Bubble Sort

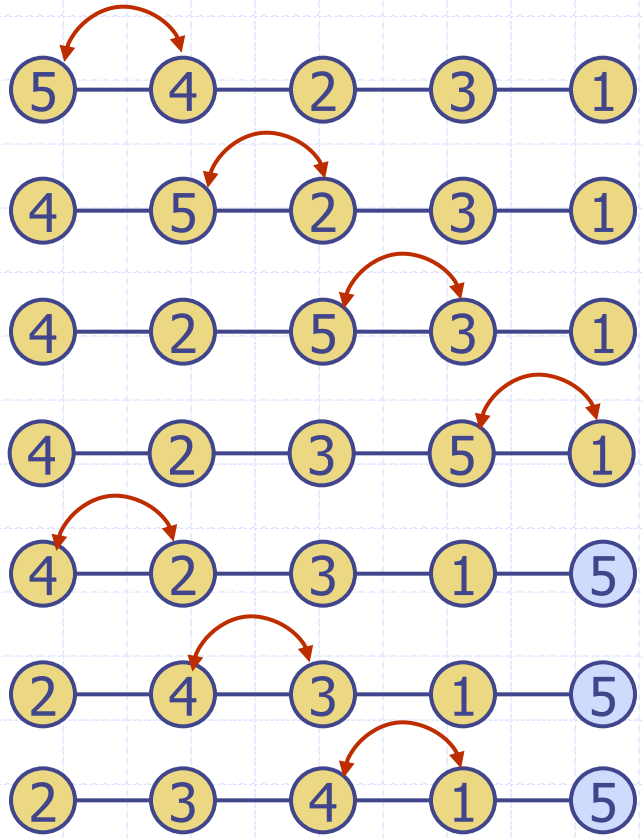
- Considere uma seqüência de n elementos que se deseja ordenar. O método da bolha resolve esse problema através de várias passagens sobre a seqüência
- Não é um algoritmo eficiente, é estudado para fins de desenvolvimento de raciocínio
- funcionamento:
  - Na primeira passagem, uma vez encontrado o maior elemento, este terá sua colocação trocada até atingir a última posição
  - Na segunda passagem, uma vez encontrado o segundo maior elemento, este terá sua colocação trocada até atingir a penúltima posição
  - E assim por diante

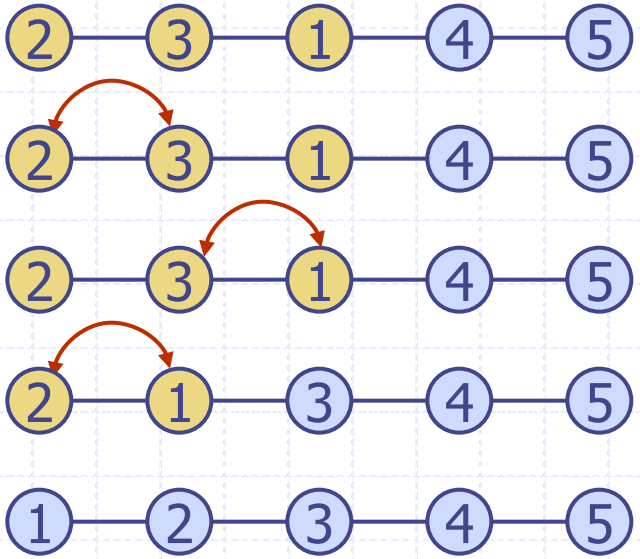
- Tempo total $O(n^2)$

```
Algorithm bubbleSort(A)
    Input array A com n elementos
    Output array A ordenado
    for i=0 até n-2
        for j=0 até n-2-i
            if (A[j] > A[j+1] )
                aux ← A[j]
                A[j] ← A[j+1]
                A[j+1] ← aux
```

Bubble Sort

- Exemplo





Select Sort

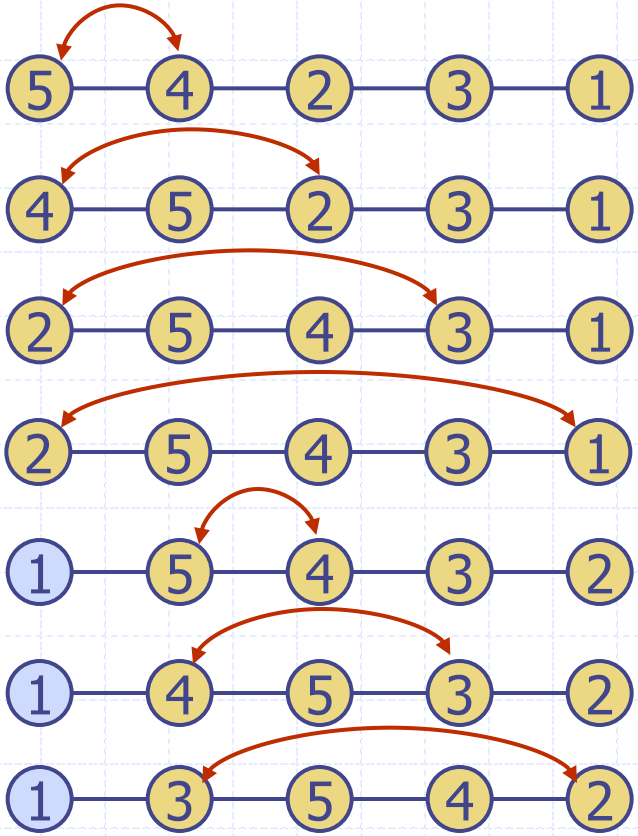
- O método de ordenação por Seleção Direta é levemente mais eficiente que o método Bubblesort
- Trata-se de um algoritmo apenas para estudo e ordenação de pequenos arranjos
- funcionamento:
  - Varre-se o arranjo comparando todos os seus elementos com o primeiro.
  - Caso o primeiro elemento esteja desordenado em relação ao elemento que está sendo comparado com ele no momento, é feita a troca.
  - Ao se chegar ao final do arranjo, teremos o menor valor (ou o maior, conforme a comparação) na primeira posição do arranjo

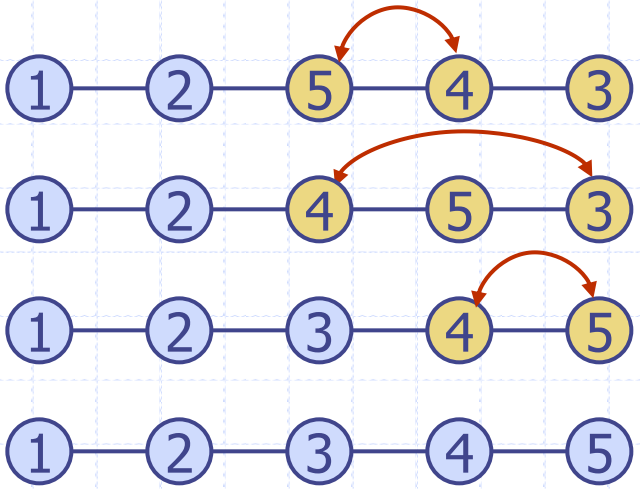
- Tempo total $O(n^2)$

```
Algorithm selectSort(A)
    Input array A com n elementos
    Output array A ordenado
    for i=0 até n-2
        for j=i+1 até n-1
            if (A[j] < A[i] )
                aux ← A[j]
                A[j] ← A[i]
                A[i] ← aux
```

Select Sort

- Exemplo





Insert Sort

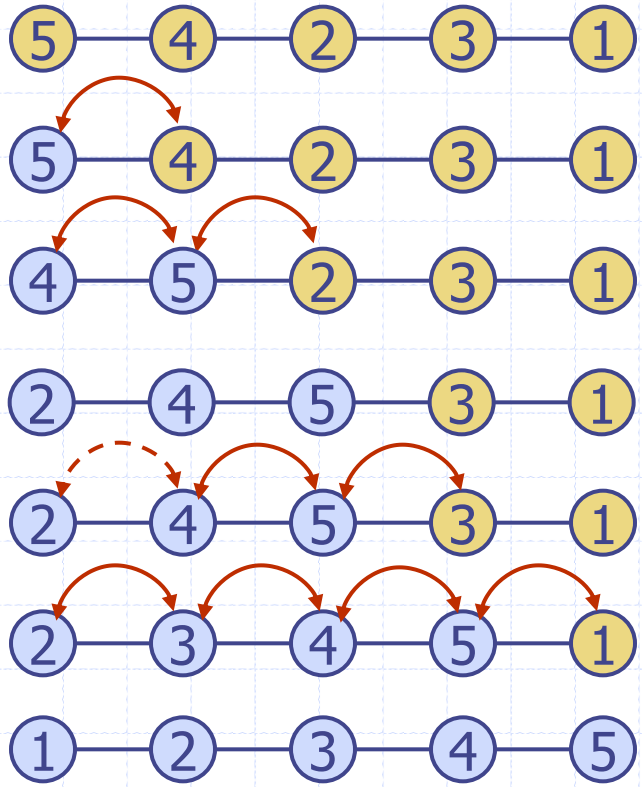
- O método de ordenação por Inserção Direta é o mais rápido entre os outros métodos considerados básicos (Bubblesort e Seleção Direta)
- A principal característica deste método consiste em ordenarmos nosso arranjo utilizando um sub-arranjo ordenado localizado em seu inicio.
- A cada novo passo, acrescentamos a este sub-arranjo mais um elemento, até que atingimos o último elemento do arranjo fazendo assim com que ele se torne ordenado

- Tempo total $O(n^2)$

Algorithm *insertSort*(*A*)
  Input *array* **A** com **n** elementos
  Output *array* **A** ordenado
  for *i=1 até n-1*
    aux ← *A[i]*
    j ← *i* – *1*
    while (*j*>=*0* *e* *aux* < *A*[*j*])
      A[j+1] ← *A[j]*
      j ← *j* – *1*
    A[j+1] ← *aux*

Insert Sort

- Exemplo



Heap-Sort

- Considere uma fila de prioridade com n itens implementado com um heap
  - O espaço usado é $O(n)$
  - métodos insert e removeMin rodam em tempo $O(\log n)$
  - métodos size, isEmpty, e min rodam em tempo $O(1)$
- Usando uma fila de prioridade baseada em heap, podemos ordenar uma sequência de n elementos em tempo $O(n \log n)$
- O algoritmo é chamado de *heap-sort*
- *heap-sort* é muito mais rápido do que algoritmos quadráticos, como inserção e seleção

Divisão e Conquista

- Divisão e Conquista é um paradigma de desenvolvimento de algoritmo:
  - Divisão: divida a entrada S em dois conjuntos disjuntos S_1 and S_2
  - Recursão: solucione os problemas associados com S_1 e S_2
  - Conquista: Combine as soluções para S_1 **e** S_2 dentro da solução S
- O caso base para a recursão são problemas de tamanho 0 ou 1
- Merge-sort é um algoritmo baseado no paradigma divisão e conquista
- Como o heap-sort
  - Ele usa um comparador
  - O tempo é $O(n \log n)$
- Diferente heap-sort
  - Não usa uma fila de prioridade auxiliar
  - Ele acessa os dados de forma sequencial

Merge-Sort

- Merge-sort com uma seqüência de entrada S com n elementos consiste de três passos:
  - Divide: dividir S em duas sequencias S_1 and S_2 de aproximadamente $n/2$ elementos cada
  - Recursão: recursivamente ordene S_1 e S_2
  - Conquista: junte S_1 *e* S_2 em uma única sequência ordenada

Algorithm *mergeSort*(S, C)

Input sequence S with n elements, comparator C

Output sequence S sorted according to C

if $S.size() > 1$

(S_1, S_2) ← *partition*(S, $n/2$)

mergeSort(S_1, C)

mergeSort(S_2, C)

S ← *merge*(S_1, S_2)

Juntando Duas Sequencias Ordenadas

- O passo de conquista do merge-sort consiste de juntar duas sequências ordenadas A e B em uma sequência S contendo a união dos elementos de A e B
- Unindo duas sequências ordenadas, cada uma com $n/2$ elementos e implementado por uma lista duplamente encadeada leva o tempo $O(n)$

Algorithm *merge*(*A*, *B*)

Input sequences *A* and *B* with
n/2 elements each

Output sorted sequence of $A \cup B$

S ← empty sequence

while ¬*A.isEmpty*() ∧ ¬*B.isEmpty*()
if *A.first*().*element*() < *B.first*().*element*()
S.insertLast(*A.remove*(*A.first*()))
else
S.insertLast(*B.remove*(*B.first*()))

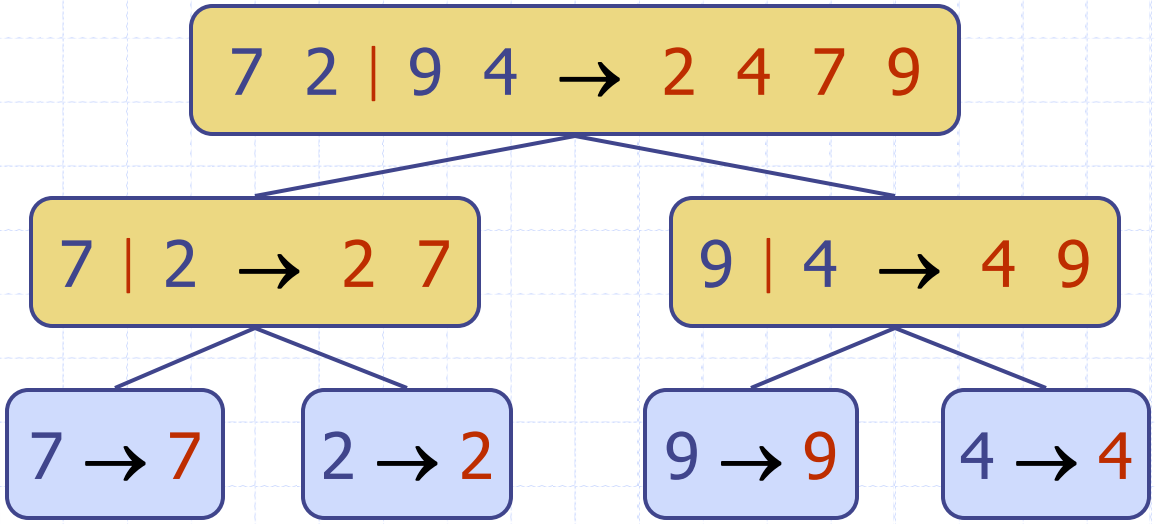
while ¬*A.isEmpty*()
S.insertLast(*A.remove*(*A.first*()))

while ¬*B.isEmpty*()
S.insertLast(*B.remove*(*B.first*()))

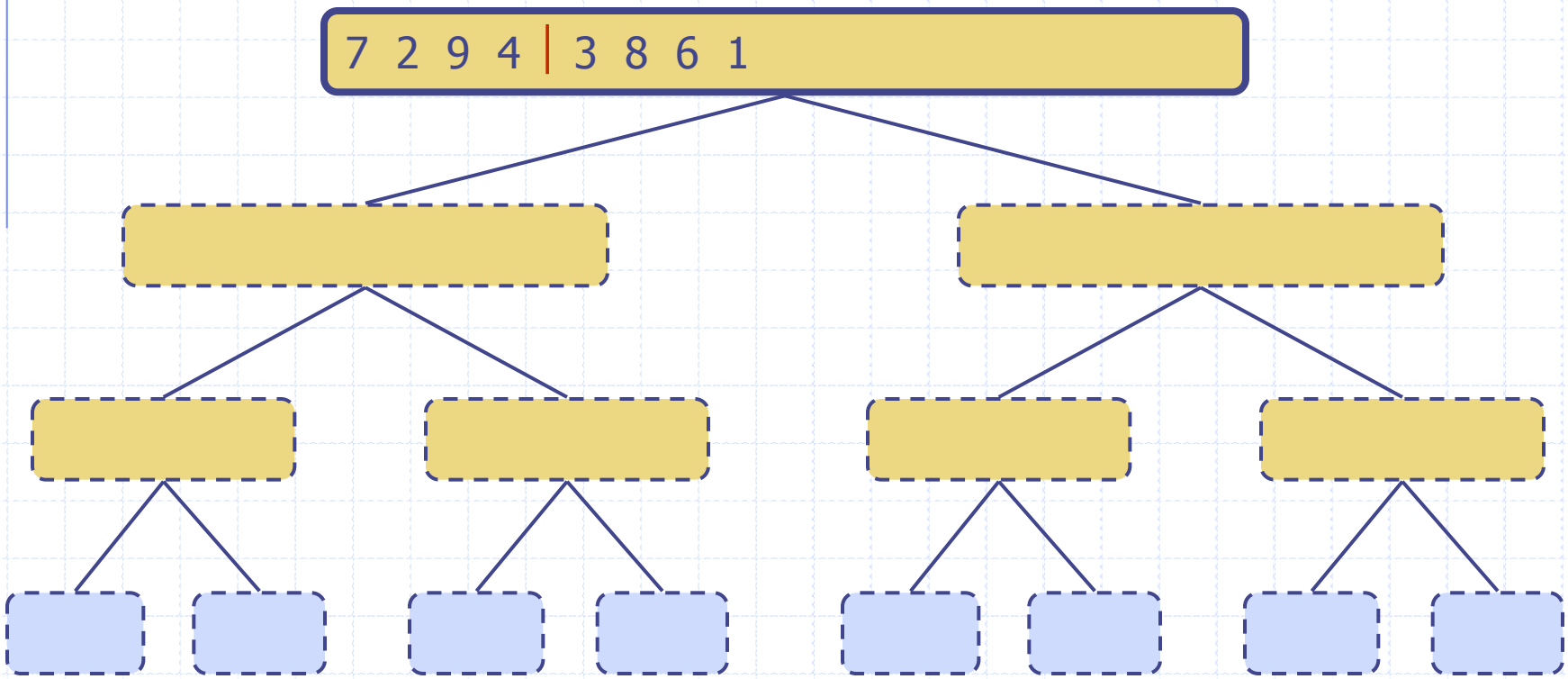
return *S*

Árvore Merge-Sort

- Uma execução do merge-sort pode ser vista como uma árvore binária
  - Cada nó representa uma chamada recursiva do merge-sort e armazena
    - Sequências desordenadas antes da execução e suas partições
    - Sequências ordenadas no fim da execução
  - A raiz é a chamada inicial
  - As folhas são chamadas de subseqüências de tamanho 0 ou 1



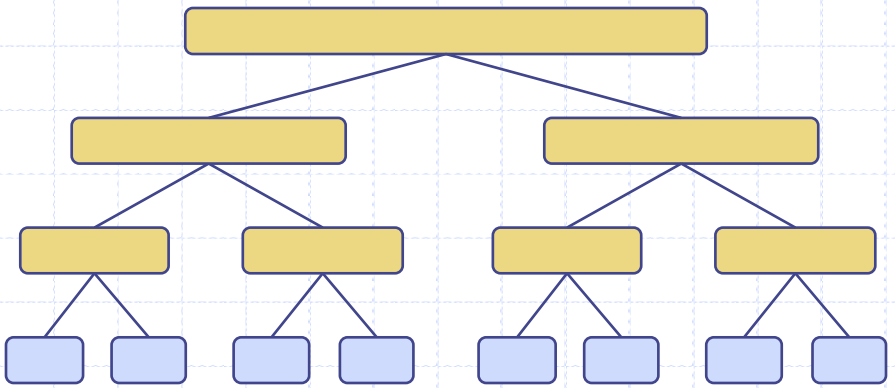
Exemplo



Análise do Merge-Sort

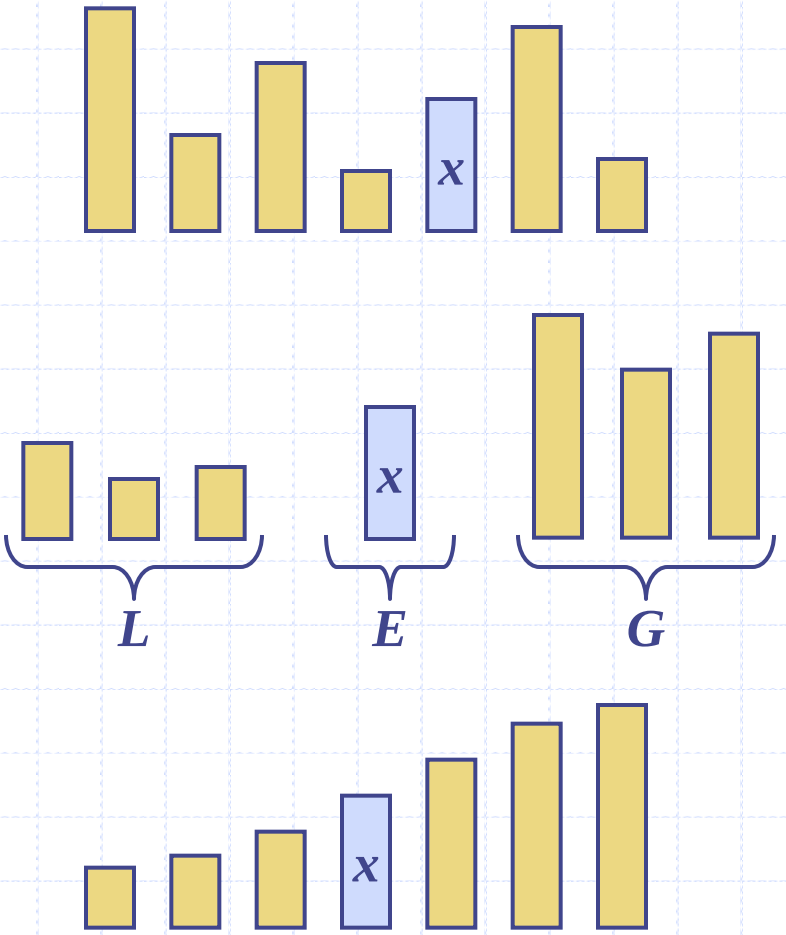
- A altura h da árvore merge-sort é $O(\log n)$
  - Em cada chamada recursiva a sequência é dividida pela metade
- A quantidade de trabalho no nó de profundidade i é $O(n)$
  - Nós particionamos e juntamos 2^i seqüências de tamanho $n/2^i$
  - Nós fazemos 2^{i+1} chamadas recursivas
- Assim, o tempo do merge-sort é $O(n \log n)$

depth	#seqs	size
0	1	n
1	2	$n/2$
i	2^i	$n/2^i$
…	…	…



Quick-Sort

- Quick-sort é um algoritmo aleatório baseado no paradigma de divisão e conquista
- paradigma:
  - Divisão: pegue um elemento x aleatório (chamado pivô) e particione S em
    - L elementos menor que x
    - E elementos igual a x
    - G elementos maiores que x
  - Recursão: ordene L e G
  - Conquista: junte L, E e G



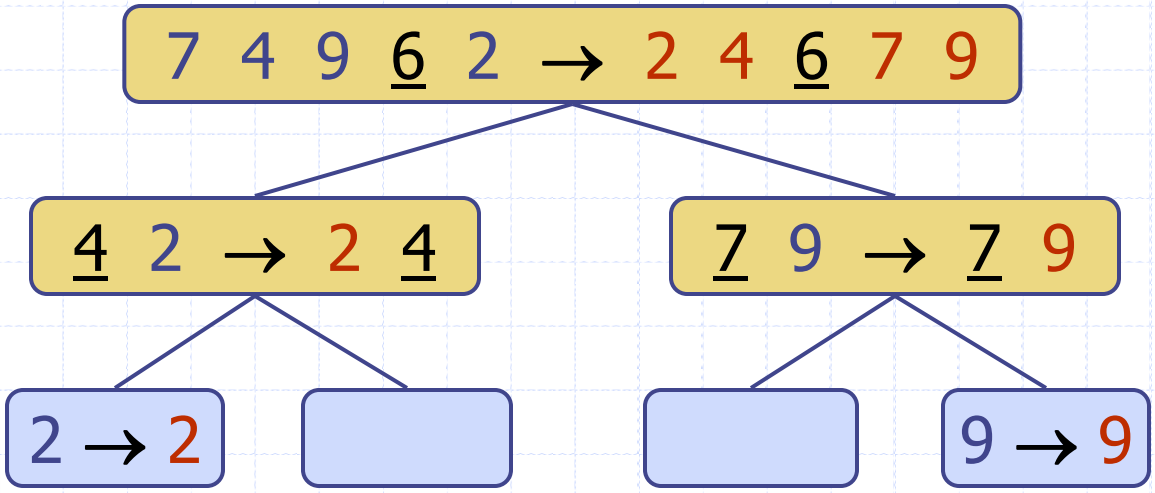
Partição

- Particiona-se a sequência de entrada da seguinte forma:
  - Remover cada elemento y de S e
  - Inserer y em L, E or G, dependendo do resultado da comparação com o pivô x
- Cada inserção e remoção é feita no início ou fim da sequênciais, e leva o tempo $O(1)$
- A partição do quick-sort leva um tempo proporcional a $O(n)$

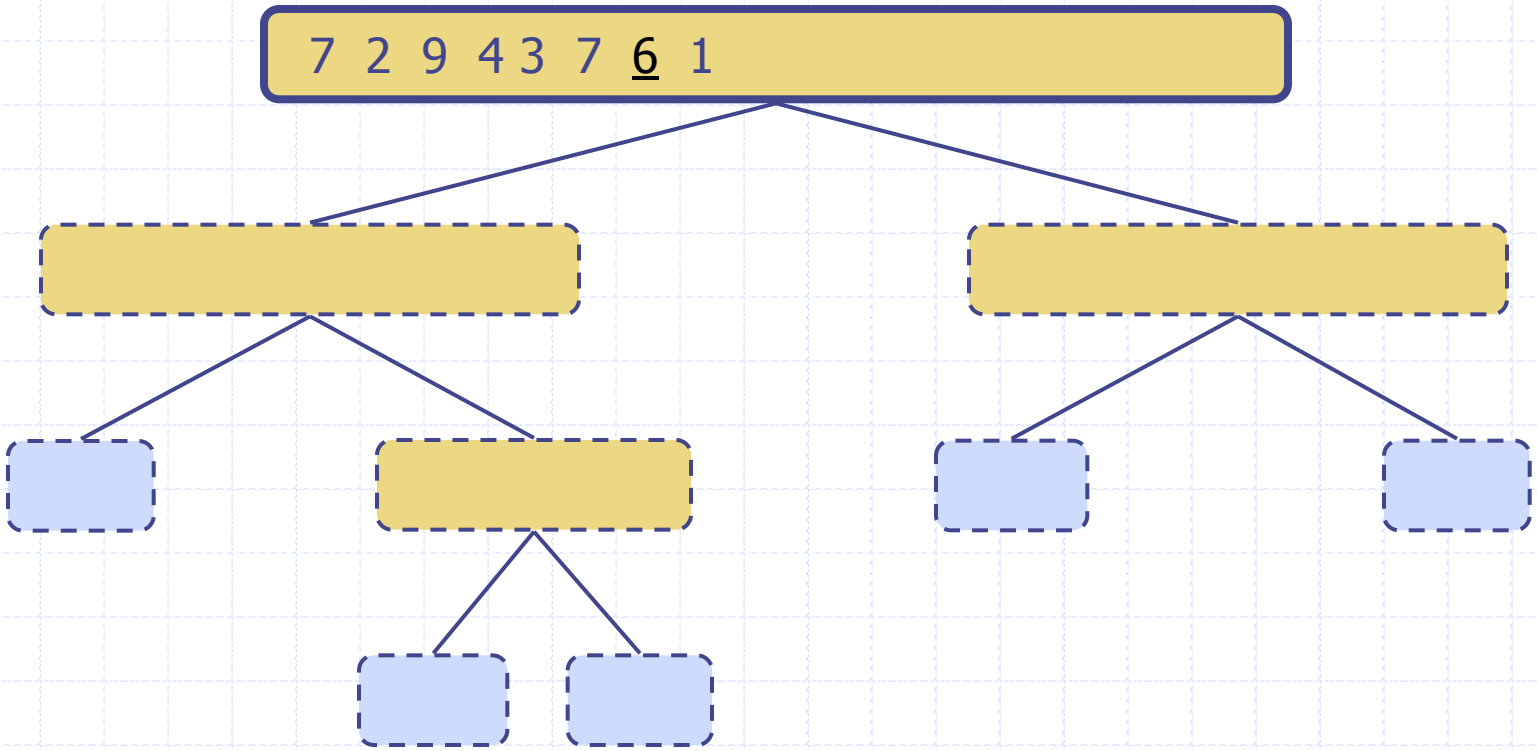
```
Algorithm partition(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the
        elements of S less than, equal to,
        or greater than the pivot, resp.
    L, E, G ← empty sequences
    x ← S.remove(p)
    while ¬S.isEmpty()
        y ← S.remove(S.first())
        if y < x
            L.insertLast(y)
        else if y = x
            E.insertLast(y)
        else { y > x }
            G.insertLast(y)
    return L, E, G
```

Árvore Quick-Sort

- Uma execução do quick-sort pode ser vista como uma árvore binária
  - Cada nó representa uma chamada recursiva do quick-sort e armazena
    - Sequencia desordenada antes da execução e seu pivô
    - Sequência ordenada no final da execução
  - A raiz é a chamada inicial
  - As folhas são chamadas de subsequências de tamanho 0 ou 1



Exemplo

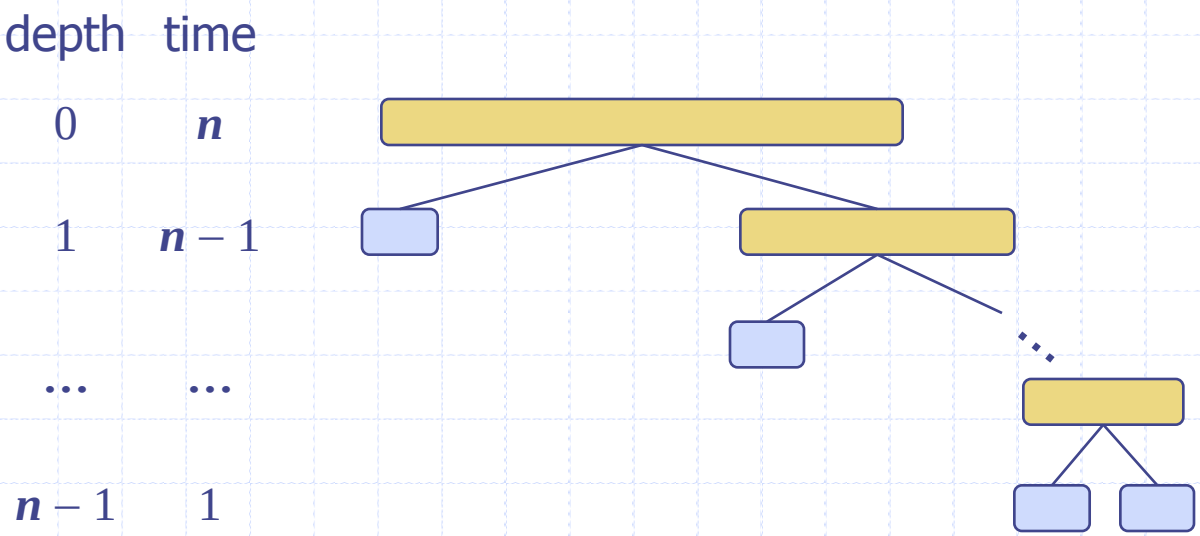


Tempo de Excução no Pior Caso

- O pior caso para o quick-sort ocorre quando o pivô é estritamente o elemento mínimo or máximo
- Um deles L ou G tem tamanho $n-1$ e o outro tem tamanho 0
- O tempo é proporcional a soma

$$n + (n-1) + \ldots + 2 + 1$$

- Assim, o pior caso do quick-sort é $O(n^2)$
- O tempo esperado do quick-sort randomizado é $O(n \log n)$



Resumo dos Algoritmos de Ordenação

Algoritmo	Tempo	OBS
Bubble-sort	$O(n^2)$	◆ lento (bom para pequenas entradas)
selection-sort	$O(n^2)$	◆ lento (bom para pequenas entradas)
insertion-sort	$O(n^2)$	◆ lento (bom para pequenas entradas)
quick-sort	$O(n \log n)$ esperado	◆ randomizado ◆ muito rápido (bom para grandes entradas)
heap-sort	$O(n \log n)$	◆ rápido (bom para grandes entradas)
merge-sort	$O(n \log n)$	◆ rápido (bom para entradas muito grandes)