é linear no tamanho do subvetor sendo intercalado,
 - ou seja, $O(r - p)$.
- Para verificar isso, note que o laço realiza $r - p$ iterações,
 - realizando trabalho constante em cada iteração.

Eficiência de espaço do separa2:

- $O(1)$, pois o número e tamanho das variáveis auxiliares é constante
 - em relação ao tamanho do vetor de entrada.

Relembrando o código do quickSort:

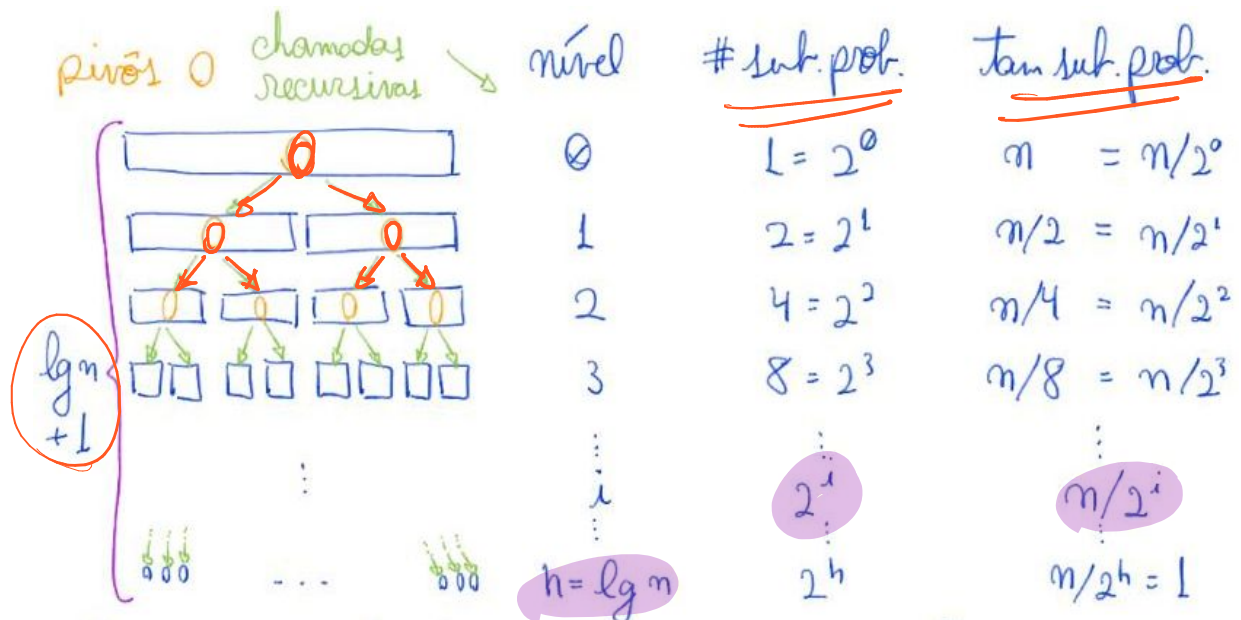
```
// p indica a primeira posicao e r a ultima
void quickSortR(int v[], int p, int r) {
    int i;
    if (p < r) { // se vetor corrente tem mais de um elemento
        i = separa2(v, p, r); // i é posição do pivô após separação
        quickSortR(v, p, i - 1);
        quickSortR(v, i + 1, r);
    }
}
```

Eficiência de tempo do quickSort depende de quão bem o vetor é dividido.

- Por isso, vamos comparar melhor caso, pior caso e caso médio.

Melhor caso:

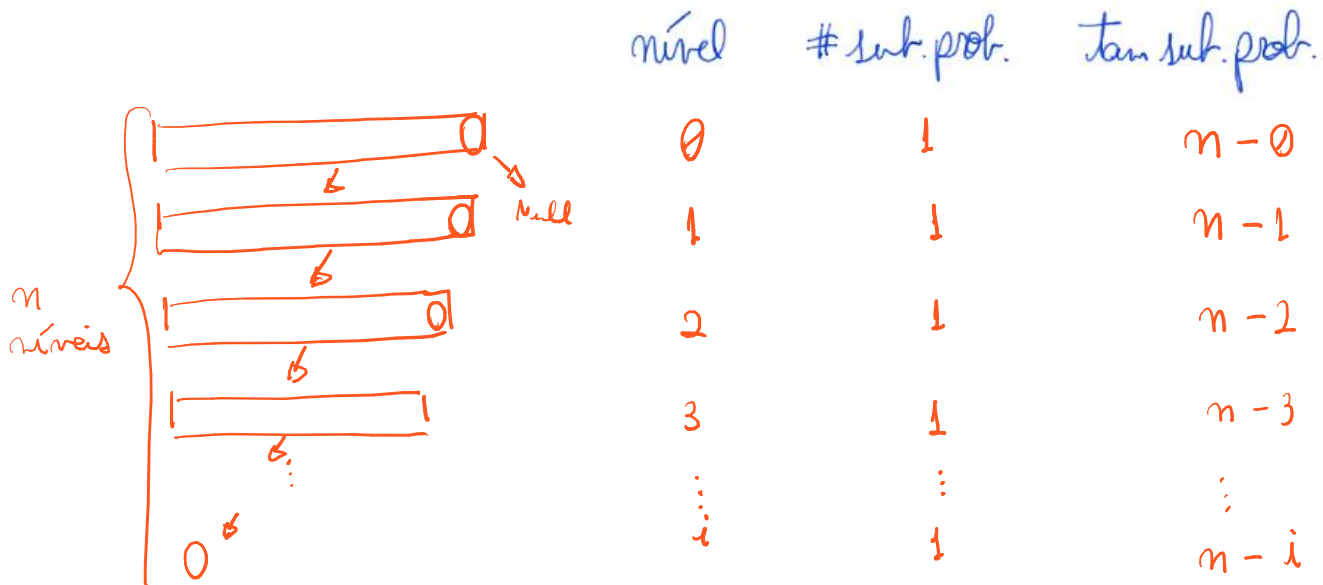
- Pivô sempre divide o vetor ao meio e número de operações é $O(n \lg n)$.
- Para chegar a esse resultado, construa uma árvore de recursão



- e observe que no nível i temos
 - 2^i subproblemas
 - e o vetor de cada subproblema tem tamanho $n / 2^i$.
- Como o trabalho das funções de separação
 - é linear no tamanho do vetor de entrada
 - o trabalho por subproblema do nível i é $z \cdot n/2^i$
 - para alguma constante z .
- Assim, trabalho total no nível i é $2^i \cdot z \cdot \frac{n}{2^i} = z \cdot n$
 - i.e., o trabalho é proporcional a n em todo nível.
- Como, no último nível h , por conta do caso base,
 - o tamanho dos subproblemas é 1,
 - temos $n / 2^h = 1 \Rightarrow 2^h = n \Rightarrow h = \lg n$.
- Portanto, o número de níveis é $(1 + \lg n)$,
 - já que começamos a contar os níveis em 0.
- Por fim, Trabalho Total = $z \cdot n \cdot (1 + \lg n) = O(n \lg n)$

Pior caso:

- Pivô sempre é o menor ou maior elemento do vetor
 - e número de operações é $O(n^2)$.
- Para chegar a esse resultado, observe que cada chamada recursiva
 - terá apenas um subproblema não trivial (tamanho vetor > 0)
 - e o vetor deste subproblema não trivial
 - será apenas uma unidade menor que o anterior.



- Assim, teremos sempre 1 subproblema por nível
 - e o tamanho do subproblema no nível i será $n-i$.
- Por isso, o trabalho no nível i será
 - para alguma constante z . $z \cdot (n-i)$
- O total de níveis será n , já que no último nível h temos
 - tamanho do subproblema = $1 = n-h \Rightarrow h = n-1$
 - e começamos a contar os níveis em 0.
- Assim, o trabalho total será a soma dos termos de uma PA

$$z(n + (n-1) + (n-2) + \dots + 2 + 1) = z(n+1) \frac{n}{2} \approx z \cdot \frac{n^2}{2} = O(n^2)$$

Caso médio:

- Quando lidando com vetores que são permutações aleatórias,
 - o número de operações fica próximo do melhor caso, i.e., $O(n \lg n)$.
- Pense que, por simetria, cada pivô tem a mesma chance
 - de ser um elemento grande ou pequeno.
- No entanto, a eficiência do quickSort determinístico
 - depende da entrada ter uma distribuição de valores favorável.
- Para não depender disso, podemos aleatorizar a escolha do pivô.
 - Com a aleatorização o tempo esperado do algoritmo é $O(n \lg n)$.
- Importante destacar que, no caso do algoritmo aleatorizado
 - a eficiência depende apenas de suas escolhas aleatórias,
 - e não da configuração do vetor de entrada.

Uso da aleatoriedade

Código quickSort recursivo aleatorizado:

```
// p indica a primeira posicao e r a ultima
```

```
void quickSortRAleat(int v[], int p, int r) { v[p..n]
```

```
    int desl, i;
```

```
    if (p < r) {
```

```
        // desl = rand() % (r - p + 1);
```

```
        desl = (int)(((double)rand() / (RAND_MAX + 1)) * ((double)(r - p + 1));
```

```
        troca(&v[p + desl], &v[r]);
```

```
        i = separa1(v, p, r);
```

```
        quickSortRAleat(v, p, i - 1);
```

```
        quickSortRAleat(v, i + 1, r);
```

```
    }
```

```
}
```

Funções de aleatorização:

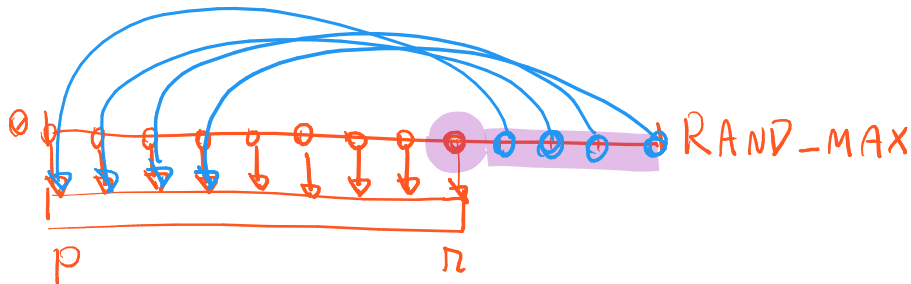
- A função `rand()`, definida na biblioteca `stdlib.h`,
 - gera um número inteiro pseudo-aleatório

- no intervalo `[0 .. RAND_MAX]`.

- Primeira opção

```
desl = rand() % (r - p + 1);
```

- Obtém um número inteiro no intervalo `[0, r - p]`, que corresponde
 - ao resto da divisão de um inteiro aleatório por $(r - p + 1)$.
- No entanto, possui um **viés que privilegia números pequenos**,
 - especialmente se $(r - p + 1)$ tem magnitude de `RAND_MAX`.



- Segunda opção

```
desl = (int)(((double)rand() / (RAND_MAX + 1)) * (r - p + 1));
```

- Analisando por partes, primeiro transforma o inteiro aleatório,
 - obtido de `rand()`, em um **número real** no intervalo `[0, 1)`

```
→ ((double)rand() / (RAND_MAX + 1))
```

- Depois, transforma esse real
 - em um número real no intervalo `[0, r - p + 1)`

```
(((double)rand() / (RAND_MAX + 1)) * ((double)(r - p + 1)))
```

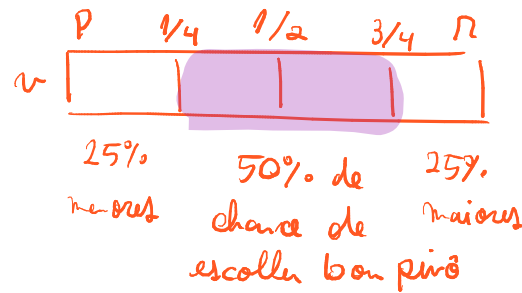
- Então, transforma esse real num inteiro no intervalo `[0, r - p]`
 - pois a conversão `(int)` trunca o valor alvo

```
(int)(((double)rand() / (RAND_MAX + 1)) * ((double)(r - p + 1)))
```

inteiro `[0, r - p]`

Eficiência de tempo esperada do quickSort aleatorizado:

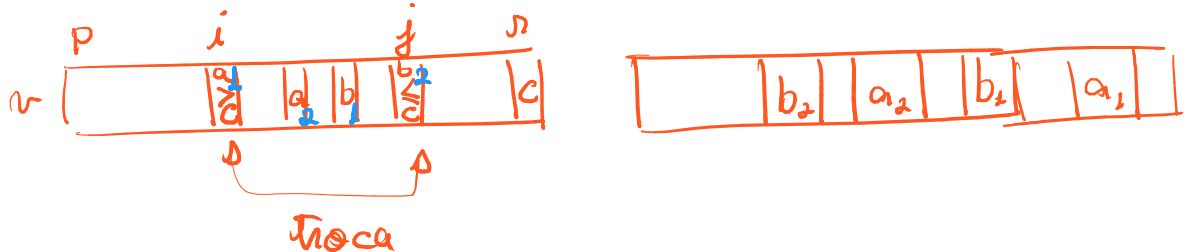
- Como dito antes, é da ordem de $n \lg n$, i.e., $O(n \lg n)$.
- Numa análise superficial, isso ocorre porque, em média,
 - a cada duas escolhas aleatórias do pivô, será escolhido
 - um pivô “bom”, que divide o vetor próximo da metade.



- É um cenário parecido com, a cada dois lances de moeda,
 - se espera obter uma cara.
- Com um pivô “bom” a cada dois, o resultado será uma árvore
 - parecida com a do melhor caso,
 - mas com um pouco mais que o dobro de níveis.
 - Aproximadamente $3,41 * (\lg n + 1)$ níveis.

Estabilidade:

- Ordenação do quickSort não é estável,
 - i.e., ele pode inverter a ordem relativa de elementos iguais.
- Isto acontece porque a rotina de separação troca elementos,
 - nas posições i e j , que estão separados por um intervalo.



- Assim, se existir um elemento x nesse intervalo, tal que $x = v[i]$ ou $x = v[j]$,
 - a ordem relativa destes elementos será invertida.

Eficiência de espaço do quickSort recursivo:

- quickSort não usa vetor auxiliar, o que levaria a classificá-lo como in place.
- No entanto, cada nova chamada recursiva
 - ocupa um pouco de memória da pilha de execução.
- Assim, quickSort ocupa memória adicional
 - proporcional à altura da pilha de execução,
 - que é igual à altura (número de níveis)
 - das árvores de recursão em nossas análises.
 - Ou seja, altura = $O(n)$ no pior caso e altura = $O(\lg n)$ nos demais casos.
- Portanto, o uso de memória cresce de acordo com o tamanho da entrada.
 - Por isso, quickSort não é propriamente in place.
- O uso de memória adicional proporcional a $\lg n$ não costuma ser crítico.
- Já, pilhas de execução de altura proporcional a n
 - podem dar problema em caso de n grande.

Bônus: melhorando a eficiência de espaço.

- Uma alternativa para garantir que o quickSort,
 - tanto na versão determinística quanto na probabilística,
- nunca chegue a produzir uma pilha de execução maior que $\lg n$
 - é sempre fazer a primeira chamada recursiva no menor subvetor,
 - que terá tamanho $\leq$ que metade do vetor anterior
 - e substituir a segunda chamada recursiva por uma versão iterativa.
- Vale destacar que isso só é possível porque
 - a segunda chamada recursiva do quickSort
 - é a última operação realizada na função.
- Isso caracteriza um caso de recursão caudal, a qual
 - pode ser convertida sistematicamente em um algoritmo iterativo.
- Para tanto, é introduzido um laço principal e
 - onde estaria a segunda chamada recursiva, é feita a atualização
 - dos índices para corresponderem ao novo subvetor.



O seguinte algoritmo implementa essa ideia na versão determinística do quickSort:

```
void quickSortRSemiIter(int v[], int p, int r) {
```

```
    int i; // caso base
```

```
    while (p < r) {
```

```
        i = separa1(v, p, r); ✓
```

```
        if (i - p < r - i) { // se o subvetor esquerdo é menor
```

```
            quickSortRSemiIter(v, p, i - 1); ←
```

```
            p = i + 1;
```

```
        }
```

```
        else { // se o subvetor direito é menor
```

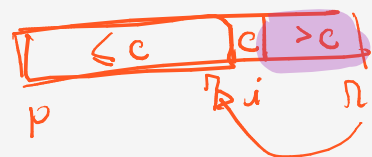
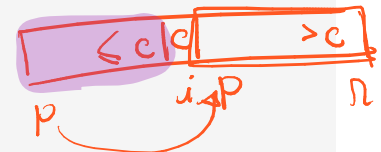
```
            quickSortRSemiIter(v, i + 1, r);
```

```
            r = i - 1;
```

```
        }
```

```
    }
```

```
}
```



Exercício:

- Uma última observação é que a eficiência do quickSort
 - pode ser comprometida se houverem muitos elementos repetidos.
- Nesse caso, mesmo a versão aleatorizada pode escolher muitos pivôs ruins.
 - Parte 1: construa um cenário em que isso acontece com quickSortR.
- Para evitar esse problema usamos o 3-way quickSort,
 - que divide o vetor entre menores, iguais e maiores que o pivô.
 - Parte 2: implemente essa versão do quickSort.

Animação:

- Visualization and Comparison of Sorting Algorithms -
www.youtube.com/watch?v=ZZuD6iUe3Pc

