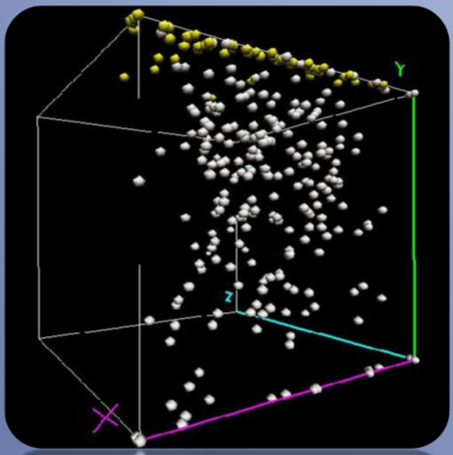


ALGORITMOS E ESTRUTURAS DE DADOS III

Tutorial 4 (usa o compilador de linguagem C Dev-C++ versão 4.9.9.2)

Parte 1 de 3 sobre o algoritmo de ordenação quick (rápido) conhecido como Quicksort.

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**, **Seleção** e **Shellsort**. 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 QUICKSORT

O Quicksort, como o Mergesort, é baseado em uma estratégia de dividir para conquistar e é um dos algoritmos de ordenação mais populares. O Quicksort é baseado no método de ordenação por trocas.

O algoritmo Quicksort pode ser dividido nos seguintes passos:

1. O *array* $A[p..r]$ é subdividido em dois *arrays* $A[p..q]$ e $A[q+1..r]$ não vazios tal que cada elemento de $A[p..q]$ é menor ou igual a cada elemento de $A[q+1..r]$. O índice q é calculado como parte deste particionamento.
2. Os dois *subarrays* $A[p..q]$ e $A[q+1..r]$ são ordenados por recursivas chamadas do Quicksort.

Por exemplo, dado o *array* **f e d h a c g b**, e tomando o valor "*d*" para partição, o primeiro passo do Quicksort rearranja o *array* da seguinte forma:

Início: **f e d h a c g b**

Antes *f* e *e* são maiores que *d* e estão à sua esquerda;
a, *c* e *b* são menores que *d* e estão à sua direita.

Passo 1: **b c a d h e g f**

Agora *b*, *c* e *a* são menores que *d* e estão à sua esquerda;
h, *e*, *g* e *f* são maiores que *d* e estão à sua direita.

Esse processo é então repetido para cada seção – isto é, "b c a" e "h e g f".

A vantagem é a não utilização de memória auxiliar.

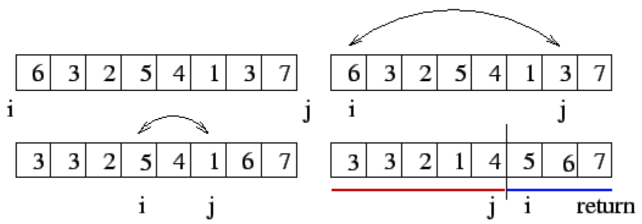
2.1 UMA OLHADA NO FUNCIONAMENTO DO ALGORITMO

Essa implementação seleciona o valor intermediário para particionar o *array* (elemento *pivot*). Para uma ordenação ótima, deveria ser selecionado o valor que estivesse precisamente no centro da faixa de valores. Porém, isso não é fácil para a maioria do conjunto de dados.

Listagem 1:

```
void qs(int *A, int esquerda, int direita) {
    register int i, j;
    int x, y;

    i = esquerda;
    j = direita;
    // Elemento intermediário como "pivot"
    x = A[(esquerda + direita) / 2];

    do {
        while((A[i] < x) && (i < direita)) i++;
        while((A[j] > x) && (j > esquerda)) j--;
        if(i <= j){
          swap(A, i, j);
          i++;
          j--;
        }
    } while(i <= j);

    if(esquerda < j) qs(A, esquerda, j);
    if(i < direita) qs(A, i, direita);
}
```

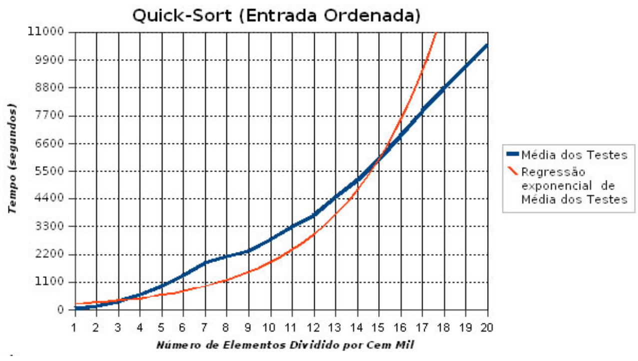
A figura abaixo ilustra o funcionamento do Quicksort (parte em **vermelho** da *listagem 1*):



pivot = 5 (elemento intermediário)

O tempo de execução do Quicksort varia conforme a partição é balanceada[1] ou não. Se a partição é balanceada o Quicksort roda tão rápido quanto o Mergesort, com a vantagem que não precisar de *array* auxiliar.

O pior caso do Quicksort ocorre quando a partição gera um conjunto com 1 elemento e outro com *n*-1 elementos para todos os passos do algoritmo. Sua desvantagem ocorre quando a divisão do arranjo (que é baseada em comparação) fica muito desbalanceada (isto ocorre quando o arranjo está quase ordenado/desordenado ou está completamente ordenado/desordenado), mostraremos isso no gráfico abaixo. Mesmo com essa desvantagem é provado que em média seu tempo tende a ser muito rápido [Cormen, 2002][2], por isso o nome, Ordenação-Rápida (ou Quicksort em inglês).



A figura mostra um gráfico do comportamento do Quicksort no pior caso.
Eixo Vertical: segundos de 0 à 11
Eixo Horizontal: número de elementos de 100.000 à 2.000.000.

Desde que a partição custa uma recorrência, neste caso torna-se

$$T(n) = T(n-1) + \Theta(n)$$

e como $T(1) = \Theta(1)$ não é difícil mostrar que

$$T(n) = \Theta(n^2)$$

Se a partição gera dois subconjuntos de tamanho $n/2$ temos a recorrência

$$T(n) = 2 \times T\left(\frac{n}{2}\right) + \Theta(n)$$

que pelo teorema *master* nos dá

$$T(n) = O(n \ln n)$$

Pode ser mostrado que o caso médio do Quicksort é muito próximo ao caso acima, ou seja $O(n \ln n)$. Na implementação acima, o elemento *pivot* foi tomado como o elemento intermediário do conjunto. Outras escolhas podem ser obtidas para gerar um melhor particionamento. Algumas versões do Quicksort escolhem o *pivot* aleatoriamente e se demonstra que esta conduta gera resultados próximos do ideal.

Pergunta

Um algoritmo baseado em comparações pode rodar em tempo menor que $O(n \ln n)$?

Para casos especiais podemos conseguir ordenação em tempo linear. O "bucketsort" é um exemplo disto.

A resposta da pergunta anterior é **não**!

[1] Diz-se que a partição é balanceada quando após o pivoteamento, a quantidade de itens à esquerda e à direita do *pivot* são iguais.

[2] Você pode encontrar uma edição mais recente em Th.H. Cormen, Ch.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms, 3rd edition*, MIT Press, 2009

Considere um algoritmo de ordenação arbitrário baseado em comparações. Podemos assumir, sem perda de generalidade, que todas as comparações feitas pelo algoritmo são do tipo $a_i \leq a_j$.

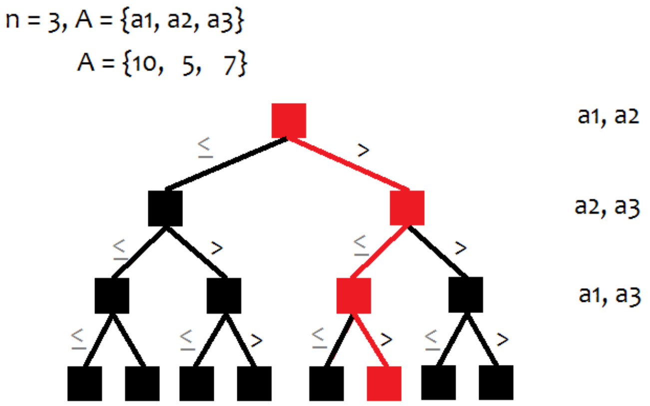
A árvore de decisão desse algoritmo é uma árvore binária cujos vértices internos estão rotulados por pares de elementos da sequência de entrada, denotados, por exemplo, por "a_i : a_j" . As arestas de um vértice interno para os seus filhos estão rotuladas uma por ≤ e a outra por >.

O rótulo da raiz da árvore corresponde à primeira comparação efetuada pelo algoritmo. A subárvore esquerda da raiz descreve as comparações subsequentes, caso o resultado desta primeira comparação, digamos, $a_i \leq a_j$, seja verdadeiro.

Já a subárvore direita descreve as comparações caso o resultado dessa comparação seja falso.

Com isso, cada sequência **(*a_1*, . . . , *a_n*)** corresponde a um caminho da raiz até uma folha da árvore de decisão.

Exemplo: A árvore de decisão do algoritmo de inserção para n = 3 é:



Por exemplo, se a_1 = 10, a_2 = 5 e a_3 = 7, o "caminho" do algoritmo na árvore de decisão acima é o caminho **vermelho**.

Existem n! permutações de (1..n). Cada uma delas deve aparecer em alguma das folhas da árvore de decisão. Portanto, a árvore de decisão deve ter pelo menos n! folhas. Por outro lado, uma árvore binária de altura h tem no máximo 2^h folhas. Assim, se h é a altura da árvore de decisão de um algoritmo de ordenação baseado em comparações, então $2^h \geq n!$.

Sabemos, pela fórmula de *Stirling*[3], que $n! \geq \left(\frac{n}{e}\right)^n$.

Então, podemos concluir que

$$h \geq \ln n! \geq \ln\left(\frac{n}{e}\right)^n = n \ln\frac{n}{e} = n(\ln n - \ln e) = O(n \ln n)$$

Portanto, de fato, qualquer algoritmo de ordenação baseado em comparações tem complexidade de pior caso $O(n \ln n)$.

2.2 Uma Olhada na Implementação do Algoritmo

Listagem 2:

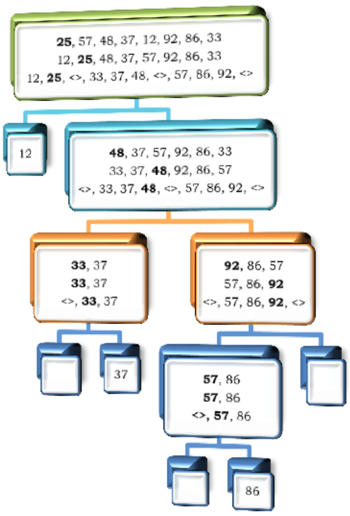
```
void swap(int* a, int* b) {
   int tmp;

   tmp = *a;
   *a = *b;
   *b = tmp;
}

int partition(int vec[], int esquerda, int direita) {
   int i, j;

   i = esquerda;
   for (j = esquerda + 1; j <= direita; ++j) {
      if (vec[j] < vec[esquerda]) {
         ++i;
         swap(&vec[i], &vec[j]);
      }
   }
   swap(&vec[esquerda], &vec[i]);

   return i;
}

void quickSort(int vec[], int esquerda, int direita) {
   int r;

   if (direita > esquerda) {
      r = partition(vec, esquerda, direita);
      quickSort(vec, esquerda, r - 1);
      quickSort(vec, r + 1, direita);
   }
}
```

[3] Em matemática, a **fórmula de Stirling** recebe o nome do matemático *James Stirling* e estabelece uma aproximação assintótica para o fatorial de um número. Na sua forma mais conhecida, ela se escreve:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

2.3 Um Exemplo Ilustrado

Para um vetor com os valores {25, 57, 48, 37, 12, 92, 86, 33}, usando a função **partition** da *listagem 2*, temos as seguintes sequências de chamadas recursivas:



ENTENDENDO O GRÁFICO A primeira linha representa o vetor original antes da chamada ao algoritmo de partição; a segunda linha representa a situação do vetor após a partição; a terceira linha representa o vetor após o retorno das chamadas recursivas. De cima para baixo, observe somente as duas primeiras linhas, depois, no retorno das chamadas (de baixo para cima), observe a terceira linha. Os valores em **negrito** representam o *pivot*. O símbolo <> representa um elemento inexistente (vetor vazio) e está na figura somente para exemplificar, não existindo de verdade.

Exercício de fixação

Faça o pivoteamento inicial usando *teste de mesa* para um vetor que tenha os valores 12, 43, 1, 6, 56, 23, 52, 9, usando a parte em vermelho da *listagem 1*, depois usando a função **partition** da *listagem 2* e finalmente, acesse o site http://www.random.org/, na caixa *True Random Number Generator* altere o valor 'Max:' para 8 e clique no botão 'Generate' até que gere um número entre 2 e 7, usando este número como valor da variável 'i' na linha 'i = esquerda;' da função **partition** da *listagem 2.* Qual dos métodos foi melhor?

2.4 Complexidade

Casos a considerar:

Melhor caso:
Verifica-se quando o vetor está sempre dividido precisamente ao meio.

Pior caso:
Uma das tabelas é vazia.

Seja

$T_p(n)$ = tempo para partir a tabela dada

$T_{qs}(i)$ = tempo para ordenar a tabela da esquerda

$T_{qs}(n-i-1)$ = tempo para ordenar a tabela da direita

Análise para o melhor caso:

$$T_{qs}(n) = T_p(n) + 2 \times T_{qs}\left(\frac{n}{2}\right)$$

$$T_{qs}(n) = c \times n + 2 \times T_{qs}\left(\frac{n}{2}\right)$$

$$T_{qs}(n) = c \times n + 2 \times c \times \left(\frac{n}{2}\right) + 4 \times T_{qs}\left(\frac{n}{4}\right)$$

$$T_{qs}(n) = c \times n + 2 \times c \times \left(\frac{n}{2}\right) + 4 \times c \times \left(\frac{n}{4}\right) + 8 \times T_{qs}\left(\frac{n}{8}\right)$$

$$T_{qs}(n) = \cdots$$

$$T_{qs}(n) = (\log_2 n) \times c \times n + 2^{\log_2 n} \times T_{qs}(1)$$

Logo: $T_{qs}(n) = \Omega(n \times \log_2 n)$

Análise para o pior caso:

$$T_{qs}(n) = T_p(n) + T_{qs}(0) + T_{qs}(n-1)$$

$$T_{qs}(n) = c \times n + T_{qs}(n-1)$$

$$T_{qs}(n) = c \times n + c \times (n-1) + T_{qs}(n-2)$$

$$T_{qs}(n) = c \times n + c \times (n-1) + c \times (n-2) + T_{qs}(n-3)$$

$$T_{qs}(n) = \cdots$$

$$T_{qs}(n) = \sum_{i=1}^{n} c \times i = c \times \sum_{i=1}^{n} i$$

$$T_{qs}(n) = c \times \frac{(n+1) \times n}{2}$$

Logo: $T_{qs}(n) = O(n^2)$

Complexidade de tempo: $\Omega(n \log_2 n)$ no melhor caso e $\Theta(n \log_2 n)$ no caso médio e $O(n^2)$ no pior caso.

Complexidade de espaço: $\Omega(\log_2 n)$ no melhor caso e $\Theta(\log_2 n)$ no caso médio e $O(\log_2 n)$ no pior caso. *R. Sedgewick* desenvolveu uma versão do Quicksort com partição por recursão de cauda que tem complexidade $O(n^2)$ no pior caso.

Estabilidade: Quicksort é um algoritmo **não estável**, uma vez que valores iguais podem ser trocados, não mantendo sua posição relativa dentro do vetor. Além do mais, é um algoritmo **destrutivo**, uma vez que não é possível mais retornar à tabela original.

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

3.1 O Usuário Fornece os Números

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

using namespace std;

// listagem 1
void qs(int *A, int esquerda, int direita) {
   register int i, j;
   int x, y;

   i = esquerda;
   j = direita;
   // Elemento intermediário como “pivot”
   x = A[(esquerda + direita) / 2];

   do {
      while((A[i] < x) && (i < direita)) i++;
      while((A[j] > x) && (j > esquerda)) j--;
      if(i <= j){
         swap(A, i, j);
         i++;
         j--;
      }
   } while(i <= j);

   if(esquerda < j) qs(A, esquerda, j);
   if(i < direita) qs(A, i, direita);
}

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");

   qs(vetor, 0, 9); // 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 swap(int* a, int* b) {
   int tmp;

   tmp = *a;
   *a = *b;
   *b = tmp;
}
```

```
int partition(int vec[], int esquerda, int direita) {
    int i, j;

    i = esquerda;
    for (j = esquerda + 1; j <= direita; ++j) {
        if (vec[j] < vec[esquerda]) {
            ++i;
            swap(&vec[i], &vec[j]);
        }
    }
    swap(&vec[esquerda], &vec[i]);

    return i;
}

void quickSort(int vec[], int esquerda, int direita) {
    int r;

    if (direita > esquerda) {
        r = partition(vec, esquerda, direita);
        quickSort(vec, esquerda, r - 1);
        quickSort(vec, r + 1, direita);
    }
}


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");

    // algoritmo da listagem 2
    quickSort(vetor, 0, MAX_SIZE - 1);

    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.

Corra para o próximo tutorial.