CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS

BRUNO NASCIMENTO DAMACENA

TRABALHO PRÁTICO 2

Trabalho apresentado à disciplina de Laboratório de Algoritmos e Estrutura de Dados I do curso de Engenharia de Computação do Departamento de Computação do CEFET-MG

Professora Natália Cosse Batista.

Belo Horizonte
Novembro de 2017

Implementação e Análise de Algoritmos de Ordenação por Comparações de Chaves

Neste trabalho, você deverá implementar algoritmos não-eficientes (BubbleSort, InsertionSort e SelectionSort) e eficientes (QuickSort, HeapSort e MergeSort) para ordenar sequências de elementos.
Os seis algoritmos foram implementados seguindo a implementação do Ziviani, disponíveis em
http://www2.dcc.ufmg.br/livros/algoritmos/cap4/codigo/c/4.1a4.7e4.14-ordenacao.c

BubbleSort

```
void BubbleSort(TipoItem *A, TipoIndice n){
  TipoIndice i,j;
  int contador;
  for(j=0;j<n-1;j++){
    contador=0;
    for(i=1;i<n-j;i++){
      if(A[i].Chave > A[i+1].Chave){
        TipoItem aux = A[i];
        A[i] = A[i+1];
        A[i+1] = aux;
        contador++;
      }
    }
    if(contador==0) return;
  }
}
```

Custo computacional:
O(n) para o melhor caso (vetor já ordenado), pois ele percorre o vetor uma vez fazendo comparações, vê que não houve troca e encerra.
O (n^2) para o pior caso (vetor inversamente ordenado), pois ele percorre o vetor e realiza todas as trocas, contabilizando $\frac{n(n-1)}{2}$ trocas.
Para o caso médio, o custo computacional é O (n^2).

InsertionSort

```
void InsertionSort(TipoItem *A, TipoIndice n){
  TipoIndice i, j;
  TipoItem x;
  for (i = 2; i <= n; i++){
    x = A[i];  j = i - 1;
    A[0] = x;
    while (x.Chave < A[j].Chave){
      A[j+1] = A[j];
      j--;
    }
    A[j+1] = x;
  }
}
```

Custo computacional:
O(n) para o melhor caso (vetor já ordenado), pois ele percorre o vetor uma vez, vê que todos os elementos já estão em seu lugar correto, e encerra.
O (n^2) para o pior caso (vetor inversamente ordenado), pois ele percorre o vetor e realiza todas as trocas, contabilizando $\frac{n(n-1)}{2}$ trocas.
Para o caso médio, o custo computacional é O (n^2).

SelectionSort

```
void SelectionSort(TipoItem *A, TipoIndice n){
  TipoIndice i, j, Min;
  TipoItem x;
  for (i = 1; i <= n - 1; i++){
    Min = i;
    for (j = i + 1; j <= n; j++){
      if (A[j].Chave < A[Min].Chave) Min = j;
    }
    x = A[Min];
    A[Min] = A[i];
    A[i] = x;
  }
}
```

Custo computacional:
Para todos os casos, o custo do SelectionSort é O (n^2), pois independente da disposição do vetor de entrada, ele faz $\frac{n(n-1)}{2}$ comparações.

QuickSort

```
void Particao(TipoIndice Esq, TipoIndice Dir,
TipoIndice *i, TipoIndice *j, TipoItem *A){
  TipoItem x, w;
  *i = Esq;  *j = Dir;
  x = A[(*i + *j) / 2];
  do{
    while(x.Chave > A[*i].Chave) (*i)++;
    while(x.Chave < A[*j].Chave) (*j)--;
    if (*i <= *j){
      w = A[*i];
      A[*i] = A[*j];
      A[*j] = w;
      (*i)++;
      (*j)--;
    }
  }while (*i <= *j);
}

void OrdenaQuick(TipoIndice Esq, TipoIndice Dir,
TipoItem *A){
  TipoIndice i, j;
  Particao(Esq, Dir, &i, &j, A);
  if (Esq < j) OrdenaQuick(Esq, j, A);
  if (i < Dir) OrdenaQuick(i, Dir, A);
}

void QuickSort(TipoItem *A, TipoIndice n){
  OrdenaQuick(1, n, A);
}
```

Custo computacional:
O custo desse algoritmo depende da escolha do pivô. Uma escolha ruim de pivô, como em uma das extremidades do vetor, gera um custo O (n^2). Agora, para o melhor caso e o caso médio, o custo é O (n log n).

HeapSort

```
void Refaz(TipoIndice Esq, TipoIndice Dir, TipoItem *A){
  TipoIndice i = Esq;
  int j;
  TipoItem x;
  j = i * 2;
  x = A[i];
  while (j <= Dir){
    if (j < Dir){
      if (A[j].Chave < A[j+1].Chave)
      j++;
    }
    if (x.Chave >= A[j].Chave) break;
    A[i] = A[j];
    i = j;  j = i * 2;
  }
  A[i] = x;
}

void Constroi(TipoItem *A, TipoIndice n){
  TipoIndice Esq;
  Esq = n / 2 + 1;
  while (Esq > 1)
    { Esq--;
      Refaz(Esq, n, A);
    }
}

void HeapSort(TipoItem *A, TipoIndice n){
  TipoIndice Esq, Dir;
  TipoItem x;
  Constroi(A, n);
  Esq = 1;  Dir = n;
  while (Dir > 1){
      x = A[1];  A[1] = A[Dir];  A[Dir] = x;  Dir--;
      Refaz(Esq, Dir, A);
    }
}
```

Custo computacional:
O custo desse algoritmo é O (n log n) para todos os casos.

MergeSort

```
void Merge(TipoItem *A, TipoIndice i, TipoIndice m,
TipoIndice j){
  TipoVetor B;
  TipoIndice x;
  TipoIndice k = i;
  TipoIndice l = m+1;

  for(x=i;x<=j;x++) B[x] = A[x];
  x = i;
  while(k<=m && l<=j){
    if(B[k].Chave <= B[l].Chave) A[x++] = B[k++];
    else A[x++] = B[l++];
  }

  while (k<=m) A[x++] = B[k++];

  while (l<=j) A[x++] = B[l++];

}
void OrdenaMerge(TipoItem *A, TipoIndice comeco,
TipoIndice fim){
  TipoIndice meio;
  if (comeco < fim){
    meio = (comeco + fim - 1) / 2;
    OrdenaMerge(A, comeco, meio);
    OrdenaMerge(A, meio + 1, fim);
    Merge(A, comeco, meio, fim);
  }
}

void MergeSort(TipoItem *A, TipoIndice n){
  OrdenaMerge(A,1,n);
}
```

Custo computacional:
O custo desse algoritmo é O (n log n) para todos os casos.

Foi utilizado no programa um gerador de entrada, que são vetores de tamanhos variados de dez a um milhão, de tipos aleatórios, ordenados, ordenados inversamente e quase ordenados. O programa principal percorreu todos esses vetores, ordenou utilizando todos os seis algoritmos e guardou o tempo de execução de cada. No final, ele exibiu também o tempo de execução total do algoritmo. Informações sobre a saída do algoritmo (tempo dado em milissegundos):

Aleatório	BubbleSort	InsertionSort	SelectionSort	QuickSort	HeapSort	MergeSort
10	0	0	0	0	0	0
100	0	0	0	0	0	0
1000	3	1	1	0	0	0
10000	263	56	114	1	1	1
100000	26968	5602	11372	13	16	15
1000000	2731965	563906	1144972	156	215	187

Ordenado	BubbleSort	InsertionSort	SelectionSort	QuickSort	HeapSort	MergeSort
10	0	0	0	0	0	0
100	0	0	0	0	0	0
1000	0	0	1	0	0	0
10000	0	0	117	0	1	0
100000	0	0	11566	4	11	9
1000000	4	5	1168947	46	143	109

Quase Ordenado	BubbleSort	InsertionSort	SelectionSort	QuickSort	HeapSort	MergeSort
10	0	0	0	0	0	0
100	0	0	0	0	0	0
1000	0	0	2	0	0	0
10000	0	0	161	0	1	0
100000	0	0	11416	4	11	9
1000000	14	8	1156607	47	140	109

Inversamente Ordenado	BubbleSort	InsertionSort	SelectionSort	QuickSort	HeapSort	MergeSort
10	0	0	0	0	0	0
100	0	0	0	0	0	0
1000	2	1	1	0	0	0
10000	220	115	110	0	1	0
100000	21894	11383	10834	4	12	9
1000000	2258175	1195519	1141833	48	150	113

Considerações:

Devida as melhorias implementadas no BubbleSort, ele se mostrou muito bom para os casos de vetor ordenado e quase ordenado. Porém, mesmo assim, ele foi o pior algoritmo em tempo de execução. Como esperado, o InsertionSort foi muito bom para vetores ordenados.
Na média, os algoritmos eficientes tiveram um desempenho muito melhor do que os não eficientes, levando mais de meia hora de diferença no tempo de vetores muito grandes. Ou seja, para n grandes, é totalmente inviável o uso de algoritmos de ordem quadrática, já que o tempo vai de 40 minutos a 100 milissegundos.

Referências Bibliográficas

http://www2.dcc.ufmg.br/livros/algoritmos/cap4/codigo/c/4.1a4.7e4.14-ordenacao.c
https://sites.google.com/site/nataliacefetmg/home/laed1_20172/Mergesort.c
https://pt.wikipedia.org/wiki/Bubble_sort
https://pt.wikipedia.org/wiki/Insertion_sort
https://pt.wikipedia.org/wiki/Selection_sort
https://pt.wikipedia.org/wiki/Quicksort
https://pt.wikipedia.org/wiki/Heapsort
https://pt.wikipedia.org/wiki/Merge_sort

Instruções de como utilizar o programa

Observação: para melhor funcionamento do programa, este deve ser compilado e executado em Linux, e eu irei ensinar a fazê-lo neste SO. Porém, uma vez que você tenha os algoritmos e uma IDE, você também pode executá-lo no Windows.

Abra o terminal no diretório do código e digite uma das opções:

- "make" para compilar o algoritmo;
- "make entrada" para compilar o algoritmo gerador de entrada e executá-lo;;
- "make run" para compilar o algoritmo, gerar o executável e executá-lo;
- "make clean" para apagar os arquivos executáveis e os gerados pela compilação.

Essas instruções também podem ser encontradas no repositório do GitHub.