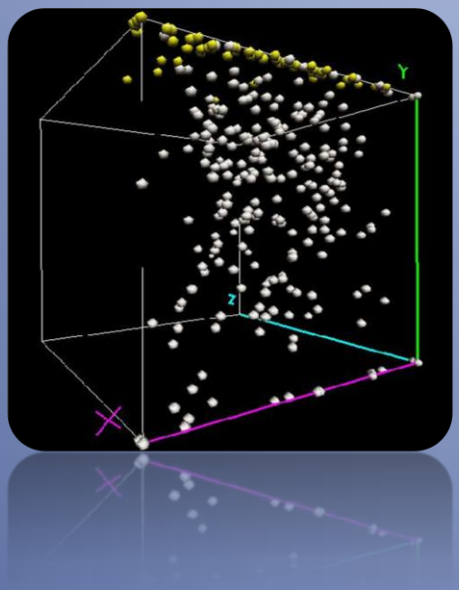

ALGORITMOS E ESTRUTURAS DE DADOS III

Tutorial 1 (usa o compilador de linguagem C Dev-C++ versão 4.9.9.2)

Parte 1 de 3 sobre o algoritmo de ordenação shell (concha) conhecido como Shellsort.

1 Introdução

Esta série de tutoriais sobre *Algoritmos e Estruturas de Dados III* foi escrita usando o **Microsoft Windows 7 Ultimate**, **Microsoft Office 2010**, **Bloodshed Dev-C++** versão 4.9.9.2 (pode ser baixado em http://www.bloodshed.net), referências na internet e notas de aula do professor quando estudante. Ela cobre desde os algoritmos de ordenação, passando pela pesquisa em memória primária e culminando com a pesquisa em memória secundária.

Nós entendemos que você já conhece o compilador Dev-C++. No caso de você ainda não o conhecer, dê uma olhada nos tutoriais Dev-C++ 001 a 017, começando pelo Tutorial Dev-C++ - 001 - Introdução.

Se não tem problemas com a linguagem C/C++ e o compilador Dev-C++, então o próximo passo é saber ler, criar e alterar arquivos em disco usando linguagem C/C++. Se ainda não sabe como fazê-lo, dê uma olhada nos tutoriais Dev-C++ 001 e 002, começando pelo Tutorial Dev-C++ 001 – Criação, Leitura e Alteração de Arquivos.

Se sabe todas as coisas anteriores, então a próxima etapa é conhecer os algoritmos mais básicos de ordenação. Em minhas notas de aula você encontra um material básico, porém detalhado e com algoritmos resolvidos, dos principais métodos de ordenação existentes.

Adotaremos o livro ***Projeto de Algoritmos com Implementação em Pascal e C***, Editora Cengage Learning, de Nivio Ziviani, como livro-texto da disciplina. Nele você encontrará os métodos de ordenação que iremos estudar.

Seu próximo passo será estudar os algoritmos de ordenação por **Inserção** e por **Seleção**. Você pode usar os links anteriores (em inglês) ou fazer uso do livro-texto.

Se você seguiu todos os passos até aqui, está pronto para prosseguir com este tutorial.

2 O Algoritmo de Ordenação Shellsort

Criado por Donald Shell em 1959, publicado pela Universidade de Cincinnati, Shellsort é o mais eficiente algoritmo de classificação dentre os de complexidade quadrática. É um refinamento do método de inserção direta (se não conhece, releia a introdução deste tutorial). O algoritmo difere do método de inserção direta pelo fato de no lugar de considerar o vetor a ser ordenado como um único segmento, ele considera vários segmentos sendo aplicado o método de inserção direta em cada um deles. Basicamente o algoritmo passa várias vezes pela lista dividindo o grupo maior em menores. Nos grupos menores é aplicado o método da ordenação por inserção.

2.1 Uma Olhada no Funcionamento do Algoritmo

Note como o tamanho dos anéis de uma concha (*shell*) vão ficando cada vez menores à medida que se aproximam do centro. Assim também é o funcionamento do algoritmo Shellsort.

Uma vez calculado um determinado número de posições a comparar, separadas e equidistantes, inicialmente com grande separação, à medida que o algoritmo avança e as trocas são feitas, esta separação diminui para um terço, até que a separação seja apenas de 1 item.

Explicando melhor, se houver um vetor de 14 posições a ordenar, primeiramente calcula-se um número inicial de posições a comparar. Por exemplo, digamos que o número calculado seja 13. Assim, o algoritmo vai comparar os itens 1º e 14º (1 + 13), ordenando-os caso necessário. Em seguida, o salto, que era 13, é dividido por 3, passando a ser 4. Agora o algoritmo vai comparar os itens 1º, 5º (1 + 4), 9º (5 + 4) e 13º (9 + 4), ordenando-os caso necessário. Em seguida, o salto, que era 4, é dividido por 3, passando a ser 1. O algoritmo prossegue, comparando os itens 1 a 1, desde o 1º, fazendo a ordenação final.

Como ocorre na concha natural, os espaços (*gaps*) vão diminuindo à medida que ocorre a ordenação.

Vamos dar uma olhada na implementação do algoritmo Shellsort na próxima listagem, no item 2.2 e no item 2.3 veremos uma explicação detalhada com exemplo real. Programas completos no item 3.

Ainda não se sabe porque este método é eficiente, mas ninguém ainda conseguiu analisar o algoritmo, por conter problemas matemáticos muito difíceis. Sabe-se que a sequência de incrementos não deve ser múltipla. Para a sequência de incrementos da listagem 1, existem duas conjeturas para o número de comparações, a saber:

Conjetura 1: $C(n) = O(n^{1,25})$

Conjetura 2: $C(n) = O(n(\ln n)^2)$

Shellsort é ótimo para arquivos de tamanho moderado, uma vez que sua implementação é simples e requer pouco código. Existem métodos mais eficientes, mas mais complexos de implementar. O tempo de execução é sensível à ordem inicial dos dados. É um algoritmo **não-estável**.

2.2 Uma Olhada na Implementação do Algoritmo

Listagem 1:

```
void shellSort(int *vet, int size) {
  int i, j, value;
  int gap = 1;
  do {
    gap = 3 * gap + 1;
  } while(gap < size);
  do {
    gap /= 3;
    for(i = gap; i < size; i++) {
      value = vet[i];
      j = i - gap;
      while(j >= 0 && value < vet[j]) {
        vet [j + gap] = vet[j];
        j -= gap;
      }
      vet [j + gap] = value;
    }
  } while(gap > 1);
}
```

2.3 Um Exemplo Passo-a-passo

O Shellsort divide o vetor em várias partes e muitas vezes.

Funciona assim:

A primeira coisa que ele faz é pegar o tamanho dos "pulos" (*gap*, em inglês) para "montar" diversos vetores de menor tamanho (subvetores dentro do vetor inicial), através do trecho (em azul, na listagem 1):

```
do {
  gap = 3 * gap + 1;
} while(gap < size);
```

Imagine um vetor de 10 posições:

7 2 5 6 3 1 0 8 9 4

Supondo que "gap" = 3, então teremos as seguintes posições selecionadas:

7 2 5 **6** 3 1 **0** 8 9 **4**

Assim, 7 6 0 4, será um dos subvetores para ordenar. O que aconteceu aqui? "gap" é o tamanho dos pulos para selecionar o primeiro vetor. (A cada três posições ele pega um elemento.)

Ele então ordena esse vetor, que ordenado fica:

0 4 6 7

Entretanto, ele faz isso dentro do vetor original, que ficará assim, após a primeira passagem pelo algoritmo:

0 2 5 **4** 3 1 **6** 8 9 **7**

Agora ele repete as operações acima só que ao invés de iniciar da posição 0, ele iniciará da posição 1, ficando selecionados:

0 **2** 5 4 **3** 1 6 **8** 9 7

Ele ordena esse novo vetor, ficando:

0 **2** 5 4 **3** 1 6 **8** 9 7

Começa então pela posição 2:

0 2 **5** 4 3 **1** 6 8 **9** 7

Ordenando ficará:

0 2 **1** 4 3 **5** 6 8 **9** 7

Pronto! Ele terminou a primeira parte da ordenação. Repare que agora o vetor está semi-ordenado. E tudo o que foi feito até agora, encontra-se nessa parte do código (em vermelho na listagem 1):

```
for(i = gap; i < size; i++) {
  value = vet[i];
  j = i - gap;
  while(j >= 0 && value < vet[j]) {
    vet [j + gap] = vet[j];
    j -= gap;
  }
  vet [j + gap] = value;
```

}

Agora ele divide o gap por 3, ficando igual a 1 e repete a operação anterior.

Até o momento o vetor está assim:

0 2 1 4 3 5 6 8 9 7

Agora vamos repetir a operação anterior com gap = 1, selecionando:

0 2 4 3 5 6 8 9 7

Ordenando a partir da posição 0:

0 2 4 3 5 6 8 9 7

0 2 **3 4** 5 6 8 9 7

0 2 3 4 5 6 8 9 7

0 2 3 4 5 6 8 **7 9**

Ele repete a operação a partir da posição 1:

0 2 3 4 5 6 7 8 9

Ele continua repetindo o trecho em vermelho da listagem 1, a partir da posição 2, posição 3 e assim por diante.

Finalmente o algoritmo para, pois a comparação **while(gap > 1)**, após o trecho em vermelho da listagem 1, já no final do algoritmo, é falsa.

Exercício de fixação
Faça o *teste de mesa* para um vetor que tenha os valores 12, 43, 1, 6, 56, 23, 52, 9. Depois confira se o resultado foi idêntico ao que se encontra aqui.

 Resista à tentação e só expie depois.

3 Implementando o algoritmo ShellSort em Linguagem C/C++

3.1 O Usuário Fornece os Números

```
#include <cstdlib>
#include <iostream>

using namespace std;

// listagem 1
void shellSort(int *vet, int size) {
    int i, j, value;
    int gap = 1;

    do {
        gap = 3 * gap + 1;
    } while(gap < size);

    do {
        gap /= 3;
        for(i = gap; i < size; i++) {
            value = vet[i];
            j = i - gap;
            while(j >= 0 && value < vet[j]) {
                vet [j + gap] = vet[j];
                j -= gap;
            }
            vet [j + gap] = value;
        }
    } while(gap > 1);
}
int main(int argc, char *argv[])
{
    int vetor[10];
    int i;

    printf("Forneca 10 numeros...\n");
    for(i=0;i<10;i++) {
        printf("Numero %2d: ", i+1);
        scanf("%d", &vetor[i]);
    }

    printf("\nVetor original: {");
    for(i=0;i<10;i++) {
        printf("%d", vetor[i]);
        if(i<9) printf(", ");
    }
    printf("}\n");

    shellSort(vetor, 10); // algoritmo da listagem 1

    printf("\nVetor ordenado: {");
    for(i=0;i<10;i++) {
        printf("%d", vetor[i]);
        if(i<9) printf(", ");
    }
    printf("}\n\n");

    system("PAUSE");
    return EXIT_SUCCESS;
}
```

3.2 O Programa Gera Uma Quantidade Fixa de Números Aleatórios

```
#include <cstdlib>
#include <iostream>
#include <time.h>

using namespace std;

const int MAX_SIZE = 250; // altere aqui a quantidade

void shellSort(int *vet, int size) {
   int i, j, value;
   int gap = 1;

   do {
      gap = 3 * gap + 1;
   } while(gap < size);

   do {
      gap /= 3;
      for(i = gap; i < size; i++) {
         value = vet[i];
         j = i - gap;
         while(j >= 0 && value < vet[j]) {
            vet [j + gap] = vet[j];
            j -= gap;
         }
         vet [j + gap] = value;
      }
   } while(gap > 1);
}

int main(int argc, char *argv[])
{
   int vetor[MAX_SIZE];
   int i;

   // nova semente para numeros aleatorios
   srand(time(NULL));

   printf("Gerando %d numeros inteiros aleatoriamente.\nAguarde...\n\n", MAX_SIZE);
   for(i=0;i<MAX_SIZE;i++) {
      // gera numeros entre 0 e 99999
      vetor[i]=rand()%100000*rand()%100000;
   }

   printf("\nVetor original:\n{");
   for(i=0;i<MAX_SIZE;i++) {
      printf("%d", vetor[i]);
      if(i<MAX_SIZE-1) printf(", ");
   }
   printf("}\n");

   shellSort(vetor, MAX_SIZE);

   printf("\nVetor ordenado:\n{");
   for(i=0;i<MAX_SIZE;i++) {
      printf("%d", vetor[i]);
      if(i<MAX_SIZE-1) printf(", ");
   }
   printf("}\n\n");

   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Maximize a tela de saída e use a barra de rolagem vertical para ver a saída completa do programa.

3.3. O Programa Gera Uma Quantidade Qualquer de Números Aleatórios Solicitada Pelo Usuário

Deixarei este a cargo de vocês. Minha sugestão, para simplificar, é fazer um grande vetor, digamos, um milhão de inteiros, e ter uma variável que controle o tamanho máximo dos dados que o usuário desejar. Forneça os valores de forma aleatória.

4 Terminamos

Terminamos por aqui. Clique no menu Arquivo, depois clique na opção Sair.

Corra para o próximo tutorial.