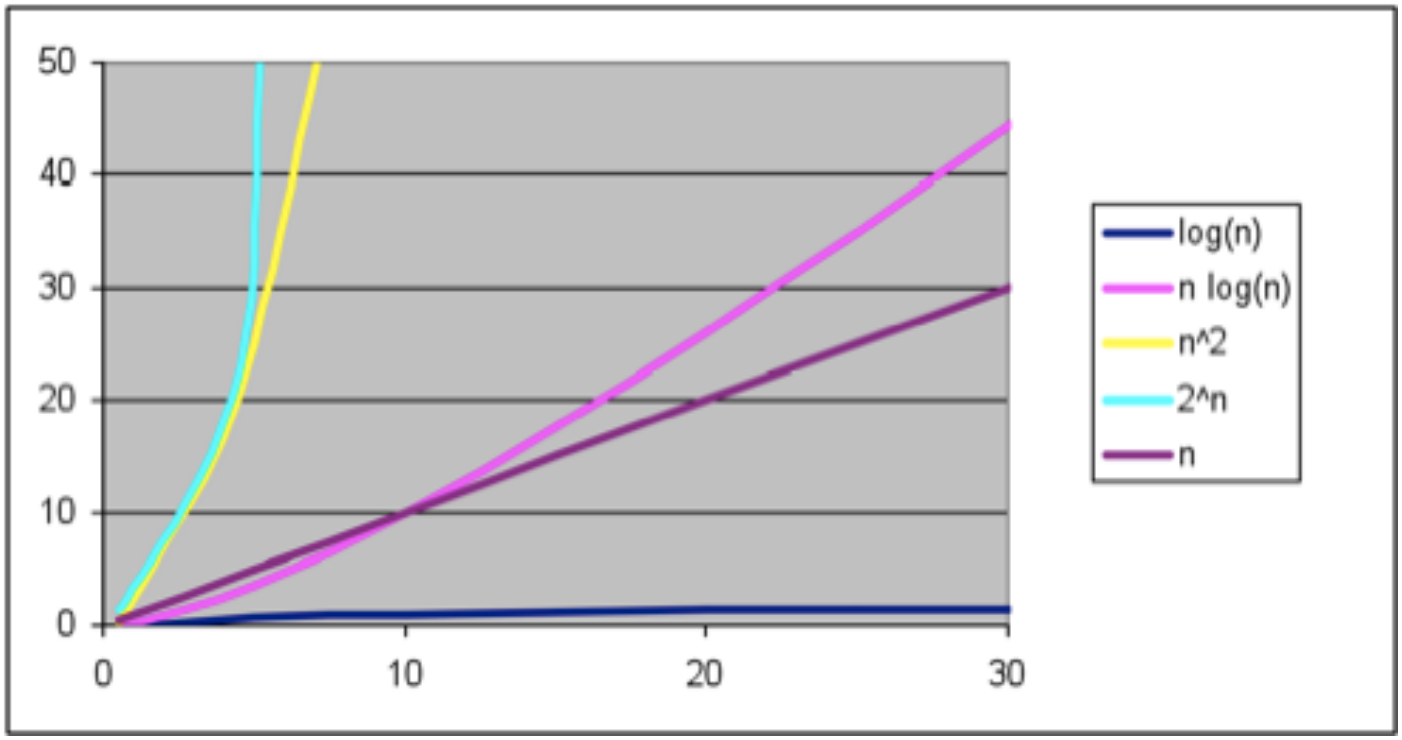
Complexidade de Algoritmos

- Frequentemente, desejamos saber quão **eficiente** é um determinado algoritmo usado para resolver um problema, ou seja, quanto tempo (ou espaço de memória) será necessário para a **execução** do algoritmo.
- Isto é chamado de **complexidade** de tempo (ou de espaço) do algoritmo. Tipicamente, a complexidade de um algoritmo é medida em função dos valores de seus dados de entrada.
- **Complexidade assintótica** (complexidade para entradas bem grandes): Notação de ordem
  - Seja N o conjunto dos inteiros não-negativos. Sejam f e g funções de N para N.
  - Escrevemos f(n) = O(g(n)) – lê-se "f é **ordem** g" – se existem inteiros positivos c e n_0 tais que, para todo $n \geq n_0$, $f(n) \leq c.g(n)$.

Complexidade de Algoritmos

- Informalmente, f(n) = O(g(n)) significa que f **cresce mais lentamente** do que g (ou, no máximo, tanto quanto g).

- Algumas funções bem conhecidas:



Complexidade de Algoritmos

- **Exemplos**
  - Seja f(n) um polinômio de grau k. Então f(n) = $O(n^k)$, ou seja, a taxa de crescimento de um polinômio é capturada pelo seu termo de maior grau.
  - Seja g(n) = c^n, para algum c > 1. Então, qualquer se seja o valor de k, f(n) cresce mais lentamente do que g(n) porque, para valores de n suficientemente grandes, f(n) ≤ g(n).

    Para ilustrar:

n	n^{50}	2^n
100	1,0E+100	1,3E+30
200	1,1E+115	1,6E+60
300	7,2E+123	2,0E+90
400	1,3E+130	2,6E+120
500	8,9E+134	3,3E+150

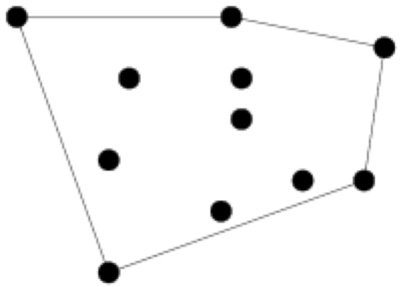
Algoritmos de Ordenação

Exemplos motivadores

- **Busca**: *Existe um valor igual a x em um vetor v?* Se o vetor estiver desordenado, é preciso comparar x com todos os n elementos do vetor. Assim, o algoritmo terá complexidade assintótica O(n). Com o vetor ordenado, um algoritmo de **busca binária** resolve o problema em O(log n).
- **Par mais próximo**: *Dados n números, encontre o par de números que estão mais próximos um do outro*. Se os números estiverem ordenados, os números do par mais próximo serão vizinhos e, portanto, um algoritmo O(n) resolve o problema.
- **Elemento único**: *Dado um vetor de tamanho n, todos os elementos são únicos ou existem elementos duplicados?* Este é um caso especial do problema do par mais próximo e, portanto, pode ser resolvido com um algoritmo O(n).

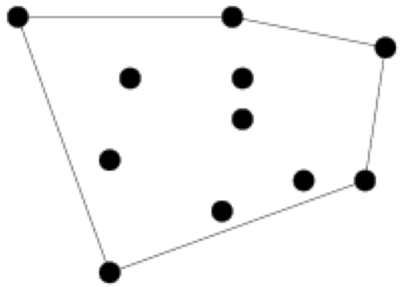
Algoritmos de Ordenação

- **Frequência de distribuição (moda)**: *Dado um vetor de n elementos, qual elemento ocorre mais vezes no vetor?* Com o vetor ordenado, uma varredura linear O(n) pode contar o número de elementos iguais adjacentes.
- **Seleção**: *Dado um vetor com n elementos, qual é o k-ésimo maior elemento do vetor?* Ordenado o vetor, o k-ésimo maior elemento pode ser encontrado em tempo constante simplesmente procurando-se na k-ésima posição do vetor.
- **Envoltória convexa**: *Dados n pontos no plano, encontrar o polígono de menor área que contém todos os pontos*. Por exemplo:



Algoritmos de Ordenação

- Ordenados pela coordenada-x, os pontos podem ser inseridos na envoltória da esquerda para a direita, pois o ponto mais à direita sempre vai estar na envoltória.



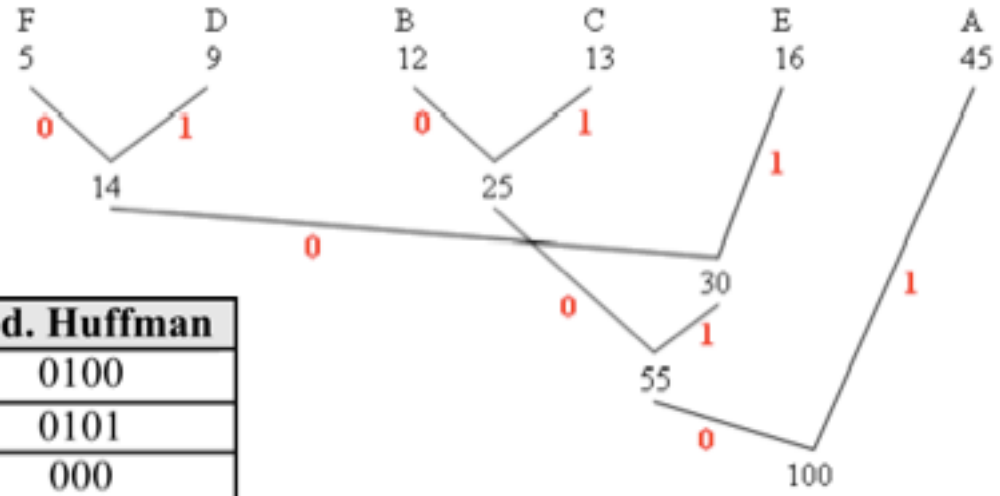
- Ao inserir um novo ponto mais à direita na envoltória, pontos inseridos anteriormente podem deixar de estar na nova envoltória. Neste caso, um ponto que sai não irá mais pertencer à envoltória.
- Sem a ordenação, é preciso verificar, a cada vez, se o novo ponto está dentro ou fora da atual envoltória, podendo entrar e sair da envoltória diversas vezes.

Algoritmos de Ordenação

- **Código de Huffman**: Para minimizar a quantidade de espaço consumido por um texto, podemos codificar as letras do alfabeto por um código cujo tamanho depende da frequência com que as letras aparecem no texto. Assim, letras mais frequentes podem ter uma codificação mais curta e letras menos frequentes, uma codificação mais longa.
- Exemplo:

Letra	% Ocorrência
F	5
D	9
B	12
C	13
E	16
A	45

Letra	Cód. Huffman
F	0100
D	0101
B	000
C	001
E	011
A	1



Algoritmo *BubbleSort*

- Ideia: percorrer o vetor sequencialmente e comparar cada elemento com seu sucessor, efetuando uma troca se esses elementos não estiverem na ordem correta.

```
void Troca(int *x, int *y)
{
  int t;

  t = *x;
  *x = *y;
  *y = t;
}

void BubbleSort(int v[], int n)
{
  int k;
  int limite = n-1;
  int ultima_troca;

  while (limite > 0)
  {
    ultima_troca = 0;
    for (k = 0; k < limite; k++)
    {
      if (v[k] > v[k+1])
      {
        Troca(&v[k],&v[k+1]);
        ultima_troca = k;
      }
    }
    limite = ultima_troca;
  }
}
```

Uma das principais vantagens do algoritmo *BubbleSort* é sua simplicidade. O algoritmo tem complexidade assintótica $O(n^2)$ e funciona muito bem em vetores quase ordenados.

Ordenação por Seleção

- Este algoritmo usa um marcador para dividir as partes ordenada (à esquerda) e desordenada (à direita) do vetor. Procura-se na parte desordenada pelo menor elemento e troca-se esse elemento com o elemento sob o marcador. Em seguida, avança-se o marcador. O processo se repete até que exista apenas um elemento após o marcador.

```
void OrdenacaoSelecao(int v[], int n)
{
  int k;
  int menor,posicao,marcador;

  marcador = 0;
  while (marcador < n-1)
  {
    menor = v[marcador];
    posicao = marcador;
    for (k = marcador+1; k < n; k++)
    {
      if (v[k] < menor)
      {
        menor = v[k];
        posicao = k;
      }
    }
    Troca(&v[marcador],&v[posicao]);
    marcador++;
  }
}
```

A análise deste algoritmo é simples. Na primeira iteração o algoritmo faz (n-1) comparações. Na segunda iteração, faz (n-2) comparações, e assim por diante. Portanto, no total o algoritmo faz:

$$(n - 1) + (n - 2) + \ldots + 1 = n \times (n - 1)/2$$

comparações, e portanto o algoritmo tem complexidade $O(n^2)$.

Ordenação por Inserção

- Também usa um marcador. Neste algoritmo, inicialmente, a parte ordenada contém 1 elemento (marcador = 1). Seja *x* o primeiro elemento da parte desordenada. Troca-se *x* de posição com os elementos que aparecem à esquerda até que *x* esteja em sua posição correta, e avança-se o marcador. O processo se repete até que a parte desordenada do vetor esteja vazia.

```
void OrdenacaoInsercao(int v[], int n)
{
  int k;
  int x,marcador;

  for (marcador = 1; marcador < n; marcador++)
  {
    x = v[marcador];
    k = marcador-1;
    while ((k >= 0) && (v[k] > x))
    {
      v[k+1] = v[k];
      k--;
    }
    v[k+1] = x;
  }
}
```

No começo de cada iteração do comando **for** deste algoritmo, os elementos que aparecem nas posições de **0** até **marcador-1** são os elementos contidos originalmente no vetor, mas na ordem correta. Esta propriedade é conhecida como **invariante** e ajuda a mostrar que um algoritmo iterativo é correto.

Ordenação por Inserção

- Um **invariante** é uma relação entre os valores das variáveis que vale no início de cada iteração de um processo iterativo.
- Usando invariantes é possível provar (por um processo que se assemelha à prova por indução) que um algoritmo iterativo é correto. O processo para isto consiste em mostrar que:
  a) O invariante é verdadeiro antes da primeira iteração;
  b) Se o invariante é verdadeiro antes de uma iteração, este invariante continuará verdadeiro antes da próxima iteração;
  c) Ao final do laço o invariante corresponde a uma propriedade útil para mostrar que o algoritmo é correto.

Ordenação por Inserção

Aplicando para o algoritmo de ordenação por inserção:

a) Antes da primeira iteração, **marcador = 1**. Portanto, a parte ordenada (de 0 até marcador-1) contém apenas v[0]. Portanto, o invariante é válido antes da primeira iteração.

b) Informalmente, o laço do **for** consiste em deslocar os elementos v[marcador-1], v[marcador-2], ... uma posição para a direita até encontrar a posição para *x* = v[marcador], e então inserir *x* em sua posição correta. Portanto, ao final do laço o invariante continuará verdadeiro antes da próxima iteração (uma demonstração formal exigiria identificar um invariante para o laço mais interno, estabelecido pelo comando **while**).

c) O **for** termina quando **marcador = n**. Então, o invariante indica que os elementos de **0** até **n-1** estão em suas posições corretas, ou seja, o algoritmo é correto.

Ordenação por Inserção

- O algoritmo de ordenação por inserção é bem eficiente para ordenar um vetor com pequeno número de elementos. O tempo de execução do algoritmo depende de como estão os elementos do vetor. O melhor caso ocorre quando o vetor já está ordenado. Neste caso, não será necessário deslocar elementos no laço interno e o tempo de execução é O(n). No pior caso, os elementos do vetor estão ordenados em ordem inversa. Neste caso, na primeira iteração será necessário deslocar 2 elementos, na segunda iteração 3, e assim por diante. Neste caso, o tempo de execução será:

$$T(n) = \sum_{j=1}^{n-1} (j+1) = 2 + 3 + \cdots + n = \frac{n(n-1)}{2} - 1 = O(n^2)$$

Algoritmo *MergeSort*

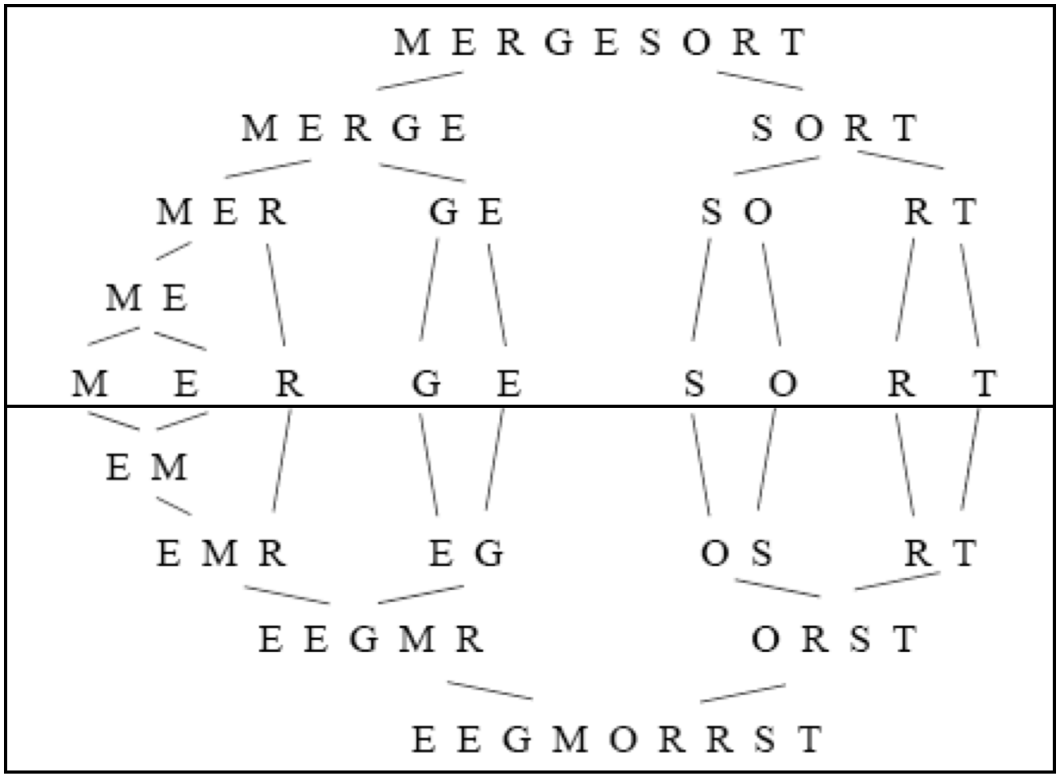
- Alguns algoritmos de ordenação baseiam-se na técnica de "Dividir e Conquistar". A ideia é reduzir um problema grande em problemas menores, resolver cada um destes subproblemas recursivamente, e combinar as soluções parciais para obter a solução do problema original.
- No algoritmo *MergeSort* a combinação das soluções parciais (vetores menores já ordenados) consiste em intercalar os elementos dos dois vetores ordenados para obter a ordenação total dos elementos.

```
void MergeSort(int v[], int inicio, int fim)
{
  int meio;

  if (inicio < fim)
  {
    meio = (inicio + fim)/2;
    MergeSort(v,inicio,meio);
    MergeSort(v,meio+1,fim);
    Merge(v,inicio,meio,fim);
  }
}
```

Algoritmo *MergeSort*

- Exemplo:



Fase de divisão

Fase de intercalação (merge)

- A eficiência do algoritmo depende de quão eficientemente será a intercalação dos dois vetores (ordenados) em um único vetor ordenado.

Algoritmo *MergeSort*

- A intercalação pode ser feita removendo-se o menor elemento dos dois vetores e incluindo-se este elemento no vetor final. Por exemplo:

Vetores iniciais		Vetor final
5, 7, 12, 19	4, 6, 13, 15	4
5, 7, 12, 19	6, 13, 15	4, 5
7, 12, 19	6, 13, 15	4, 5, 6
7, 12, 19	13, 15	4, 5, 6, 7
12, 19	13, 15	4, 5, 6, 7, 12
19	13, 15	4, 5, 6, 7, 12, 13
19	15	4, 5, 6, 7, 12, 13, 15
19		4, 5, 6, 7, 12, 13, 15, 19

Note que o algoritmo **MergeSort** exige também O(n) espaço adicional para o processo de intercalação.

- Portanto, a intercalação pode ser feita fazendo-se, no máximo, (n – 1) comparações, onde n é o número total de elementos dos dois vetores originais, ou seja, o algoritmo de intercalação é O(n).

Algoritmo *MergeSort*

- O tempo de execução de algoritmos recursivos pode ser descrito por uma **recorrência**. Uma recorrência é uma equação ou desigualdade que descreve uma função em termos de seu valor para entradas menores.
- A equação de recorrência do *MergeSort* é:

$$T(n) = 2T(n/2) + O(n)$$

onde:

  - T(n) representa o tempo de execução do algoritmo para um problema de tamanho n;
  - 2T(n/2) indica que, a cada execução, duas chamadas recursivas (2T) serão executadas para entradas de tamanho (aproximadamente) n/2;
  - O(n) indica que os resultados das chamadas recursivas serão intercalados por um algoritmo O(n).

Algoritmo *MergeSort*

- Formalmente, a equação de recorrência do *MergeSort* é:

$$T(n) = \begin{cases} O(1) & se\ n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n) & se\ n > 1 \end{cases}$$

- Para resolver uma recorrência podemos usar o método da **árvore de recursão**. Numa árvore de recursão cada nó representa o custo de um único subproblema da respectiva chamada recursiva. Somam-se os custos de todos os nós de um mesmo nível da árvore, para obter o custo daquele nível. Finalmente, somam-se os custos de todos os níveis para obter o custo da árvore.

Algoritmo *MergeSort*

- No caso do algoritmo *MergeSort* temos:
  - Quantos níveis tem á árvore de recursão (fase de divisão do algoritmo)?
    $\log_2(n) + 1$
  - Quantos nós tem cada nível *i* da árvore?
    2^i
  - Qual o tamanho do problema em cada nível *i* da árvore?
    $(n / 2^i)$
  - Qual é o custo de cada nível *i* da árvore?
    $2^i \times O(n / 2^i) = O(n)$
  - Qual é o custo da árvore?

$$\sum_{i=0}^{\log_2(n)} O(n) = \underbrace{O(n) + \cdots + O(n)}_{\log_2(n)+1 \ \ vezes} = (\log_2(n)+1) \times O(n) = O(n \times \log n)$$

Algoritmo *QuickSort*

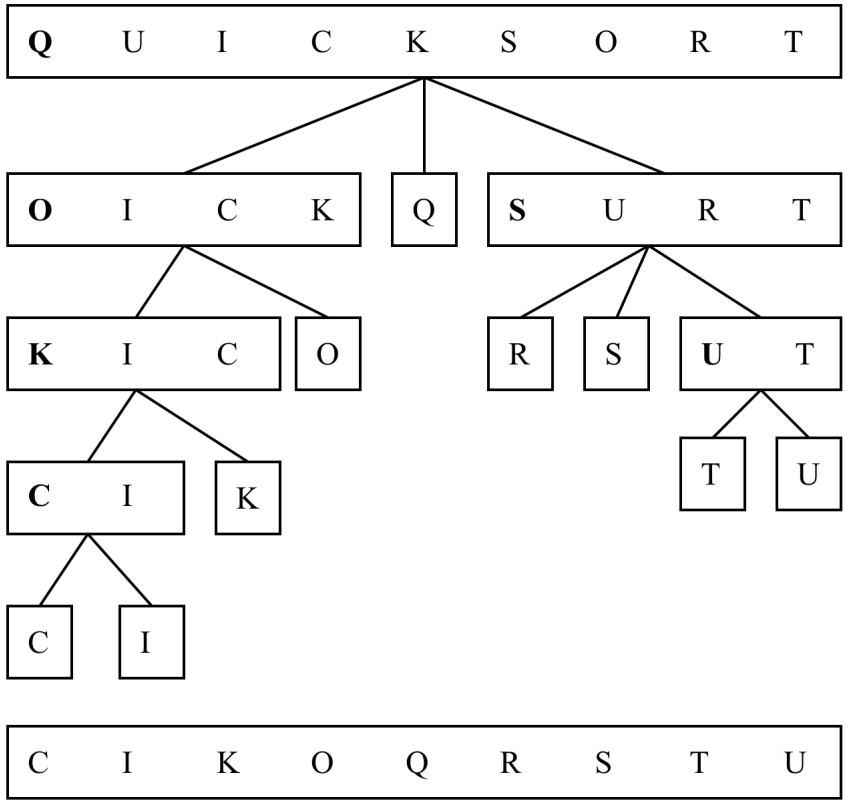
- *QuickSort* é outro algoritmo de ordenação que usa a técnica “Dividir e Conquistar”. O algoritmo baseia-se na escolha de um elemento (denominado **pivô**) e dividir o vetor desordenado e duas partes: a parte da esquerda, contendo elementos menores do que o pivô e, a parte da direita, com elementos maiores do que o pivô. O pivô irá ficar na posição entre essas duas partes. Note que o pivô já vai estar na posição correta no vetor ordenado. O problema se reduz então em ordenar cada uma dessas partes

```
void QuickSort(int v[], int inicio, int fim)
{
  int meio;

  if (inicio < fim)
  {
    meio = Particao(v,inicio,fim);
    QuickSort(v,inicio,meio-1);
    QuickSort(v,meio+1,fim);
  }
}
```

Algoritmo *QuickSort*

- Exemplo:



Neste exemplo, os **pivôs** usados para a partição do vetor estão mostrados em **negrito**.

- A eficiência do algoritmo *QuickSort* depende do algoritmo de partição.

Algoritmo *QuickSort*

- Algoritmo de partição:

```
int Particao(int v[], int inicio, int fim)
{
  int i;
  int pivo,pos;

  pivo = v[inicio];
  pos = inicio;
  for (i = inicio+1; i <= fim; i++)
  {
    if (v[i] < pivo)
    {
      pos++;
      Troca(&v[i],&v[pos]);
    }
  }
  Troca(&v[inicio],&v[pos]);

  return pos;
}
```

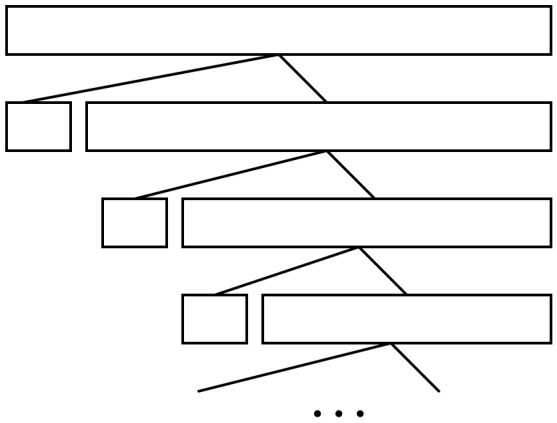
Como o passo de particionamento consiste de, no máximo, n trocas, a complexidade deste algoritmo é O(n).

Na melhor situação, cada passo de particionamento divide um problema de tamanho n em dois problemas de tamanho (aproximadamente) n/2. Neste caso teremos $\log_2(n)$ níveis na árvore de recursão. Portanto, o tempo de execução do algoritmo *QuickSort* é $O(n \times \log n)$.

Note, no entanto, que este algoritmo é melhor do que o *MergeSort*, pois não requer espaço adicional.

Algoritmo *QuickSort*

- Mas, o desempenho do algoritmo *QuickSort* depende da escolha do pivô. Imagine, por exemplo, no pior caso, que o pivô escolhido é sempre o menor elemento. Neste caso, a árvore de recursão terá a seguinte forma:



Neste caso, em vez de $\log_2 n$ níveis, vão existir (n – 1) níveis na árvore de recursão e, portanto, a complexidade do algoritmo será $O(n^2)$.

- Na média, o algoritmo *QuickSort* tem bom desempenho. Boas estratégias são: escolher o pivô aleatoriamente ou como o elemento na posição central do vetor. Isto evita que, no caso do vetor já estar ordenado (ou quase ordenado) a árvore de recursão seja como a do pior caso.