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, $T(n) = O(n \log n)$.