Disciplina: BC1424 - Algoritmos e Estruturas de Dados I
1^o quad/2013 – **Prof^a.** Letícia Rodrigues Bueno

Projeto: Comparação empírica do tempo de execução dos algoritmos de ordenação Inserção, Seleção, Bolha, *Mergesort*, *Heapsort* e *Quicksort*.
Projeto proposto pelo professor André Balan

1 Instruções

- **Entrega até** 14/07/2013 no TIDIA: AED1-2013.1-Leticia
- **Pode ser feito em dupla ou individual.**
- Faça na linguagem C ou C++.
- Cada um dos algoritmos de ordenação deve ser um método que recebe um vetor de inteiros e o número n de elementos no vetor. Protótipos dos métodos:

```
void bubble_sort(int V, int n);
void insertion_sort(int V, int n);
void selection_sort(int V, int n);
void quick_sort(int V, int n);
void merge_sort(int V, int n);
void heap_sort(int V, int n);
```

1.1 Metodologia

- Cada algoritmo de ordenação deverá ser testado com vetores de elementos do tipo inteiro com as seguintes quantidades de elementos: 10.000, 30.000, 90.000, 270.000, 810.000, 2.430.000, 7.290.000, 21.870.000, 65.610.000.
- Os vetores que serão fornecidos como entrada aos algoritmos devem ser gerados aleatoriamente, com distribuição uniforme, usando a função

`rand()` da biblioteca `stdlib.h`. O comando `v[i]=rand();` gera um inteiro entre 0 e 2.147.483.647.

- Para cada tamanho de vetor, o programa deve rodar cada método de ordenação com 6 sequências aleatórias diferentes. Assim, o número de ordenações que você deve executar é: (6 algoritmos) $\times$ (9 tamanhos) $\times$ (6 sequências) = 324 ordenações.

- Para garantir uma comparação justa, em cada "rodada" de ordenações, todos os algoritmos devem ordenar a mesma sequência aleatória para que a comparação seja justa. Para gerar a mesma sequência aleatória em lugares diferentes do código, você deve usar a mesma semente (seed):

```
// criamos uma sequência
int seed=4;
srand(seed);
for(int i=0; i<n; i++) V[i]=rand();

// criamos uma sequência igual à anterior
srand(seed);
for(int i=0; i<n; i++) V[i]=rand();
```

- Neste trabalho, você deve criar 6 sequências pseudoaleatórias usando as seguintes sementes:

```
seed[0]=4;
seed[1]=81;
seed[2]=151;
seed[3]=1601;
seed[4]=2307;
seed[5]=4207;
```

- Seu programa também deve realizar uma rodada de ordenações com uma sequência de 50.000 elementos em ordem inversa.

- **Atenção:** o Quicksort pode falhar ao processar grandes sequências com um erro de stack overflow. Para que o Quicksort não estoure a pilha de execução, modifique o método particiona para ser aleatório, como vimos em sala de aula. Essa versão do Quicksort deve ser usada também em todas as demais execuções.

- Para cada ordenação, o programa deve medir o tempo de execução em segundos. Ou seja, cada par (método, tamanho do vetor) terá 6 medições de tempo: uma para cada sequência pseudoaleatória. Como medir o tempo?

```
time_t segundos = time(NULL);
// processamento que terá o tempo medido
int tempo_decorrido = time(NULL) - segundos;
cout << tempo_decorrido << "\n";
```

- Após preparar o programa, chegou a hora de colocar pra rodar e ir dormir. Deixe apenas seu programa rodar. Não faça nada no computador enquanto o experimento estiver rodando (não mexa nem o mouse).

- O programa deve lhe retornar todos os dados necessários para você realizar o estudo empírico. Pode ser necessário gravar os dados em arquivo:

```
// escreve arquivo de saída
void writeResults(int n, int tempo){
    // abre arquivo e aponta onde próxima linha será escrita
    ofstream myfile;
    // append the text at the final position of the file
    myfile.open("ordenacao.txt",ios::app);
    myfile << "Método: " << metodo << "\n";
    myfile << "Tempo: " << tempo << "\n \n";
    myfile.close();
}
```

1.2 Resultados

- Com todos os dados obtidos, você deve elaborar um relatório contendo:
  - gráficos que permitam uma comparação fácil e rápida entre os métodos;
  - análise dos gráficos;
  - um gráfico com a média dos tempos gastos por cada método para as seis diferentes sequências aleatórias testadas.

- É provável que os métodos ineficientes (Bolha, Seleção e Inserção) irão desbalancear as comparações. Assim, você decidir se estes métodos devem ser colocados em gráficos separados.

- Você pode decidir também um timeout para cada experimento (vamos combinar 20 minutos) para que não demore a vida inteira para rodar o projeto todo. Assim, se algum método excedeu 20 minutos ao tentar ordenar um vetor de um certo tamanho, você não precisa rodar o experimento com outros vetores desse tamanho ou maiores!

- Para cada par (método, tamanho do vetor) que não foi executado por completo, estimar o quanto ele demoraria para rodar e justificar sua estimativa.
- Entregar o código fonte e o relatório em pdf no TIDIA até 14/07/2013.

1.3 Códigos dos Algoritmos de Ordenação

1.3.1 Inserção

```
void insertion_sort(int V[ ],int n){
    int key;
    int i;
    for (int j=1; j < n; j++){
        key = V[j];
        i = j-1;
        while ((i>=0) && (V[i]>key)){
            V[i+1] = V[i];
            i=i-1;
        }
        V[i+1] = key;
    }
}
```

1.3.2 Seleção

```
void selection_sort(int V[ ], int n){
    int key;
    int index;
    for (int i=0; i < n; i++){
        key = V[i];
        index = i;
        for (int j = i+1; j < n; j++){
            if (V[j]<key){
                key = V[j];
                index = j;
            }
        }
        V[index]= V[i];
        V[i] = key;
    }
}
```

1.3.3 Bolha

```
void bubble_sort(int V[ ], int n){
    int key;
    int pass=1;
    bool ordenado=false;

    while ((pass<n) && (ordenado==false)){
        ordenado=true;
        for (int i=0; i < n-pass; i++){
            if (V[i]>V[i+1]){
                key = V[i];
                V[i] = V[i+1];
                V[i+1] = key;
                ordenado=false;
            }
        }
        pass=pass+1;
    }
}
```

1.3.4 Mergesort

```
void merge(int V[ ], int ini, int meio, int fim){
    int p = meio-ini+1;
    int m = fim-meio;
    int L[fim], R[fim];

    for (int i=0; i<p; i++) L[i] = V[ini+i];

    for (int j=1; j<=m; j++) R[j-1] = V[(meio+j)];

    L[p] = INT_MAX;
    R[m] = INT_MAX;
    int i = 0;
    int j = 0;
    for (int k=ini; k<=fim; k++){
        if (L[i]<=R[j]){
            V[k] = L[i];
            i = i+1;
        } else {
            V[k] = R[j];
            j = j+1;
        }
    }
}

void mergesort2(int V[ ], int ini, int fim){
    if (ini<fim){
        int meio = floor((ini+fim)/2);
        mergesort2(V,ini,meio);
        mergesort2(V,meio+1,fim);
        merge(V,ini,meio,fim);
    }
}

void merge_sort(int V[ ], int n){
    mergesort2(V,0,n-1);
}
```

1.3.5 Heapsort

```
void Heapify(int V[ ], int i, int NHeap){
    int le, ri, maior, aux;
    le = (2*i);
    ri = ((2*i)+1);
    if ((le<NHeap) && (V[le]>V[i]))
        maior = le;
    else maior = i;

    if ((ri<NHeap) && (V[ri]>V[maior]))
        maior = ri;

    if (maior != i){
        aux = V[i];
        V[i] = V[maior];
        V[maior] = aux;
        Heapify(V,maior,NHeap);
    }
}

void HeapBuild(int V[ ], int n){
    for(int i=floor(n/2); i>=0; i--)
        Heapify(V,i,n);
}

void heap_sort(int V[ ], int n){
    int aux;
    int NHeap=n;
    HeapBuild(V,n);
    for(int i=(n-1); i>=0; i--) {
        aux = V[0];
        V[0] = V[i];
        V[i] = aux;
        NHeap = NHeap-1;
        Heapify(V,0,NHeap);
    }
}
```

1.3.6 Quicksort

```
int particao(int V[ ], int p, int r){
    int aux;
    int pivo = V[r];
    int i = p-1;
    for (int j=p; j <= r-1; j++){
        if (V[j] <= pivo){
            i=i+1;
            aux = V[i];
            V[i] = V[j];
            V[j] = aux;
        }
    }
    aux = V[i+1];
    V[i+1] = V[r];
    V[r] = aux;
    return (i+1);
}

void quicksort2(int V[ ], int p, int r){
    if (p<r){
        int q = particao(V,p,r);
        quicksort2(V,p,q-1);
        quicksort2(V,q+1,r);
    }
}

void quick_sort(int V[ ], int n){
    quicksort2(V,0,n-1);
}
```