Algoritmos de Classificação de Tabelas - Sorting

A classificação de tabelas e conjuntos de dados é uma operação bastante usada em qualquer área da computação. Os algoritmos de classificação estão dentre os algoritmos fundamentais.
Existem várias formas de se classificar uma tabela. Veremos algumas delas a seguir. Vamos nos restringir a classificação interna, isto é, tabelas que estão na memória. A classificação externa, de arquivos, segue a mesma linha, com algumas adaptações para otimizar a quantidade de operações de entrada e saída.

Trocar os valores de 2 elementos de um vetor

Nos algoritmos de classificação uma operação que ocorre com frequência é a troca de valores de dois elementos do vetor. Vejamos algumas formas de fazer isso.

Considere a função **`troca`** abaixo:

```
void troca(int* x, int* y) {
   int aux = *x; *x = *y; *y = aux;
}
```

Para trocar **`a[i]`** com **`a[j]`** a chamada seria: **`troca(&a[i], &a[j])`** .

Outra forma de **`troca`**:

```
void troca(int x[], int k1, int k2) {
    /* troca os valores de x[k1] com x[k2] */
    int aux = x[k1]; x[k1] = x[k2]; x[k2] = aux;
}
```

Agora para trocar **`a[i]`** com **`a[j]`** a chamada seria: **`troca(a, i, j)`**.

Uma maneira mais eficiente de trocar **`a[i]`** com **`a[j]`** é simplesmente usar as três atribuições no próprio corpo do programa, sem usar uma função:

```
int aux = a[i]; a[i] = a[j]; a[j] = aux;
```

Para ficar mais claro podemos fazer uma macro usando o **`#define`**:

```
#define troca(x, y) { int aux = x; x = y; y = aux; }
```

O uso desta macro seria então:

```
troca(a[i], a[j]);
```

Vamos usar esta última forma nos algoritmos a seguir.

Classificação – método da seleção

O algoritmo imediato para se ordenar uma tabela com `n` elementos é o seguinte:

1. Determinar o mínimo a partir do primeiro e trocar com o primeiro
2. Determinar o mínimo a partir do segundo e trocar com o segundo
3. Determinar o mínimo a partir do terceiro e trocar com o terceiro

. . . .
. . . .
n-1. Determinar o mínimo a partir do (n-1)-ésimo e trocar com o (n-1)-ésimo

Exemplo:

```
6    2    2    2
8    8    4    4
4    4    8    5
2    6    6    6
5    5    5    8
```

A função `selecao` abaixo classifica a sequência pelo método da seleção.
O programa abaixo, dado `n`, gera uma sequência com `n` elementos usando `rand()`, imprime a sequência gerada, classifica a sequência usando `selecao` e imprime a sequência classificada.

```
#include <stdio.h>
#include <stdlib.h>

void selecao(int a[], int n) {
   int i, j, imin;
   /* varrer o vetor de a[0] até a[n-2] (penúltimo) */
   for (i = 0; i < n - 1; i++) {
         /* achar o mínimo a partir de a[i] */
         imin = i;
         for (j = i + 1; j < n; j++)
               if (a[imin] > a[j]) imin = j;
         /* troca a[imin] com a[i] */
         troca(a[imin], a[i]);
   }
}

void geravet(int v[], int k) {
  /* gera vetor com k elementos usando rand() */
  int i;
  srand(9999);
  for (i = 0; i < k; i++) v[i] = rand()%1000;
}

void impvet(int v[], int k) {
  /* imprime vetor com k elementos */
```

```
  int i;
  for (i = 0; i < k; i++) printf("%5d", v[i]);
}

int main() {
  int vet[1000], n, kk, x;
  /* ler n */
  printf("entre com n:");
  scanf("%d", &n);
  /* gera o vetor e imprime */
  geravet(vet, n);
  printf("\n****** vetor gerado ******\n");
  impvet(vet, n);
  /* ordena o vetor */
  selecao (vet, n);
  /* imprime vetor ordenado */
  printf("\n****** vetor ordenado ******\n");
  impvet(vet, n);
 }
```

Um exemplo de execução do programa:

```
entre com n:100

****** vetor gerado ******
  691  995  720  855  388  105  163  288  657  851  499  771  920  363   73    3
  514  633  815  153  899  757  934  575  387  841  748  872  726  818  172   62
  473  640   93  724  734  620  136   86  370  610  740  421  605  119   84  631
  217  740  580  286  974  436   40  335  526  923  918  511  973  535  892  945
  115  515  269   43   93  929  984  330  688  488  961  266  152   14  100  899
  264  344   56    8   50  200   88  274  554  135  976   20  998  435  966  592
  209  658  158  643
****** vetor ordenado ******
    3    8   14   20   40   43   50   56   62   73   84   86   88   93   93  100
  105  115  119  135  136  152  153  158  163  172  200  209  217  264  266  269
  274  286  288  330  335  344  363  370  387  388  421  435  436  473  488  499
  511  514  515  526  535  554  575  580  592  605  610  620  631  633  640  643
  657  658  688  691  720  724  726  734  740  740  748  757  771  815  818  841
  851  855  872  892  899  899  918  920  923  929  934  945  961  966  973  974
  976  984  995  998
```

Classificação – método da seleção – análise simplificada

Número de trocas: é sempre n–1. Não seria necessário trocar se o mínimo fosse o próprio elemento.

Número de comparações: é sempre (n-1)+(n-2)+...+2+1 = n(n-1)/2

O número de comparações representa a quantidade de vezes que o laço principal do algoritmo é repetido. Assim, o tempo deste algoritmo é proporcional a n^2, ou seja, o método da seleção é $O(n^2)$.

Classificação – Método bubble (da bolha)

Um outro método para fazer a ordenação é pegar cada um dos elementos de a[1] até a[n-1] e subi-los até que encontrem o seu lugar.

```
6    6    4    4    4    2    2    2
8    4    6    6    2    4    4    4
4    8    8    2    6    6    6    5
2    2    2    8    8    8    5    6
5    5    5    5    5    5    8    8
```

Observe como cada elemento sobe até encontrar o seu lugar. Dizemos que cada elemento borbulha até encontrar o seu lugar.

A função `bubble` abaixo classifica a sequência por esse método.
O programa abaixo, dado n, gera uma sequência com n elementos usando rand(), imprime a sequência gerada, classifica usando a função `bubble` e imprime a sequência classificada.

```
#include <stdio.h>
#include <stdlib.h>

void bubble(int a[], int n) {
   int i, j;
   /* tentar subir com a[i], i = 1, 2, . . ., n-1 */
   for (i = 1; i < n; i++) {
         /* suba com a[i] até encontrar um menor ou chegar em a[0] */
         j = i;
         while (j > 0 && a[j] < a[j-1]) {
               /* troca a[j] com a[j-1] */
               troca(a[j], a[j-1]);
               /* continua subindo */
               j--;
         }
   }
}

void geravet(int v[], int k) {
  /* gera vetor com k elementos usando rand() */
   int i;
   srand(9999);
   for (i = 0; i < k; i++) v[i] = rand()%1000;
}

void impvet(int v[], int k) {
   /* imprime vetor com k elementos */
   int i;
   for (i = 0; i < k; i++) printf("%5d", v[i]);
}

int main() {
  int vet[1000], n, kk, x;
  /* ler n */
  printf("entre com n:");
  scanf("%d", &n);
```

```
  /* gera o vetor e imprime */
  geravet(vet, n);
  printf("\n****** vetor gerado ******\n");
  impvet(vet, n);
  /* ordena o vetor */
  bubble(vet, n);
  /* imprime vetor ordenado */
  printf("\n****** vetor ordenado ******\n");
  impvet(vet, n);
 }
```

Nada de novo na saida do programa.

```
entre com n:100

****** vetor gerado ******
  691  995  720  855  388  105  163  288  657  851  499  771  920  363   73    3
  514  633  815  153  899  757  934  575  387  841  748  872  726  818  172   62
  473  640   93  724  734  620  136   86  370  610  740  421  605  119   84  631
  217  740  580  286  974  436   40  335  526  923  918  511  973  535  892  945
  115  515  269   43   93  929  984  330  688  488  961  266  152   14  100  899
  264  344   56    8   50  200   88  274  554  135  976   20  998  435  966  592
  209  658  158  643
****** vetor ordenado ******
    3    8   14   20   40   43   50   56   62   73   84   86   88   93   93  100
  105  115  119  135  136  152  153  158  163  172  200  209  217  264  266  269
  274  286  288  330  335  344  363  370  387  388  421  435  436  473  488  499
  511  514  515  526  535  554  575  580  592  605  610  620  631  633  640  643
  657  658  688  691  720  724  726  734  740  740  748  757  771  815  818  841
  851  855  872  892  899  899  918  920  923  929  934  945  961  966  973  974
  976  984  995  998
```

Exercícios

Considere as 120 (5!) permutações de 1 2 3 4 5:

1) **Encontre algumas que precisem exatamente de 5 trocas para classificá-la pelo método bubble.**
2) **Idem para 7 trocas**
3) **Qual a sequência que possui o número máximo de trocas e quantas trocas são necessárias?**

Classificação - método bubble (da bolha) – análise simplificada

Quantas vezes o comando `troca(a[j], a[j-1])` é executado?
No pior caso, quando a sequência está invertida, é executado i vezes para cada valor de i = 1, 2, 3, ..., n-1. Portanto 1 + 2 + 3 + ... + n – 1 = n.(n – 1)/2.

O método bubble é $O(n^2)$ no pior caso.
Será que no caso médio também é $O(n^2)$?
Vamos calcular o número médio de inversões de uma sequência.

Inversões

Seja P = a_1 a_2 ... a_n, uma permutação de 1 2 ... n.
O par (i,j) é uma inversão quando i<j e a_i>a_j.
Exemplo: 1 3 5 4 2 tem 4 inversões: (3,2) (5,4) (5,2) e (4,2)

No método bubble o número de trocas é igual ao número de inversões da sequência.
O algoritmo se resume a:

```
for (i = 1; i < n; i++) {
   /* elimine as inversões de a[i] até a[0] */
   . . .
}
```

Veja também o exemplo:

```
6 8 4 2 5 – elimine as inversões do 8 – 0
6 8 4 2 5 – elimine as inversões do 4 – 2
4 6 8 2 5 – elimine as inversões do 2 – 3
2 4 6 8 5 – elimine as inversões do 5 – 2
2 4 5 6 8
```

Total de 7 inversões que é exatamente a quantidade de inversões na sequência.

Para calcularmos então o número de trocas do bubble, basta calcular o número de inversões da sequência.
É equivalente a calcular o número de inversões de uma permutação de 1 2 ... n.

Qual o número médio de inversões em todas as sequências possíveis?
É equivalente a calcular o número médio de inversões em todas as permutações de 1 2 ... n.
Pensando de maneira recursiva, cada permutação de n-1 elementos gera n outras permutações de n elementos, pela introdução de n em todas as posições possíveis.

Se uma dada permutação P de n-1 elementos tem k inversões, as n permutações geradas por esta terá:
(k+0) + (k+1) + (k+2) + ... + (k+n-1) = n.k + n.(n-1)/2

Seja então I(P) o número de inversões na permutação P de n elementos.
Seja $J_n(P)$ o total de inversões em todas as permutações possíveis de n elementos.
$J_n(P) = \sum I(P)$ (P permutação de n elementos)
$J_n(P) = \sum n.I(P)+n.(n-1)/2$ (P permutação de n-1 elementos)
$J_n(P) = n.\sum I(P) + [n.(n-1)/2].(n-1)! = n.J_{n-1}+(n-1).n!/2$
$= n.[(n-1) J_{n-2}+(n-2).(n-1)!/2] + (n-1).n!/2$
$= n.(n-1) J_{n-2} + (n-2).n!/2 + (n-1).n!/2$
= . . .
$= n! J_0 + n.(n-1)/2 . n!/2 = n.(n-1).n!/4$

Esse é o total de inversões.

Como há n! permutações, o número médio é J_n/n! ou seja n.(n-1)/4.

Portanto o número médio de trocas no bubble é n.(n-1)/4.
Note que é exatamente a média entre o mínimo e o máximo.
Ou seja, continua sendo $O(n^2)$.

Classificação – método da inserção

O método da inserção é uma variação do bubble.
A idéia é pegar cada um dos elementos a partir do segundo e abrir um espaço para ele (inserir), deslocando de uma posição para baixo, todos os maiores que estão acima. Observe que é até um pouco melhor que trocar com o vizinho de cima.

```
void insercao(int a[], int n) {
   int i, j, x;
   /* todos a partir do segundo */
   for (i = 1; i < n; i++) {
      x = a[i]; /* guarda a[i] */
      /* desloca os elementos necessários para liberar um lugar para a[i] */
      for (j = i - 1; j >= 0  && a[j] > x; --j) a[j + 1] = a[j];
      /* coloca x no lugar correto */
      /* observe o valor de j ao sair do for acima */
      a[j + 1] = x;
   }
}
```

A análise é a mesma que o bubble.

Classificação - método shell

Um pouco melhor que bubble.

A idéia é tentar eliminar inversões de maneira mais rápida.
Repete-se então o bubble com uma sequência de passos: n/2, n/4, ... 1.
Quanto o passo é 1, temos o próprio bubble. Porém, neste momento a sequência já está com menos inversões. Portanto teremos menos trocas.

Veja um exemplo numa sequência de 5 elementos. Vamos executar o bubble com passos 2 e 1:

```
h=2          h=1
6   4   4    4   2   2
8   8   2    2   4   4
4   6   6    5   5   5
2   2   8    8   8   6
5   5   5    6   6   8
```

Observe que tivemos 5 trocas. No caso do bubble para esta mesma sequência tivemos 7.

A implementação abaixo é parecida com o bubble

```
void shell(int a[], int n) {
   int i, j, h;
   h = n/2;
   while (h > 0) {
      /* tentar subir com a[i], i = h, h+1, ..., n-1 */
      for (i = h; i < n; i++) {
         /* suba com a[i] até encontrar um menor ou chegar em a[0] */
         j = i;
         while (j >= h && a[j] < a[j-h]) {
               /* troca a[j] com a[j-h] */
               troca(a[j], a[j-h]);
               /* continua subindo com passo h */
               j = j-h;
         }
    h = h/2; /* novo h */
   }
}
```

Outra variação parecida com o método da inserção:

```
void shell(int a[], int n) {
   int i, j, h, v;
   h = n/2;
   while (h > 0) {
      /* deslocar os elementos a partir de a[i], i = h, h+1, ..., n-1 */
      for (i = h; i < n; i++) {
         /* abrir espaço para a[i] */
         j = i; v = a[i];
         while (j >= h && v < a[j-h]) {
              a[j] = a[j-h];
               /* continua subindo com passo h */
               j = j-h;
         }
         a[j] = v;
      }
      h = h/2; /* novo h */
   }
}
```

Classificação - método shell – análise

A análise do método shell depende do passo h que é usado.
É uma análise bastante complexa e não existe uma análise geral para ao mesmo.
Existem alguns resultados parciais que sugerem que o shell é melhor que $O(n^2)$.
Alguns resultados parciais:

1) O método shell faz menos que $O(N^{3/2})$ quando se usa a sequência 1, 4, 13, 40, 121, ... ou seja, a sequência $h_i = 3.h_{i-1}+1$. Esse resultado é de Knuth-1969.

2) O método shell faz menos que $O(N^{4/3})$ quando se usa a sequência 1, 8, 23, 77, 281, ... ou seja, a sequência $4^{i+1} + 3.2^{i} + 1$, para i=0,1,2, ...

3) A sequência 1, 2 , 4, 8, ... proposta por Shell-1959 não é a melhor sequência. Em alguns casos pode levar ao tempo $O(N^2)$. Com esta sequência de passos, os elementos de posições pares não são comparados com elementos de posições ímpares. Considere o caso em que temos a metade dos elementos menores nas posições ímpares e a metade dos elementos maiores nas posições pares. Exemplo – sequência de 16 elementos.

```
1 9 2 10 3 11 4 12 5 13 6 14 7 15 8 16
```

Veja o que acontece com os passos 8, 4 e 2. Nada é feito. Tudo fica para o passo 1. (bubble).

A versão abaixo usa a sequência sugerida por Knuth (primeiro caso).

```
void shell(int a[], int n) {
   int i, j, h;

   /* descobre o primeiro h */
   for (h=1; h<=n/9; h=3*h+1);

   while (h > 0) {
      /* deslocar os elementos a partir de a[i], i = h, h+1, ..., n-1 */
      for (i = h; i < n; i++) {
         /* abrir espaço para a[i] */
         j = i; v = a[i];
         while (j >= h && v < a[j-h]) {
               a[j] = a[j-h];
               /* continua subindo com passo h */
               j = j-h;
         }
         a[j] = v;
      }
      h = h/3; /* novo h */
   }
}
```

Classificação - método Merge

Considere o seguinte problema. Dados dois vetores **a** e **b** de **n** e **m** elementos já ordenados, construir outro vetor **c** de **m+n** elementos também ordenado com os elementos de **a** e **b**.

Uma primeira solução seria colocar em **c** os elementos de **a**, seguidos dos elementos de **b** e ordenar o vetor **c**. Existe forma melhor, que é fazer a intercalação (merge) dos elementos de **a** e **b** em **c**. Vejamos:

a = 2 5 8 9
b = 1 3 4
c = 1 2 3 4 5 8 9

Basta pegar o menor entre **a[i]** e **b[j]** e colocar em **c[k]**. A cada passo incrementar **i** ou **j** e **k**.

```
void intercala (int a[], int b[], int c[], int n, int m) {

 int i = 0, j = 0, k = 0;
 while (k < n + m)
    if (i == n) {c[k] = b[j]; j++; k++;} /* ou c[k++] = b[j++] */
    else if (j == m) {c[k] = a[i]; i++; k++;} /* idem */
         else if (a[i] < b[j]) {c[k] = a[i]; i++; k++;} /* idem */
              else {c[k] = b[j]; j++; k++;} /* idem */
}
```

Outra forma:

```
void intercala (int a[], int b[], int c[], int n, int m) {

   int i = 0, j = 0, k = 0;
   while (i < n && j < m)
      if (a[i] <= a[j]) c[k++] = a[i++];
      else c[k++] = b[j++];
   /* basta agora mover todos os de a ou todos os de b */
   while (i < n) c[k++] = a[i++];
   while (j < m) c[k++] = b[j++];
}
```

Agora suponhamos que temos dois trechos contíguos do mesmo vetor (**a[p..q-1]** e **a[q..r-1]**) em ordem crescente e queremos intercalar esses dois trechos, obtendo um só trecho (**a[p..r-1]**) ordenado. Vamos usar um vetor auxiliar **aux** com o mesmo número de elementos de **a**. Para intercalar vamos também usar o mesmo trecho em **aux** ou seja **aux[p..r-1]**.

```
void intercala(int a[], int p, int q, int r) {
    int aux[max];
    int i = p, j= q, k = p;
    while (i < q && j < r)
       if (a[i] <= a[j]) aux[k++] = a[i++];
       else aux[k++] = a[j++];
    /* basta agora mover os que sobraram de a[p..q-1] ou de a[q..r-1] */
    while (i < q) aux[k++] = a[i++];
    while (j < r) aux[k++] = a[j++];
    /* move de volta para a */
    for (i = p; i < r; i++) a[i] = aux[i];
}
```

O vetor **aux**, pode ser alocado dinamicamente (usando **malloc**), porém quando **p** é zero e **r = max – 1**, ele terá o tamanho de **a**. Note que nos algoritmos anteriores, não era necessário memória auxiliar para classificar o vetor. O método merge precisa de memória auxiliar no pior caso igual ao tamanho do próprio vetor a ser classificado.

A função **intercala** acima em qualquer dos casos tem complexidade linear.
O(n+m) é o mesmo que O(2.n) que é o mesmo que O(n).

Exercício – achar alguma outra forma de fazer a função intercala acima.

Com a função intercala, podemos construir um algoritmo interessante de ordenação. O algoritmo é recursivo e divide a sequência em duas metades, estas em outras duas metades, etc., até que cada sequência tenha 0 ou 1 elementos, caso em que nada é feito. Após isso, intercala as sequências vizinhas.

A função mergesort abaixo classifica o vetor **a[p], a[p+1], ... , a[r-1]** pelo método mergesort.
A chamada inicial **mergesort(a, 0, n)** solicita a classificação de toda sequência: **a[0], a[1],..., a[n-1]**.

```
void mergesort(int a[], int p, int r) {
   /* verifica se a sequência tem mais de 1 elemento */
   if (p < r - 1) {
       /* acha o elemento médio */
       q = (p+r) / 2;
       /* chama recursivamente mergesort para as 2 metades */
       mergesort(a, p, q);
       mergesort(a, q, r);
       /* intercala as 2 sequências anteriores já ordenadas */
       intercala(a, p, q, r)
   }
}
```

Classificação - método Merge – análise simplificada

No caso da função **intercala**, não tem comparações ou trocas para contarmos a quantidade. No entanto, o tempo necessário para fazer a intercalação de duas sequências com um total de **n** elementos é proporcional a **n**. Digamos **c.n**, onde **c** é uma constante.

O tempo necessário para fazer o mergesort de **n** elementos **T(n)**, é o tempo necessário para fazer o merge de **2** sequências de **n/2** elementos mais o tempo necessário para se fazer a intercalação de **n** elementos. Ou seja

$$T(n) = 2.T(n/2) + c.n$$
$$T(n) = 2.(2.T(n/4)+c.n/2)+c.n = 4.T(n/4) + 2.c.n$$
$$T(n) = 4.(2.T(n/8)+c.n/4)+2.c.n = 8.T(n/8)+3.c.n$$
$$....$$
$$T(n) = 2^k.T(n/2^k)+k.c.n$$

Supondo $n = 2^k$, e portanto $k = lg(n)$ (base 2). Se **n** não for potência de **2**, considere como limitante superior o menor **n'** maior que **n** que seja desta forma.

$$T(n) = n.T(1)+lg(n).c.n = n.(1+c.lg(n))$$

Assim, **`T(n)`** é proporcional a **`n.lg(n)`** ou **`O(n.log(n))`**.

Observe que nos primeiros algoritmos o tempo é proporcional a n^2.

Classificação - método Quick

Existem algoritmos que particionam uma sequência em duas, determinando um elemento pivô, tal que todos à esquerda são menores e todos à direita são maiores. Com isso usa-se a mesma técnica do Merge, isto é, aplica-se o algoritmo recursivamente na parte esquerda e na parte direita.

O algoritmo abaixo trabalha com dois apontadores. O apontador **`i`** começa com o primeiro elemento enquanto que o apontador **`j`** começa com o último. Apenas um deles é modificado a cada comparação (**`i++`** ou **`j--`**). Quando **`A[i] < A[j]`** trocam-se os elementos, fixa-se o apontador e inverte-se o apontador que estava sendo modificado. O processo continua até que **`i`** fique igual a **`j`**. Quando isso acontece, temos a seguinte situação:

`A[k] < A[i]` para **`k=0,1,...,i-1`**
`A[k] >= A[i]` para **`k=i+1,...,n-1`**
`A[i]` já está em seu lugar definitivo.

Veja a seguinte sequência de passos:

```
i             j
→             ←
5 3 9 7 2 8 6 (Avança i)
5 3 6 7 2 8 9 (6 é o maior da esquerda e todos menores que 9. Avança j)
5 3 2 7 6 8 9 (6 é o menor da direita e todos maiores. Avança i)
5 3 2 6 7 8 9 (6 é o maior da esquerda e todos menores. Avança j)
              (i ficou igual a j. 6 já está em seu lugar; todos da esquerda
               são menores; todos da direita são maiores; os da direita já estão em
               ordem neste exemplo mas é coincidência)
              (Vamos repetir o mesmo para esquerda e direita)
5 3 2         (Avança i)
2 3 5         (Avança j. 2 está no lugar)
  3 5         (Avança i. 5 já está no lugar)
        7 8 9 (Avança i. 9 já está no lugar)
        7 8   (Avança i. 8 já está no lugar)
2 3 5 6 7 8 9 (tudo no lugar)

Outro exemplo:

6 3 9 2 5 4 1 8 7 (avança i)
6 3 7 2 5 4 1 8 9 (todos a esquerda de i menores que j;avança j)
6 3 1 2 5 4 7 8 9 (todos a direita de j maiores que todos menores que i; avança i)
                  (i=j no 7; todos a esquerda menores e todos a direita maiores)

6 3 1 2 5 4       (avança i)
```

```
4 3 1 2 5 6        (avança j)
2 3 1 4 5 6        (avança i - i fica igual a j no 4)

2 3 1              (avança i)
1 3 2              (avança j - i fica igual a j no 1)

  3 2              (avança i)
  2 3              (avança j - i fica igual a j no 2)

              8 9  (avança i - i fica igual a j no 9)
```

A sequência ficou classificada.

Vejamos primeiro a função particiona.

```
int particiona(int a[], int ini, int fim) {

   /* devolve k tal que a[k] já está em sua posição correta */
   /* a[m] < a[k] para m = ini..k-1 e a[m] >= a[k] para m = k+1..fim */
   int i = ini, j = fim, dir;
   dir = 1; /* fixa j e incrementa i */
   while (i < j) {
       if (a[i] > a[j]) {
           troca(a[i], a[j]);
           dir = - dir; /* inverte a direção */
       }
       /* incrementa i ou decrementa j */
       if (dir == 1) i++; else j--;
   }
   return i;
}
```

Exercício – existem algumas variações para a função `particiona`. Tente achar alguma.

Agora o Quick recursivo, fica parecido com o Merge.
A primeira chamada é **`quicksort(a, 0, n-1)`**.

```
void quicksort(int a[], int ini, int fim) {
   int k;
   if (ini < fim) {
       k = particiona(a, ini, fim);
       quicksort(a, ini, k-1);
       quicksort(a, k+1, fim);
   }
}
```

Classificação – método Quick - versão não recursiva

A recursão é usada só para guardar a ordem das subsequências que devem ser classificadas. Podemos em vez de recursão usar uma pilha para guardar as tais subsequências.

Usando uma pilha com vetores int:

```
static int pilhaE[MAX], pilhaD[MAX], topo;

void emp(int x, int y) {
   topo++;
   pilhaE[topo] = x; pilhaD[topo] = y;
}

void des(int *x, int *y) {
   *x =  pilhaE[topo];
   *y =  pilhaD[topo];
   topo--;
}

void quick(int a[], int n) {
   int i, f, k;
   topo=-1; /* inicia pilha */
   emp(0, n-1);  /* empilha primeira subsequência */
   /* repete enquanto tem algum elemento na pilha */
   while (topo >= 0) {
      des(&i, &f);
      k = particiona(a, i, f);
      if (i<k-1) emp(i, k-1); /* empilha parte esquerda */
      if (k+1<f) emp(k+1, f); /* empilha parte direita */
   }
}
```

Classificação - método Quick – análise simplificada

O processo de partição do vetor é proporcional a n. O problema é saber quantas partições serão necessárias. Supondo que cada partição é dividida em 2 de igual tamanho, teremos um caso análogo ao do mergesort, isto é o tempo é proporcional a n.log(n). No pior caso, cada partição é dividida em uma partição de 1 e outra de n-1 elementos. Porém no caso médio, onde a partição pode estar em qualquer lugar a proporcionalidade é com n.log(n).

Somente para efeito de análise vamos supor que a sequência vai de a[1] até a[n].
Seja Q(n) o número médio de trocas para classificar n elementos.
No primeiro passo são necessárias n-1 trocas no máximo.
A partição pode ocorrer em qualquer ponto de 1 a n.

$Q(n) <= n +(1/n).\sum [Q(I-1)+Q(I)] \ (I=1,n)$

O multiplicador 1/n ocorre porque supomos a partição acontecendo em qualquer dos n elementos com a mesma chance.

$\sum [Q(I-1)+Q(I)] \ (I=1,n) =$
$Q(0)+Q(n-1)+$
$Q(1)+Q(n-2)+$

Q(2)+Q(n-3)+
. . .
Q(n-1)+Q(0) = 2. ∑ Q(I) (I=0,n-1)

Q(n) <= n+(2/n). ∑ Q(I) (I=0,n-1)

Vamos mostrar por indução em n que se n>=2, Q(n) <= 2.n.ln(n)
(n=2) Q(2) <= 2+2/2 . (Q(0)+Q(1)) = 2. E 2 <= 2.2ln(2).
(supondo para k<n) Q(n) <= 2.k.ln(k)
(provar para n):
Como Q(n) <= n + (2/n). ∑ Q(I) (I=0,n-1) = n+(2/n).[Q(0)+Q(1)+ ∑ Q(I) (I=2,n-1)] <=
<= n +(2/n). ∑ 2.I.ln(I) (I=2,n-1) = n +(4/n). ∑ I.ln(I) (I=2,n-1)

∑ I.ln(I) (I=2,n-1) <= ∫ x.ln(x) (x=2,n-1) <= ∫ x.ln(x) (x=0,n) =
= (x^2/2).ln(x)-x^2/4 (x=0,n) = (n^2/2).ln(n)-n^2/4

Portanto Q(n) <= n +(4/n). [(n^2/2).ln(n)-n^2/4] = n+2.n.ln(n)-n = 2.n.ln(n)

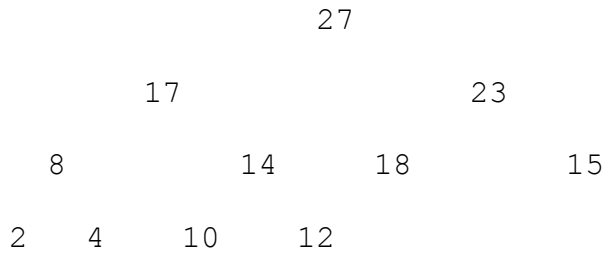
Finalmente, o Quick é O(n.log(n))

Merge e Quick

Observe agora que o algoritmo que particiona a sequência, não usa um vetor auxiliar como no caso do algoritmo de intercalação de sequências ordenadas. Isso torna o Quick melhor para sequências muito grandes, pois menos memória é usada.

Classificação - método Heap

Uma árvore binária é um Heap se e só se cada nó que não é folha armazena um elemento maior que seus filhos. A árvore binária abaixo é Heap.



Podemos armazenar uma árvore binária num vetor, percorrendo os níveis:

índice	1	2	3	4	5	6	7	8	9	10	11
valor	27	17	23	8	14	18	15	2	4	10	12

Cada nó **`i`** tem filhos **`2i`** e **`2i+1`** e **`a[i] >= a[2i]`** e **`a[i] >= a[2i+1]`**.

As folhas estão a partir do elemento **`a[n/2+1]`** até **`a[n]`**.

Para o caso do algoritmo Heap (Heapsort) vamos considerar o vetor **`a[1..n]`** e não **`a[0..n-1]`**. O que em C, implica em declarar o vetor com um elemento a mais que o máximo e ignorar **`a[0]`**.

Abaixo a função **`Heap(a, k, n)`** que arruma os filhos de **`a[k]`**, os netos, bisnetos, etc . destes até o final do vetor;

A função Heap - versão recursiva

Nessa versão, basta verificar os dois filhos e chamar Heap recursivamente para esses filhos:

```
void Heap(int a[], int k, int n) {
   /* compara o filho esquerdo */
   if (2*k <= n && a[k] < a[2*k]) {
       /* troca com o pai e aplica Heap ao novo filho */
       troca(a[k], a[2*k]);
       Heap(a, 2*k, n);
   )
   /* compara com o filho direito */
   if (2*k+1 <= n && a[k] < a[2*k+1]) {
       /* troca com o pai e aplica Heap ao novo filho */
       troca(a[k], a[2*k+1]);
       Heap(a, 2*k+1, n);
   )
}
```

A função Heap - versão não recursiva

```
void Heap(int a[], int k, int n) {
   /* verifica a situação Heap com os filhos, netos, etc . . */
   while (2*k <= n) {
       j = 2*k;
       /* decide o maior entre a[j] e a[j+1] (filhos) */
       if (j < n && a[j] < a[j+1]) j++;
       /* neste ponto a[j] é o maior entre os 2 filhos */
       /* verifica se deve trocar com o pai */
       if (a[j] > a[k]) troca(a[j], a[k]);
       else break; /* retorna pois nada mais a fazer */
       /* se trocou pode ser que a troca introduziu algo não Heap */
       /* verifica portanto o filho que foi trocado */
       k = j;
   }
}
```

Classificação - método Heap

Note agora que se a tabela é Heap e **a[1]** é o maior. Portanto, para ordenar a tabela:

a) troca-se **a[1]** com **a[n]**
b) **a[n]** já está em seu lugar
c) aplica-se o Heap em **a[1],** pois é o único que está em posição errada
d) voltar para a) com uma tabela de **n-1** elementos

```
void Heapsort(int a[], int n) {
   int k;
   /* aplica o Heap aos elementos acima da metade */
   for (k = n/2; k >=1; k--) Heap(a, k, n);
   /* a[1] é o maior. Troca com o último que já fica em seu lugar. */
   /* aplica Heap em a[1] numa tabela com 1 elemento a menos */
   while (n > 1) {
       troca(a[1], a[n]);
       Heap(a, 1, --n);
   }
}
```

Classificação - método Heap – análise simplificada

A função Heap acima, a partir de **k**, verifica **a[2k]** e **a[2k+1]**, **a[4k]** e **a[4k+1]** e assim por diante, até **a[2^ik]** e **a[2^ik + 1]** até que estes índices sejam menores ou iguais a **n**. Portanto o tempo é proporcional a **log n** (base 2).

A função **Heapsort** acima chama a **Heap n/2+n = 3n/2** vezes (o **for** é executado **n/2** vezes e o **while** é executado **n** vezes)
Dessa forma, o tempo é proporcional a **3n/2.log(n)**, ou seja proporcional a **n.log(n)**. Assim, o método Heap é **O(n.log(n))**.

Classificação - Exercícios

1. Ordene a sequência abaixo pelos métodos Merge, Quick, Heapsort e Bubble. Em cada um dos casos diga qual o número de comparações e de trocas feitas para a ordenação.

12 23 5 9 0 4 1 12 21 2 5 14

2. Encontre permutações de (1, 2, 3, 4, 5) que ocasionem:

a. número de trocas máximo no QUICKSORT
b. número de trocas máximo no BUBBLESORT.

3. Dado um vetor V de N elementos, em ordem decrescente, escreva um algoritmo que ordena esse vetor em ordem crescente.

- Quantas trocas foram necessárias?

4. Em cada situação abaixo discuta sobre qual algoritmo de ordenação usar:

a. Um vetor de inteiros de tamanho <= 8.
b. Uma lista de nomes parcialmente ordenada.
c. Uma lista de nomes em ordem aleatória.
d. Uma lista de inteiros positivos distintos menores que 100.
e. Um arquivo que não cabe na memória principal.

5. Sejam os seguintes procedimentos de classificação de um vetor a[0..n-1]:

```
void ordena(int a[], int n)
  {int i, j, imin, x;
   for(i = 0, i < n; i++)
       {imin:= i;
        for (j= i+1; j < n; j++)
   (I)      if (a[j] < a[imin]) ...........(II)
                imin:= j;
         troca(a[i], a[imin]); ....(III)
       }
   }

void class (int a[], int n)
   {int i, j, x;
    for (i = 2; i < n; i++)
        for(j = i; j == 1; j--)
   (I)      if (a[j] < a[j-1]) ...........(II)
              troca(a[j], a[j-1]); ....(III)
    }
```

Para cada um dos algoritmos responda às seguintes perguntas:

a. O comando (I) é executado iterativamente para i=1,2,...n-2 em ordena e para i=2,3,..., n-1 em class. Ao final da i-ésima iteração qual é a situação do vetor a ? Justifique.

b. Baseado em sua resposta, justifique que cada um dos procedimentos realmente ordena o vetor a.

c. Qual o número máximo e mínimo de vezes que a comparação (II) é executada e em que situações ocorre.

d. Idem para o comando (III) que troca elementos.

6. Faça um procedimento recursivo que ordena uma sequência A de n>1 elementos baseado no seguinte algoritmo:

Obtenha um número p, 1 <= p <= n. Divida a sequência em duas partes, a primeira com p elementos e a segunda com n-p. Ordene cada uma das duas partes. Após isso, supondo que:

a [1] <= a [2] <= ... <= a [p]
e a [p+1] <= a [p+2] <= ... <= a [n]

Intercale as duas sequências de forma a obter o vetor A ordenado (pode usar vetores auxiliares).

7. Considere o algoritmo da questão 6 e os seguintes valores de p:

(i) 1
(ii) n / 3
(iii) n / 2

a. Você reconhece o algoritmo com algum dos valores de p ?
b. Qual das três escolhas sugeridas para p é a mais eficiente?
c. Por quê ?

8. Encontre uma permutação de 1,2,3,4,5 em quem sejam necessárias 5 ou mais trocas para ordená-la pelo método Quick.

9. Seja a1,a2,...,an uma permutação qualquer de 1,2,...n. Chama-se inversão a situação em que i<j e ai>aj. Assim, para n=3 a permutação 231 tem 2 inversões.

Considere a permutação a1 a2...ai...aj...an onde ai>aj.

A permutação a1 a2...aj...ai...an pode ter mais inversões que a anterior?

10. Justifique porque no algoritmo Bubble o número de trocas necessárias para ordenar a sequência coincide com o número de inversões.