Notas de Aula - Algoritmos e Estrutura de Dados I (AE42CP) – Algoritmos de Ordenação (parte 2)
Prof. Jefferson T. Oliva

Na aula anterior vimos dois algoritmos de ordenação simples: bollha (*bubble sort*), seleção (*selection sort*) e inserção (*insertion sort*)

Hoje veremos dois algoritmos de ordenação mais avançados: *quicksort* e *heapsort*.

Quicksort

Outro algoritmo de ordenação por troca bastante conhecido é o *quicksort*, que é baseado na estratégia de divisão e conquista:
- na parte da divisão, o conjunto X[p...q] é dividido em dois subconjuntos $X[p ... r - 1]$ e $X[r + 1 ... q]$, tais que $X[p ... r - 1] <= X[r] <= X[r + 1 ... q]$
- A conquista ocorre a partir da ordenação dos dois subconjuntos por meio de chamadas recursivas do algoritmo de ordenação

Procedimento quicksort(X, p, q)
1 - definir o pivô r e as posições i = p e j = q
2 - enquanto i ≤ j, trocar de posição os elementos maiores (lado esquerdo do arranjo) com os itens menores (lado direito) que o pivô
3 - quicksort(X, p, j)
4 - quicksort(X, i, q)

Implementação:

```
void quicksort(int x[], int esq, int dir){
        int i = esq, j = dir, pivo = x[(i + j) / 2], aux;

        do{
                while (x[i] < pivo)
                        i++;
                while (x[j] > pivo)
                        j--;
                if (i <= j){
                        aux = x[i];
                        x[i] = x[j];
                        x[j] = aux;
                        i++;
                        j--;
                }
        }while (i <= j);
        if (j > esq)
                quicksort(x, esq, j);
        if (i < dir)
                quicksort(x, i, dir);
}
```

Neste código, o valor do pivô é definido pela posição do meio entre i e j. Em seguida é executado o laço do-while, enquanto i for menor igual a j

Ver slides de 7 a 9

Outra versão do quicksort:

```
void troca(int *a, int *b) {
   int aux = *a;
   *a = *b;
   *b = aux;
}


static int particionar(int v[], int esq, int dir) {
   int pivo = v[esq];
   int i = esq + 1;
   int j = dir;

   while (i <= j) {
      while ((v[i] <= pivo) && (i <= dir))
         i++;

      while ((v[j] > pivo) && (j >= esq))
         j--;

      if (i < j) {
         troca(&v[i], &v[j]);
      }
   }

   troca(&v[esq], &v[j]);

   return j;
}


void quicksort2(int v[], int esq, int dir){
   if (esq < dir) {
      int j = particionar(v, esq, dir);
      quicksort2(v, esq, j - 1);
      quicksort2(v, j + 1, dir);
   }
}
```

Ver slides de 11 a 13

O algoritmo *quicksort* possui a complexidade na ordem de O(n^2) no pior caso, mas no melhor e no caso médio, a complexidade é O(*n log_{2} n)*.

O pior caso de particionamento ocorre quando o elemento pivô divide a lista de forma desbalanceada, ou seja, divide a lista em duas sub listas: uma com tamanho 1 e outra com tamanho n - 1 (no qual n se refere ao tamanho da lista original).

Heapsort

Baseado no princípio de ordenação por seleção em árvore binária.

O método consiste em duas fases distintas:
1 - Montagem da árvore binária (HEAP)
2 - Seleção dos elementos na ordem desejada

ver slide 18

Em uma heap
→ O sucessor à esquerda do elemento de índice i é o elemento de índice 2 * (i + 1) − 1, se 2 * (i + 1) − 1 < n, caso contrário não existe
→ O sucessor à direita do elemento de índice i é o elemento de índice 2 * (i + 1), se 2 * (i + 1) < n, caso contrário não existe

Código para rearranjar um vetor para que o mesmo atenda a condição para ser uma heap

```
static void gerarHeap(int v[], int n){
        int esq = n / 2;

        // Para o vetor ser rearranjado como heap, o processo começa
        // pelos "nós folhas" e a troca pode ocorrer até o "nó raiz"
        while (esq >= 0){
                refazer(v, esq, n - 1);
                esq--;
        }
}
```

```
static void refazer(int v[], int esq, int dir){
        int j = (esq + 1) * 2 - 1; // posição de um nó descendente (esquerda)
                                   // do nó localizado na posição esq
        int x = v[esq]; // representa o nó raiz, ou seja, o elemento a partir
                        // do qual será testada a condição de heap

        // Pa partir do nó na posição esq, o arranjo é percorrido até o
        // "nó folha"
        while (j <= dir){
                // É verificado se o "nó filho" esquerdo é menor que o direito
                if ((j < dir) && (v[j] < v[j + 1]))
                        j++;

                // Se o x for maior que o seu descendente, a condição de heap
                // não foi violada
                if (x >= v[j])
                        break;

                // Quando a condição de heap é violada, devem haver trocas
                // de posições entre os elementos
                v[esq] = v[j];
                esq = j; // posição de um dos descendentes do elemento localizado
                        // em esq
                j = (esq + 1) * 2 - 1; // descendente "esquerdo" de v[esq]
        }

        v[esq] = x;
}
```

O fazheap tem o propósito de rearranjar o vetor para que o mesmo atenda a condição para ser uma heap. Esse processo é iniciado na primeira metade do arranjo. Assim, como se fosse em uma árvore binária, a primeira troca poderá entre um nó folha e o seu respectivo pai. Esse processo segue até chegar ao “nó raiz”, que seria o primeiro elemento do vetor. Para a construção da heap, as movimentações são realizadas na função refazheap para impor a condição de heap no arranjo.

No refazheap, os parâmetros são o vetor, a posição de um nó da heap e a posição do último elemento do vetor. No método, primeiramente é definida a posição de um nó descendente (esquerda) do nó localizado na posição esq. Em seguida, na variável x é atribuído o valor do arranjo na posição esq, que será o “nó raiz” (da “árvore” (esq = 0) ou “sub-árvore” (esq > 0)). A partir desse “nó”, são explorados os seus respectivos descendentes (no laço while). Dentro do laço while é verificado qual descendente é maior: v[j] ou v[j + 1]. Assim, na variável j será armazenada a posição do descendente com maior valor, o qual será explorado na próxima “rodada” de operações do laço while. Caso a “raiz” explorada seja maior que um descendente (v[j]), não fará sentido a continuação do laço while, pois o maior elemento está na “raiz”. Caso contrário, v[esq] receberá v[j], ou seja, o maior valor será posicionado no “nó raiz” explorado. Em seguida, o próximo “nó raiz” passará a ser o elemento na posição j (esq passará a receber j) e j receberá o valor da posição de um dos descendentes no novo “nó raiz”. Esse laço while é processado até um nó maior que os

seus descendentes imediatos sejam encontrado ou alcançar um “nó folha”. Por fim, v[esq] receberá o menor valor processado (x). Esse método parece não ser eficiente, mas ele é usado apenas durante a geração da heap e a aplicação da ordenação. Para a geração da heap, o processamento começa a partir do nível mais baixo da “árvore”, ou seja, a partir dos nós folhas e os seus respectivos “nós raizes”. O processo é repetido até chegar ao processamento do “nó raiz” (v[0]), no qual, sempre o maior valor estará posicionado.

Função principal

```
void heapsort(int v[], int n){
        int x;
        int dir = n - 1;

        gerarHeap(v, n);

        while (dir > 0){
                x = v[0];
                v[0] = v[dir];
                v[dir] = x;
                dir--;

                refazer(v, 0, dir);
        }
}
```

Na função heapsort, o primeiro passo é a geração da heap. Em seguida, no laço while é o elemento “raiz” do heap (v[0]) é posicionado em sua respectiva posição no arranjo (v[dir]). Após, a variável dir é decrementada para atualizar a posição onde a “raiz” da heap deverá ser colocada na próxima “rodada” do loop. Por fim, a heap é ajustada para manter o arranjo entre 0 e dir em condições de heap. O loop é executado até a ordenação completa da estrutura.

A primeira vista, o algoritmo aparenta não ser eficiente por causa da remoção do maior elemento da heap, pois o arranjo deve ser manipulado para manter a condição de heap. Mas, o algoritmo é rápido.

Heap sort não é um algoritmo de ordenação estável.

Esse algoritmo não é recomendado para pequenos conjuntos de dados, pois neste caso, um algoritmo simples é mais que suficiente.

Por fim, o custo de tempo é logarítmico: O (n log n)

Shell sort

Extensão do insert sort

Contorna o principal problema do insertion sort possibilitando troca de registros que estão distantes um do outro.

Tem como objetivo aumentar o passo de movimento dos elementos ao invés das posições adjacentes.

Consiste em classificar sub-arranjos do original.

Esses sub-arranjos contêm todo h-ésimo elemento do arranjo original.

O valor de h é chamado de incremento.

Por exemplo, se h é 5, o sub-arranjo consiste dos elementos x[0], x[5], x[10], etc
- Sub-arranjo 1: x[0], x[5], x[10]
- Sub-arranjo 2: x[1], x[6], x[11]
- Sub-arranjo 3: x[2], x[7], x[12]
- Sub-arranjo 4: x[3], x[8], x[13]

Após a ordenação dos sub-arranjos:
- Define-se um novo incremento menor que o anterior
- Gera-se novos sub-arquivos
- Aplica-se novamente o método da inserção

O processo é realizado repetidamente até que h seja igual a 1

O valor de h pode ser definido de várias formas, por exemplo
- $h(s) = 3h(s - 1) + 1$, para $s > 1$
- $h(s) = 1$, para $s = 1$

Implementação

```
void shellsort(int v[], int n){
        int h = 1;
        int x, i, j;

        while (h < n)
                h - 3 * h + 1;

        h /= 3;

        while (h >= 1){
                for (i = h; i < n; i++){
                        x = v[i];
                        j = i;

                        while ((j >= h) && (x < v[j - h])){
                                v[j] = v[j - h];
                                j -= h;
                        }

                        v[j] = x;
                }

                h /= 3;
        }
}
```

Ver exemplo no slide 32.

Um problema com o shell sort ainda não resolvido é a escolha dos incrementos que fornecem os melhores resultados.

É desejável que ocorra o maior número possível de interações entre as diversas cadeias.

Ótima opção para arquivos de tamanho moderado.

Tempo de execução sensível à ordem dos dados.

O algoritmo não é estável.

Custo de tempo é estimado entre: O(n ^{1,25}) e O(n^2)

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. Third edition, The MIT Press, 2009.

Horowitz, E., Sahni, S. Rajasekaran, S. Computer Algorithms. Computer Science Press, 1998.

Rosa, J. L. G. Métodos de Ordenação. SCE-181 - Introdução à Ciência da Computação II. Slides. Ciência de Computação. ICMC/USP, 2018.

Ziviani, N. Projeto de Algoritmos - com implementações em Java e C++. Thomson, 2007.