ORDENAÇÃO

- **Ordenar** é o processo de organizar uma lista de informações similares em ordem crescente ou decrescente. Especificamente, dada uma lista de *n* itens *r*[0], *r*[1], *r*[2], ..., *r*[*n*-1], cada item na lista é chamado **registro**. Uma **chave**, *k*[i], é associada a cada registro *r*[*i*]. Diz-se que a lista está **ordenada pela chave** se *i* precede *j* implicar que *k*[*i*] < *k*[*j*] (ou *k*[*i*] > *k*[*j*]) em alguma ordenação nas chaves

1. ORDENAÇÃO POR TROCA

1.1 Ordenação por Bolha

- Em cada um dos exemplos subseqüentes, *x* é um vetor de inteiros do qual os primeiros *n* devem ser ordenados de modo que $x[i] \leq x[j]$ para $0 \leq i < j <$ n.
- A idéia básica por trás da ordenação por bolha é percorrer a lista seqüencialmente varias vezes. Cada passagem consiste em comparar cada elemento na lista com seu sucessor (*x*[*i*] com *x*[*i*+1]) e trocar os dois elementos se ele não estiverem na ordem correta

Exemplo, 25, 57, 48, 37, 12, 92, 86, 33

Primeira Passagem

x[0] com x[1]	(25 com 57)	nenhuma troca
x[1] com x[2]	(57 com 48)	troca
x[2] com x[3]	(57 com 37)	troca
x[3] com x[4]	(57 com 12)	troca
x[4] com x[5]	(57 com 92)	nenhuma troca
x[5] com x[6]	(92 com 86)	troca
x[6] com x[7]	(92 com 33)	troca

Observe que, depois da primeira passagem, o maior elemento está em sua posição correta. Em geral $x[n-i]$ ficara na posição correta depois da iteração i.

25, 57, 48, 37, 12, 92, 86, 33

O conjunto completo de iterações fica assim:

iteração 0	25 57 48 37 12 92 86 33
iteração 1	25 48 37 12 57 86 33 92
iteração 2	25 37 12 48 57 33 86 92
iteração 3	25 12 37 48 33 57 86 92
iteração 4	12 25 37 33 48 57 86 92
iteração 5	12 25 33 37 48 57 86 92
iteração 6	12 25 33 37 48 57 86 92
iteração 7	12 25 33 37 48 57 86 92

Algoritmo

```
bubble (int x[ ], int n)
{
   int j, pass;
   bool switched = true;

   for (pass = 0; pass < n-1 && switched; pass++){   /*repetição externa,
                                                       controla nº de passagens*/
        switched = false;
        for (j = 0; j < n - pass - 1; j++){      /*repetição interna, controla cada
                                                   passagem individual*/
             if (x[j] > x[j+1]){        /*elemento fora da ordem é necessária uma troca*/
                  switched = true;
                  troca(x[j], x[j+1]);
             }
        }
   }
}
```

Complexidade de Tempo

- Sem o aprimoramento

  Numero de comparações: $n(n-1)/2$

  Numero de trocas: melhor caso: nenhuma

  pior caso: $n(n-1)/2$

- Com o aprimoramento

  Numero de comparações será

  $(n-1) + (n-2) + ... + (n-k) = (2kn-k^2-k)/2$

  Como $k = O(n)$, então, o aprimoramento não muda a complexidade de tempo do algoritmo

- Conclusão final: A complexidade do algoritmo de ordenação por bolha = $O(n^2)$

1.2 QuickSort

A ordenação por troca de partição ou quicksort é provavelmente o algoritmo de ordenação mais utilizado.

Idéia Básica

- Quicksort trabalha particionando um vetor em duas partes e então as ordenando separadamente. Especificamente, seja *x* um vetor e *n* o numero de elementos no vetor a ser classificados. Escolha um elemento *a* numa posição especifica dentro do vetor, digamos a posição *j*. Os elementos de *x* são particionados de modo que *a* é colocado na posição *j* e as seguintes condições são observadas:
  1. Cada elemento nas posições 0 até *j*-1 são menor ou igual a *a*.
  2. Cada elemento nas posições *j*+1 até *n* -1 são maior que a *a*.
- O mesmo processo é repetido com os subvetores *x*[0] até *x*[*j*-1] e *x*[*j*+1] até *x*[*n*-1] e com quaisquer vetores criados pelo processo em sucessivas iterações, o resultado final será uma lista ordenada.

Algoritmo Básico

```
quicksort (int x[], int: lb, ub)
{
      int i;

      if (lb > ub) return;
       j = partition(x, lb, ub);
      quicksort(x, lb, j-1);
      quicksort(x, j+1, ub);
}
```

- Os parâmetros *lb* e *ub* delimitam os sub-vetores dentro do vetor original, dentro dos quais a ordenação ocorre.
- A chamada inicial pode ser feita com quicksort(*x*, 0, *n*-1);
- O ponto crucial é o algoritmo de partição.

Exemplo:

Ordenação do vetor inicial 25 57 48 37 12 92 86 33.

Suponhamos que o primeiro elemento (25) é escolhido para colocar na sua posição correta, teremos:

12 25 57 48 37 92 86 33.

Como 25 está na sua posição final, o problema foi decomposto na ordenação dos subvetores:

(12) e (57 48 37 92 86 33).

O subvetor (12) já está classificado. Repetir o processo para *x*[2]...*x*[7] resulta em:

12 25 (48 37 33) 57 (92 86)

Exemplo:

Se continuarmos particionando 12 25 (48 37 33) 57 (92 86), teremos:

12 25 (37 33) 48 57 (92 86)

12 25 (33) 37 48 57 (92 86)

12 25 33 37 48 57 (92 86)

12 25 33 37 48 57 (86) 92

12 25 33 37 48 57 86 92

Método de Particionamento

Considere *a* = *x*[*lb*] como o elemento cuja posição final é a procurada.

Dois ponteiros *up* e *down* são inicializados como os limites máximo e mínimo do subvetor que vamos analisar. Em qualquer ponto da execução, todo elemento acima de *up* é maior do que *a* e todo elemento abaixo de *down* é menor ou igual a *a*.

Os dois ponteiros *up* e *down* são movidos um em direção ao outro da seguinte forma:

1. Incremente *down* em uma posição até que $x[down] > a$.
2. Decremente *up* em uma posição até que $x[up] \leq a$.
3. Se a $up > down$, troque $x[down]$ por $x[up]$.

O processo é repetido até que a condição descrita em 3. falhe (quando $up \leq down$). Neste ponto $x[up]$ será trocado por $x[lb]$, cuja posição final era procurada, e *up* é retornado em *j*.

Exemplo: $a = 25$

down-->							up
25	57	48	37	12	92	86	33
	down						<--up
25	57	48	37	12	92	86	33
	down					<--up	
25	57	48	37	12	92	86	33
	down				<--up		
25	57	48	37	12	92	86	33
	down			up			
25	57	48	37	12	92	86	33
	down			up			
25	12	48	37	57	92	86	33

```
        down-->                         up
25   12   48   37   57   92   86   33
             down                       up
25   12   48   37   57   92   86   33
             down                  <--up
25   12   48   37   57   92   86   33
             down  <--up
25   12   48   37   57   92   86   33
          <--up  down
25   12   48   37   57   92   86   33
        up   down
25   12   48   37   57   92   86   33
        up   down
12   25   48   37   57   92   86   33
```

Algoritmo de Particionamento

```
int partition(int x[], int: lb, ub)
{
      int a, down, up;

      a = x[lb];
      up = ub; down = lb;
      while (down < up) {
            while (x[down] <= a && down < ub)
                  down++;
            while (x[up] > a)
                  up--;
            if (down < up)
                  swap(x[down], x[up]);
       }
        x[lb] = x[up];
        x[up] = a;
        return up;
}
```

Eficiência do Quicksort

O tempo de execução do QuickSort depende se o particionamento é balanceado ou não.

1. **O pior caso** do QuickSort ocorre quando o particionamento gera um conjunto com 1 elemento e outro com n-1 elementos para todos os passos do algoritmo. Desde que o particionamento custa $O(n)$ a recorrência neste caso torna-se

   $$T(n) = T(n\text{-}1) + O(n)$$

   como $T(1) = O(1)$, não é difícil mostrar que $T(n) = O(n^2)$.

   $(T(n) = \alpha n + T(n\text{-}1) = \alpha n + \alpha(n\text{-}1) + \alpha(n\text{-}2) + ... + \alpha 2 + T(1))$

2. **O melhor caso** ocorre quando o particionamento sempre gera dois sub-conjuntos de tamanho $n/2$, temos a recorrência

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

   Então, $T(n) = O(n\log n)$.

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

$$= 2\left(2T\left(\frac{n}{4}\right) + \alpha\frac{n}{2}\right) + \alpha n = 2^2 T\left(\frac{n}{4}\right) + \alpha n + \alpha n$$

$$= 2^2\left(2T\left(\frac{n}{8}\right) + \alpha\frac{n}{4}\right) + \alpha n + \alpha n 2 = 2^3 T\left(\frac{n}{8}\right) + \alpha n + \alpha n + \alpha n$$

$$= 2^{\log n} T(1) + \underbrace{\alpha n + \ldots + \alpha n}_{\log n}$$

$$= 2^{\log n} + \alpha n \log n$$

$$= O(n \log n)$$

$$\frac{n}{2^m} = 1$$
$$m = \log n$$

$$2^{\log n} < n \log n$$
$$\log 2^{\log n} < \log(n \log n)$$
$$\log n < \log n + \log(\log n)$$

3. Caso Médio.

O algoritmo *partition* leva tempo proporcional ao *n*, denotado αn. Suponhamos que a lista é separada em duas sublistas de comprimento *k* e *n* -1-*k*, respectivamente. Então, temos

$$T(n) = \alpha n + T(k) + T(n - 1 - k)$$

Mas, não sabemos o valor exato do *k*. Portanto, calculamos a media de todas possibilidades de *k*:

$$T(n) = \alpha n + \frac{1}{n}\sum_{k=0}^{n-1}\left(T(k) + T(n-1-k)\right)$$

Isso pode ser simplificado para: (Porquê ?)

$$T(n) = \alpha n + \frac{2}{n}\sum_{k=1}^{n-1} T(k)$$

$$\frac{n}{2}(T(n)-\alpha n) = T(0)+T(1)+\ldots+T(n-1)$$

$$\frac{n+1}{2}(T(n+1)-\alpha(n+1)) = T(0)+T(1)+\ldots+T(n-1)+T(n)$$

Substrai as duas formulas e rearranja:

$$\frac{T(n+1)}{n+2} = \frac{T(n)}{n+1} + \frac{\alpha(2n+1)}{(n+1)(n+2)}$$

$$S(n) = \frac{T(n)}{n+1}$$

$$S(n+1) = S(n) + \frac{\alpha(2n+1)}{(n+1)(n+2)}$$

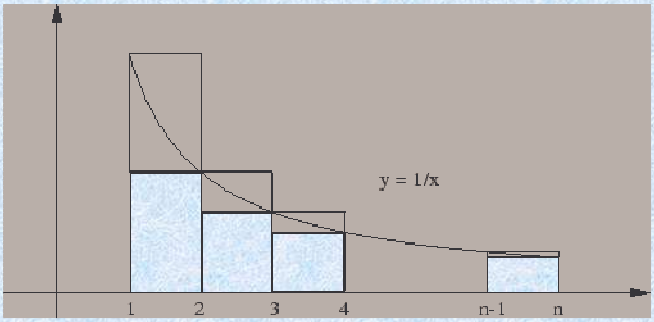
$$\begin{aligned} S(n+1) &= S(0) + 3\alpha\sum_{k=0}^{n-1}\frac{1}{k+2} - \sum_{k=0}^{n-1}\frac{1}{k+1} \\ &= S(0) + 3\alpha(H(n+1)-1) - \alpha H(n) \\ &= S(0) + 2\alpha H(n) + \frac{3\alpha}{n+1} - 3\alpha \end{aligned}$$

onde $H(n) = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \sim \ln n$

$$T(n) = 2\alpha(n+1)H(n) + (S(0)-3\alpha)(n+1) + 3\alpha$$

Então, $T(n) = O(n \ln n)$

Aproximando $H(n)$



$$1+\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n-1}>\int_1^n \frac{dx}{x}=\ln n$$

$$\frac{1}{2}+\frac{1}{3}+\ldots+\frac{1}{n-1}+\frac{1}{n}<\int_1^n \frac{dx}{x}=\ln n$$

$$\ln n+\frac{1}{n}<H(n)<\ln n+1$$

Comentários sobre o Quicksort

Para evitar o pior caso e casos ruins onde elementos estão em grupos ordenados pode-se utilizar uma estratégia probabilística: Selecione, ao invés do primeiro elemento para ser *a*, um elemento aleatório. Critérios:

- Seleção totalmente randômica: selecione qualquer elemento do subvetor usando um gerador de números aleatórios.
  Desvantagem: tempo de processamento extra para o gerador.
- Seleção da mediana entre os três elementos: lb, ub e elemento no meio.
- Seleção média: usa o valor media do subvetor. Todos estes métodos melhoram a performance média.

2 ORDENAÇÃO POR INSERÇÃO

Uma **ordenação por inserção** é a que ordena um conjunto de registros inserindo registros num arquivo ordenado já existente.

2.1 Inserção Simples

Exemplo: 4, 3, 6, 1, 5, 2

i = 5	4 3 6 1 **5** 2	temp = 5
i = 4	4 3 6 **1** 2 5	temp = 1
i = 3	4 3 **6** 1 2 5	temp = 6
i = 2	4 **3** 1 2 5 6	temp = 3
i = 1	**4** 1 2 3 5 6	temp = 4
i = 0	1 2 3 4 5 6	temp =

```
insertsort(int x[ ], int n)
{
  /*inicialmente x[n-1] é considerado um arq ordenado de um elemento*/
  /*após k passagens, os elementos x[k-1] a x[n-1] estarão em seqüência*/
  int i, j, temp;
  for (i = n-2; i >= 0; i--){
       temp = x[i] ;
       j = i+1;
       while (j <= n-1  &&  x[j] < temp) {
             x[j - 1] = x[j];
             j = j+1;
       }
      x[j - 1] = temp;        /*insere temp na posição correta*/
   }
}
```

Eficiência do Algoritmo de Inserção Simples

1. **Numero de comparações**:

$$1 + 2 + 3 + ... + (n - 1) = \frac{n(n-1)}{2}$$

2. **Numero de deslocamentos**

Para *n* grande o componente principal do tempo gasto pelo algoritmo é o laço do *while* (deslocamento dos elementos à esquerda). Para analisar complexidade do algoritmo, precisamos calcular a soma dos números de deslocamento para cada um $i = n - 2, ..., 0$.

No exemplo anterior: 4, 3, 6, 1, 5, 2

i	4	3	2	1	0
distancia	1	0	3	2	3

distancia total 9.

1. **Melhor Caso**. Numero de deslocamento é zero, uma vez a lista original já está em ordem.
2. **Pior Caso**. Para uma lista de n itens, o numero de deslocamento é

$$1 + 2 + 3 + ... + (n - 1) = \frac{n(n-1)}{2}$$

Isto acontece se a lista está em ordem reversa.

3. Caso Médio. Considere agora o algoritmo. No estagio *i* nós fazemos exame do elemento *x*[*i*] e movemos a uma distância $d_i \in \{ 0, 1..., n - i\text{-}1 \}$ a sua posição correta. *x*[*i*] não foram tocados pelo algoritmo até este ponto assim, não temos nenhuma informação sobre sua posição apropriada na lista. Ou seja todas as SHIFT possíveis para *x*[*i*] são igualmente prováveis. Assim a distancia média movida por *x*[*i*] está

$(0 + 1 + 2 + ... + (n - i - 1))/(n - i) = \left(\frac{1}{n-i}\right)\left(\frac{(n-i-1)(n-i)}{2}\right) = \frac{n-i-1}{2}$

Então, em media, o numero total de deslocamento é

$$\sum_{i=0}^{n-2} \frac{n-i-1}{2} = \frac{n(n-1)}{4}$$

Portanto, este é um $O(n^2)$ algoritmo.

2.2 Ordenação de Shell (Ordenação de Incremento Decrescente)

- O algoritmo de ordenação de Shell (nome de seu inventor Donald Shell) foi um dos primeiros a baixar a complexidade de $O(n^2)$.
- O algoritmo usa uma seqüência de incrementos e ordena os elementos cujo a distância é igual a este incremento. A cada etapa o incremento vai diminuindo até chegar a 1, ao final de uma etapa com incremento *k*, dizemos que o conjunto está *k*-ordenado. Uma importante propriedade deste algoritmo é que para $k1 < k2$ um conjunto *k*1-ordenado, que é submetido a uma ordenação com incremento *k*2, permanece *k*1-ordenado.

Por exemplo, se $k = 5$, cinco subvetores, cada um contendo um quinto do elementos do vetor original, são ordenados. São eles:

Subveotr 1	x[0]	x[5]	x[10]	...
Subvetor 2	x[1]	x[6]	x[11]	...
Subvetor 3	x[2]	x[7]	x[12]	...
Subvetor 4	x[3]	x[8]	x[13]	...
Subvetor 5	x[4]	x[9]	x[14]	...

O *i*-ésimo elemento do *j*-ésimo subvetor é x[(i-1)*5+j-1]. Se um incremento *k* diferente for escolhido, os k subvetores são divididos de modo que o *i*-ésimo elemento do *j*-ésimo subvetor seja x[(i-1)*k+j-1].

Exemplo: 25, 57, 48, 37, 12, 92, 86, 33

primeira iteração (incremento = 5) (x[0], x[5])
(x[1], x[6])
(x[2], x[7])
(x[3])
(x[4])

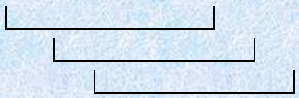
segunda iteração (incremento = 3) (x[0], x[3], x[6])
(x[1], x[4], x[7])
(x[2], x[5])

terceira iteração (incremento = 1)
(x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7])

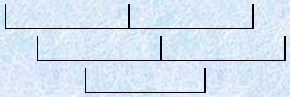
Exemplo:

lista original 25 57 48 37 12 92 86 33
passagem 1

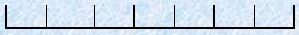
incremento = 5 25 57 48 37 12 92 86 33



passagem 2
incremento = 3 25 57 33 37 12 92 86 48



passagem 3
incremento = 1 25 12 33 37 48 92 86 57



Lista classificada 12 25 33 37 48 57 86 92

```
void shellsort(int x[ ],  int n,  int incrmnts[ ],  int numinc)
/* incrts eh um array contendo os incrementos */
{
  int incr, j, k, span, temp;

  for (incr = 0; incr < numinc; incr++){
       span = incrmnts[incr];    /* span eh o tamanho do incremento */
       for (j = span; j < n; j++){
            temp = x[j];
            /*insere elemento x[j] em sua posição correta dentro de seu  subvetor*/
            for (k = j - span;  k >= 0 && temp < x[k]; k -= span)
                 x[k+span] = x[k];
            x[k+span] = temp;
       }
  }
}
```

Idéia Básica de Ordenação de Shell

1. A ordenação por inserção simples é altamente eficiente sobre uma lista de elementos quase ordenado.
2. Quando o tamanho de lista n é pequeno, uma ordenação $O(n^2)$ é em geral mais eficiente do que uma ordenação $O(n\log n)$. Isto acontece porque usualmente as ordenações $O(n^2)$ são muito simples de programar e exigem bem poucas ações alem de comparações e trocas em cada passagem. Por causa dessa baixa sobrecarga, a constante de proporcionalidade é bem pequena. Em geral, uma ordenação $O(n\log n)$ é muito complexa e emprega um grande numero de operações adicionais em cada passagem para diminuir o trabalho das passagens subsequentes. Sendo assim, sua constante de proporcionalidade é maior.

Observação: em geral, a ordenação de Shell é recomendada para as listas de tamanho moderado, da varias centenas de elementos.

Eficiência de Ordenação de Shell

- A ordem da ordenação de Shell pode ser aproximada por $O(n(\log n)2)$ se for usada uma seqüência apropriada de incrementos.
- Para outras seqüências de incrementos, o tempo de execução pode provar-se como $O(n^{1.5})$.

3 ORDENAÇÃO POR SELEÇÃO

Uma **ordenação por seleção** é aquela na qual sucessivos elementos são selecionados em seqüência e dispostos em suas posições corretas pela ordem.

3.1 Ordenação de Seleção Direta (T(n) = $O(n^2)$)

66 33 21 84 49 50 75

66 33 21 84 49 50 75
66 33 21 75 49 50 84
66 33 21 50 49 75 84
49 33 21 50 66 75 84
49 33 21 50 66 75 84
21 33 49 50 66 75 84
21 33 49 50 66 75 84

```
selectsort (int x[ ], int n)
{
   int i, indx, j, large;
   for (i = n-1; i > 0; i--){
      /*coloca o maior elemento de x[0] até x[i] em large e seu indice em indx*/
      large = x[0];
      indx = 0;
      for (j = 1; j <= i; j++)
         if (x[j] > large) {
            large = x[j];
            indx = j;
         }
      x[indx] = x[i];
      x[i] = large;
   }
}
```

3.2 Ordenação por Árvore Binária

Idéia Básica: Coloca cada elementos de uma lista de elementos em uma arvore binaria de busca. A lista ordenada é obtida por uma atravessia em-ordem.

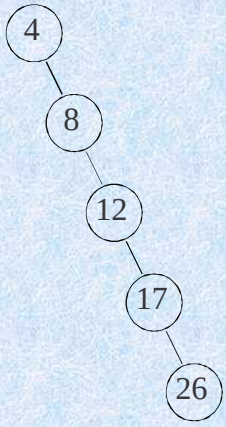
```
/*estabelece o primeiro elemento como raiz*/
tree = maketree(x[0]);
/*repete para cada elemento succesivo*/
for(i = 1; i < n; i++){
      y = x[i];
      q = root;
      p = q;
      /*atravessa a arvore até encontrar uma folha*/
      while (p != NIL){
            q = p;
            if (y < key[p])
                  p = left(p);
            else
                  p = right(p);
      }/*end while*/
      if (y < key[q])
            left[q] = y;
      else
            right[q] = y;
}/*end for */
/*a arvore está construida, percorre em em-ordem
intravel(tree);
```

Eficiência de Ordenação por Árvore Binária

- A eficiência relativa desse método dependerá da ordem original dos dados
- **Pior caso**. Se o vetor original estiver totalmente ordenado (ou ordenado em em ordem inversa), a arvore resultante aparecerá como uma seqüência de ligações direta (ou esquerda). Neste caso, a inserção do primeiro nó não exige comparações, o segundo nó requer duas comparações, o terceiro nó requer três comparações. Sendo assim , o numero total de comparações é:

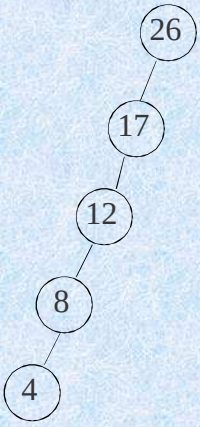
$$2+3+...+n=\frac{n(n+1)}{2}-1=O(n^2)$$

Dadas originais: 4, 8, 12, 17, 26



Numero de comparações: 14

Dadas originais: 26, 17, 12, 8, 4



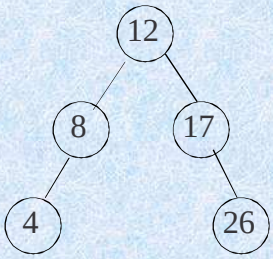
Numero de comparações: 14

- **Melhor caso**. Se os dados no vetor original estiverem organizados de modo que aproximadamente metade dos números posteriores a determinado numero *a* no vetor sejam menores que *a*, e metade sejam maiores que *a*, resultarão arvores balanceadas. Neste caso, o numero de nós em qualquer nível *l* (exceto talvez para o ultimo) é 2^l e o numero de comparações necessárias para colocar um nó no nível *l* (exceto quando *l* = 0) é *l*+1. Sendo assim, o numero total de comparações fica entre

$$d + \sum_{l=1}^{d-1} 2^l (l+1) \sim \sum_{l=1}^{d} 2^l (l+1)$$

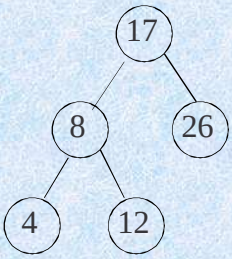
Onde $d = \lfloor \log_2(n+1) - 1 \rfloor$, é a profundidade da arvore.
Então, $T(n) = O(n \log n)$

Dadas originais: 12, 8, 17, 4, 16



Numero de comparações: 10

Dadas originais: 17, 8, 12, 4, 26



Numero de comparações: 10

- **Caso médio**. Pode-se provar que, se toda ordenação possível da entrada for considerada igualmente possível, isso resultará em arvores balanceadas mais freqüentes. O tempo médio é, portanto, $O(n\log n)$.
- **Espaço ocupado**. Essa classificação exige que um nó da arvore seja reservado para cada elemento do vetor.