QUICKSORT

Esse algoritmo, chamado "ordenação rápida", em geral é mais eficiente do que os três algoritmos simples de ordenação – BubbleSort (por trocas), Seleção (por seleção), Inserção (por inserção). É baseado em trocas, assim como o BubbleSort, mas realiza trocas com elementos não necessariamente adjacentes.

A idéia central do quicksort é colocar um elemento escolhido da lista (denominado pivô) na sua posição definitiva, deixando os elementos menores que o pivô à sua esquerda e os maiores à sua direita. Ou seja, o pivô separa a lista em duas sublistas: a dos menores, à esquerda, e a dos maiores, à direita. Esse processo é repetido para as sublistas até que elas sejam de tamanho unitário.

O algoritmo torna-se muito eficiente se o elemento escolhido como pivô separa a lista aproximadamente em duas sublistas de mesmo tamanho, isto é, se o pivô é um elemento do meio da lista na ordenação final. Essa separação ao "meio" (em todas as sublistas) faz com que a complexidade do algoritmo seja da ordem de nlgn. No pior caso, o quicksort tem desempenho proporcional a n^2.

No algoritmo a seguir, p é a primeira posição e n é a última posição da lista.

quicksort(p,n,A)

```
se (p < n)  então    j ← separarLista(p,n,A);
                     quicksort(p,j-1,A);
                     quicksort(j+1,n,A)
```

O algoritmo separarLista que apresentamos a seguir utiliza como pivô o elemento que está na posição p da sublista, que é a primeira posição. Após a execução desse algoritmo, o pivô estará na posição j, todos os elementos menores que o pivô estarão na sublista [p...j-1] e os elementos maiores que o pivô estarão na sublista [p+1...n]. O algoritmo separarLista devolve um valor inteiro, que é a posição j. (Caso a lista contenha duplicatas, a sublista esquerda deverá conter elementos que são menores ou iguais ao pivô, enquanto que a sublista direita deverá conter elementos que são maiores ou iguais ao pivô.)

separarLista(p,n,A)

```
i ← p;   j ← n+1;
enquanto  (i < j)  faça
    repita  i  ←  i+1  até que  ((A[i] ≥ A[p] ou (i=n));
    repita  j  ←  j – 1  até que  A[j] ≤ A[p];
    se (i < j) então  aux  ←  A[i];  A[i]  ←  A[j];  A[j]  ←  aux;   ;
aux  ←  A[p];  A[p]  ←  A[j];  A[j]  ←  aux;
devolver j
```

Sentinelas: o cursor j para necessariamente na posição do pivô. Para segurar o cursor i é necessário colocar uma cópia do pivô como sentinela à direita ou então colocar uma condição dupla no loop de varredura: A[i]≥A[p] ou i = n.

Simulação do algoritmo

1	2	3	4	5	6	7	8	p	n	j
5	**9**	**6**	**4**	**3**	**8**	**7**	**1**	1	8	4
4	**1**	**3**	5	6	8	7	9	1	3	3
3	**1**	4	5					1	2	2
1	3	4	5					1	1	
1	3	4	5					3	2	
1	3	4	5					4	3	
1	3	4	5	**6**	**8**	**7**	**9**	5	8	5
				6	8	7	9	5	4	
					8	**7**	**9**	6	8	7
					7	8	9	6	6	
1	**3**	**4**	**5**	**6**	**7**	**8**	**9**	8	8	

Chamadas:

```
quicksort(1,8)
    quicksort(1,3)
        quicksort(1,2)
            quicksort(1,1)
            quicksort(3,2)
        quicksort(4,3)
    quicksort(5,8)
        quicksort(5,4)
        quicksort(6,8)
            quicksort(6,6)
            quicksort(8,8)
```

HEAPSORT

Esse algoritmo tem desempenho proporcional a nlgn e utiliza uma estrutura de dados chamada "heap" que representa uma árvore binária. Construímos uma heap máxima, onde cada elemento é maior que os seus dois filhos. Em seguida, para colocar a lista em ordem crescente basta trocar o primeiro elemento (que é o maior) com o último, de modo que ele fica na sua posição definitiva, reduzir o tamanho da lista de uma unidade e repetir o processo até que a lista reduzida tenha um único elemento.

heapSort(A);

```
construirHeap(A);
ordenarHeap(A);
```

Comentários:

A lista está armazenada em uma estrutura A, representada por alocação sequencial. Consideramos que o tamanho da lista pode ser obtido em nA. A primeira fase do processo de ordenação é a construção da estrutura heap. Em seguida inicia-se a ordenação.

construirHeap(H)

```
se (nH > 1)
    então upai ← nH DIV 2;   k ← upai;   pai ← k;
          enquanto (k ≥ 1) faça
              e ← 2 × k;   d ← 2 × k + 1;   maior ← H[e];   f ← e;
              se (d ≤ nH)  então  se (H[d] > maior)  então  maior ← H[d];   f ← d;
              se (H[k] < maior)  então  trocar(H,k,f);
              se (f > upai) então  k ← pai – 1;   pai ← k;
                            senão  k ← f;
```

Comentários:

Se a lista tem pelo menos 2 itens, inicia-se a construção da heap. Esse processo é uma "restauração" da estrutura heap.

O último nó da árvore que é um nó pai é o que ocupa a posição $\lfloor n/2 \rfloor$ na lista. A construção se inicia neste nó. Seja upai a posição do último pai.

Para cada nó k, a partir de upai-1 até o nó raiz, que ocupa a posição 1, restauramos a heap máxima na subárvore de raiz k (a restauração é heap acima).

Para isso, determinamos o nó f, maior dos dois filhos do nó k (trocamos, se for o caso) e se f é uma folha então retomamos a restauração a partir do pai anterior, senão descemos para a subárvore de raiz f.

ordenarHeap(A)

```
u ← nA;
enquanto (u > 1) faça
    trocar(A,1,u);   u ← u-1;
    restaurarHeapAbaixo(A, u)
```

restaurarHeapAbaixo(H, u);

```
upai ← u DIV 2;    k ← 1;
enquanto (k ≤ upai) faça
    e ← 2 × k;   d ← 2 × k + 1;   maior ← H[e];   f ← e;
    se (d ≤ u)  então  se (H[d] > maior)  então  maior ← H[d];   f ← d;
    se (H[k] < maior)   então  trocar(H,k,f);   k ← f;
                        senão k ← upai + 1;
```

Simulação do algoritmo construirHeap

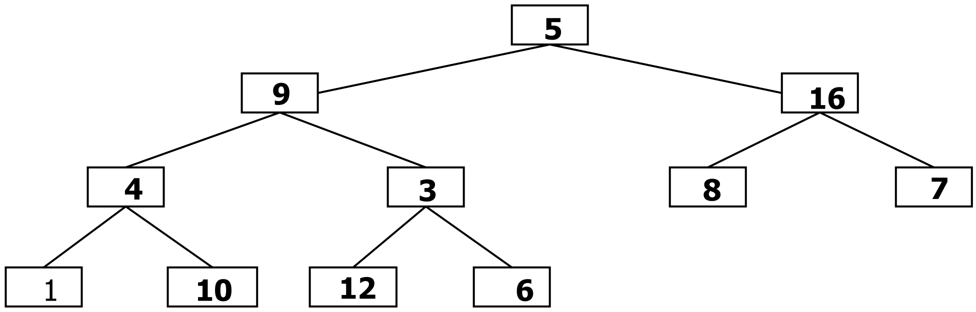
1	2	3	4	5	6	7	8	pai	
5	9	6	**4**	3	8	7	**1**	4	
5	9	**6**	4	3	**8**	**7**	1	3	trocar
5	**9**	8	**4**	**3**	6	7	1	2	
5	**9**	**8**	4	3	6	7	1	1	trocar
9	5	8	4	3	6	7	1	heap	

Simulação do algoritmo ordenarHeap

1	2	3	4	5	6	7	8	u	
9	5	8	4	3	6	7	1	8	heap
1	**5**	**8**	4	3	6	7	9	7	restaurar
8	5	**1**	4	3	**6**	**7**	9	7	restaurar
8	5	7	4	3	6	1	9	7	heap
1	**5**	**7**	4	3	6	8	9	6	restaurar
7	5	**1**	4	3	**6**	8	9	6	restaurar
7	5	6	4	3	1	8	9	6	heap
1	**5**	**6**	4	3	7	8	9	5	restaurar
6	5	1	4	3	7	8	9	5	heap
3	**5**	**1**	4	6	7	8	9	4	restaurar
5	**3**	1	**4**	6	7	8	9	4	restaurar
5	4	1	3	6	7	8	9	4	heap
3	**4**	**1**	5	6	7	8	9	3	restaurar
4	3	1	5	6	7	8	9	3	heap
1	3	4	5	6	7	8	9	2	restaurar
3	1	4	5	6	7	8	9	2	heap
1	3	4	5	6	7	8	9	1	fim

EXERCÍCIOS

1) O algoritmo separarLista, utilizado no quicksort funciona corretamente no caso em que a lista é unitária?
2) Qual o resultado fornecido pelo algoritmo separarLista no caso em que a lista tem todos os elementos iguais?
3) Qual o resultado fornecido pelo algoritmo separarLista no caso em que a lista tem somente valores de dois únicos tipos?
4) Qual o resultado fornecido pelo algoritmo separarLista no caso em que a lista não contém duplicatas e está em ordem crescente?
5) Qual o resultado fornecido pelo algoritmo separarLista no caso em que a lista não contém duplicatas e está em ordem decrescente?
6) O algoritmo separarLista produz uma ordenação estável da lista?
7) Escreva uma versão recursiva do algoritmo separarLista.
8) Qual a sequência de chamadas de quicksort, caso a lista A seja 99, 55, 33, 77 ?
9) Qual a sequência de chamadas de quicksort, caso a lista A seja 55, 44, 22, 11, 66, 33?
10) Escreva uma versão não recursiva do algoritmo quicksort.
11) A lista A = 161, 41, 101, 141, 71, 91, 31, 21, 81, 17, 16 é um heap-máximo?
12) Escreva um algoritmo que verifique se uma lista é um heap-máximo.
13) Utilize o algoritmo ordenarHeap para ordenar a lista A = 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4.
14) Utilize o algoritmo ordenarHeap para ordenar a lista A = 8,1,7,25,20,17,12,5.
15) Faça a simulação do algoritmo heapSort por meio de desenhos, considerando a lista representada a seguir:



16) Escreva uma versão recursiva do algoritmo construirHeap.
17) Escreva uma versão recursiva do algoritmo restaurarHeap.