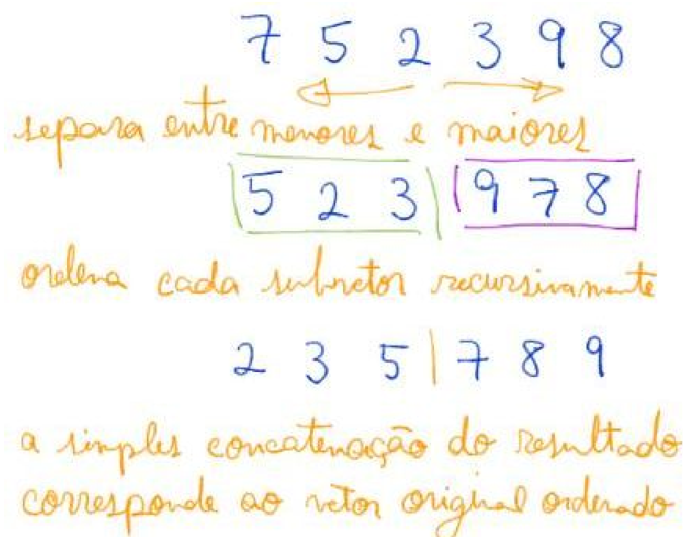
AED2 - Aula 11
Problema da separação e quicksort

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 caso 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
  - ordenar recursivamente cada subvetor resultante da separação
- Como exemplo, considere o vetor 7 5 2 3 9 8



Dificuldade:

- como definir os maiores e os menores?
  - num algoritmo de ordenação baseado em comparações
    - só podemos falar de menor ou maior relativo a outros elementos
    - 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 posicao e r a ultima
void quicksortR(int v[], int p, int r)
{
   int j;
```

```
    if (p < r)
    {
        j = separa1(v, p, r);
        quicksortR(v, p, j - 1);
        quicksortR(v, j + 1, r);
    }
}
```

Note que, no quicksort 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,
    - chamando a 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.

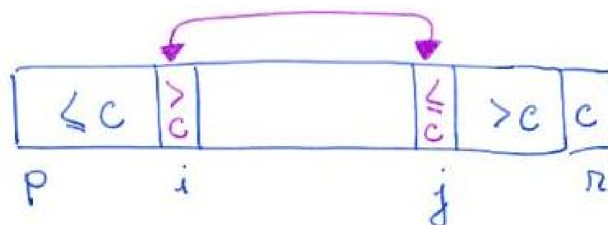
Assim como o algoritmo para o problema da intercalação é central no mergesort,

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

O problema da separação consiste de,

- dado um valor c e um vetor v com limites p e r,
    - i.e., os elementos do vetor estão em v[p .. r],
- separar os elementos do vetor de modo que
    - o prefixo deste tenha os elementos <= c,
    - e o sufixo tenha os elementos > c.
    - Isto é, c deve terminar numa posição i tal que:
        - v[p .. i - 1] <= c = v[i] < v[i + 1 .. r]
- Note que c termina na posição que ele deve ocupar no vetor ordenado.

Uma ideia para um algoritmo de separação, exemplificado na seguinte figura,



consiste de:

- escolher c = v[r]
- começar com um índice i em p e ir incrementando-o enquanto v[i] <= 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] <= c
- neste caso troca v[i] com v[j] e volta a avançar os índices.
- para o processo quando i >= j,
    - caso em que fazer a troca não tem mais sentido.
- então troca v[i] com v[r] e devolve i

Código primeiro algoritmo da separação:

```
int separa1(int v[], int p, int r)
{
   int i = p, j = r - 1, c = v[r];
   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
    - v[p .. r] é uma permutação do vetor original
    - v[p .. i - 1] <= c
    - v[j + 1 .. r - 1] > c
    - c = v[r]
- Note que, quando o algoritmo sai do laço principal temos i >= j.
    - Portanto, todo o vetor está separado, exceto por c na posição r,
        - i.e., v[p .. i - 1] <= c < v[i .. r - 1] e v[r] = c,
    - 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.

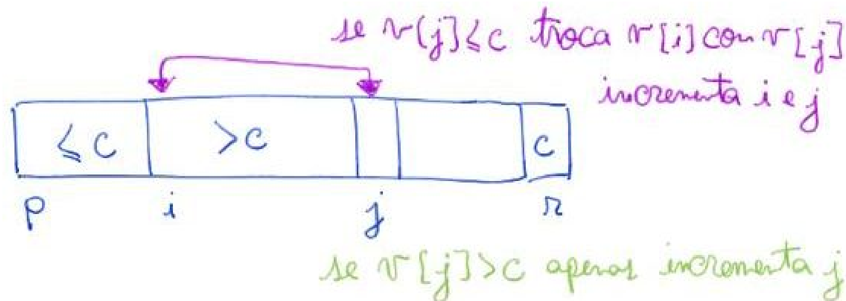
Eficiência de tempo do separa1:

- O número de operações é linear no tamanho do subvetor sendo intercalado,
    - i.e., O(r-p).
- Para verificar isso, note que

    - no início i = p e j = r - 1,
    - em cada iteração dos laços internos
        - i é incrementado ou j é decrementado,
    - e o laço principal termina quando i >= j.

Uma maneira diferente de resolver o problema da separação

- é exemplificada na seguinte figura



- e implementada, de forma sucinta, no seguinte código:

```
int separa2(int v[], int p, int r)
{
   int i, j, c = v[r];
   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 .. r] é uma permutação do vetor original
    - v[p .. i - 1] <= c < v[i .. j - 1], v[r] = c
    - p <= i <= j <= r
- Note que, como ao fim da última iteração j = r,
    - os invariantes implicam que a separação é realizada corretamente,
    - i.e., v[p .. i - 1] <= c < v[i .. r - 1] e v[r] = c,
- faltando apenas trocar o elemento em v[i] com v[r] e devolver i.

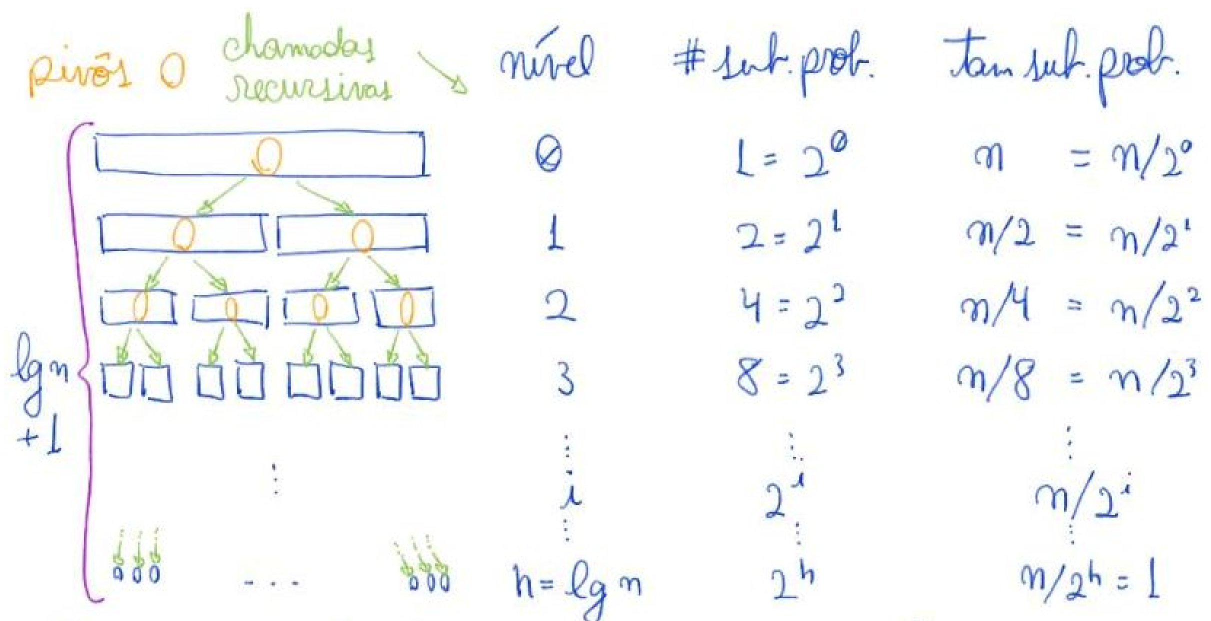
Eficiência de tempo do separa2:

- 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 tempo do quicksort:

- primeiro 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 l temos
        - 2^l subproblemas
        - e o vetor de cada subproblema tem tamanho $n/2^l$.
        - Como o trabalho das funções de separação é linear no tamanho do vetor de entrada
            - o trabalho por subproblema é const * $(n/2^l)$,
                - para alguma constante const.
        - Assim, trabalho total no nível l é
            - 2^l * const * $(n/2^l)$ = const * 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,
        - e o trabalho total = const * n * $(1 + \lg n)$ = $O(n \lg n)$.

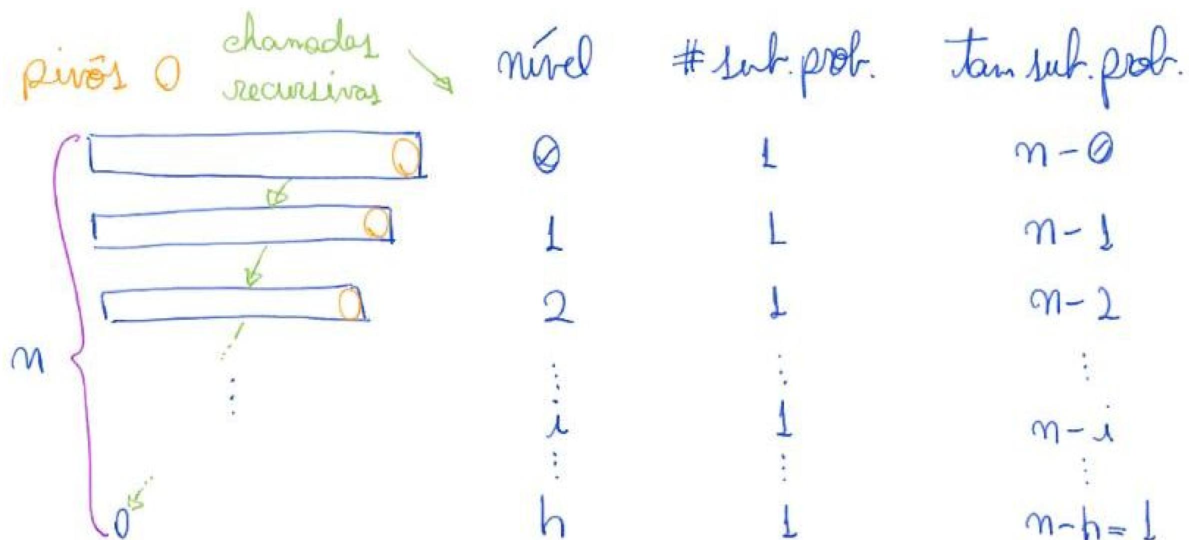


- Pior caso:
    - pivô sempre é o menor ou maior elemento do vetor

        - 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 não trivial será apenas uma unidade menor que o anterior.
        - Assim, teremos 1 subproblema por nível.
        - O tamanho do subproblema no nível l será n - l
        - Por isso o trabalho no nível l será const * (n - l).
        - O total de níveis será n,
            - já que no último nível h temos
                - tamanho do subproblema = 1 = (n - h) ⇒ h = n - 1,
            - e começamos a contar os níveis em 0.
        - Assim, o trabalho total será

const * [n + (n - 1) + (n - 2) + … + 2 + 1] =

= const * n * (n+1) / 2 ~= const * (n^2) / 2 = O(n^2).



- Caso médio:
    - quando lidando com vetores que são permutações aleatórias,
        - a ordem do número de operações fica próxima do melhor caso,
            - i.e., O(n lg n).
    - 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 das escolhas aleatórias dele,
            - e não mais da configuração do vetor de entrada.

Código quicksort recursivo aleatorizado:

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

```c
// p indica a primeira posicao e r a ultima
void quicksortRA(int v[], int p, int r)
{
   int desl, j;
   if (p < r)
   {
       // desl = rand() % (r - p + 1);
       desl = (int)(((double)rand() / (RAND_MAX + 1)) * (double)(r - p + 1));
       // printf("p = %d, r = %d, r-p+1 = %d, desl = %d\n", p, r, r - p + 1, desl);
       troca(&v[p + desl], &v[r]);
       j = separa1(v, p, r);
       quicksortRA(v, p, j - 1);
       quicksortRA(v, j + 1, r);
   }
}
```

Funções de aleatorização:

- a função rand(),
  - definida na biblioteca stdlib,
  - gera um número pseudo-aleatório
    - no intervalo fechado 0 .. RAND_MAX.
- Primeira opção

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

  - Obtém um número inteiro no intervalo [0, r - p],
    - pegando o resto da divisão de um inteiro aleatório por (r - p + 1).
  - No entanto, possui um viés que privilegia números pequenos.
- Segunda opção

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

  - Transforma o inteiro aleatório, obtido de rand(), em um número real
    - no intervalo [0, 1)

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

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

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

  - Então, transforma esse real num inteiro
    - no intervalo [0, r - p]

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

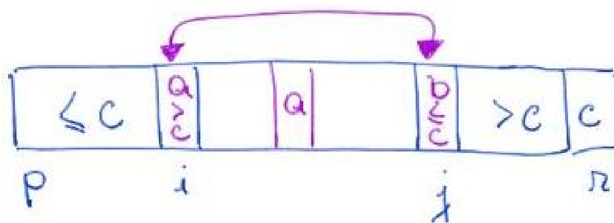
Eficiência de tempo esperada do quicksort aleatorizado:

- 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,
    - uma divide o vetor próximo da metade.
    - É um raciocínio 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.



Posição relativa dos dois elementos a é invertida

Eficiência de espaço:

- 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
    - e é armazenada na pilha de execução.
- Assim, quicksort ocupa memória adicional
    - proporcional à altura da pilha de execução,
        - que chega à altura (número de níveis) das árvores de recursão em nossas análises,
            - i.e., n no pior caso e lg n + 1 no melhor caso.
- Portanto, o uso de memória cresce de acordo com o tamanho da entrada.
    - Por isso podemos dizer que 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.
- Uma alternativa para garantir que o quicksort,
    - tanto na versão determinística quanto na probabilística,
    - não chegue a produzir uma pilha de execução maior que lg n é
        - sempre fazer a primeira chamada recursiva no menor subvetor
            - que terá tamanho <= que metade do vetor anterior

        - e substituir a segunda chamada recursiva
            - por uma versão iterativa.
        - 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.
    - 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 para um algoritmo iterativo.

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

```
void quicksortRP(int v[], int p, int r)
{
   int j;
   while (p < r)
   {
       j = separa1(v, p, r);
       if (j - p < r - j)
       { // ordena recursivamente o subvetor esquerdo
           quicksortRP(v, p, j - 1);
           p = j + 1;
       }
       else
       { // ordena recursivamente o subvetor direito
           quicksortRP(v, j + 1, r);
           r = j - 1;
       }
   }
}
```

Animação:

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