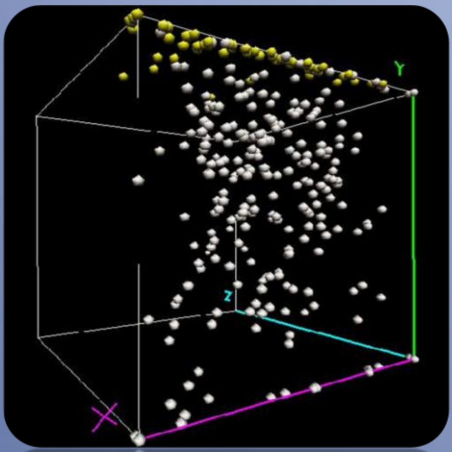


ALGORITMOS E ESTRUTURAS DE DADOS III

Tutorial 5 (usa o compilador de linguagem C Dev-C++ versão 4.9.9.2)

Parte 2 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.

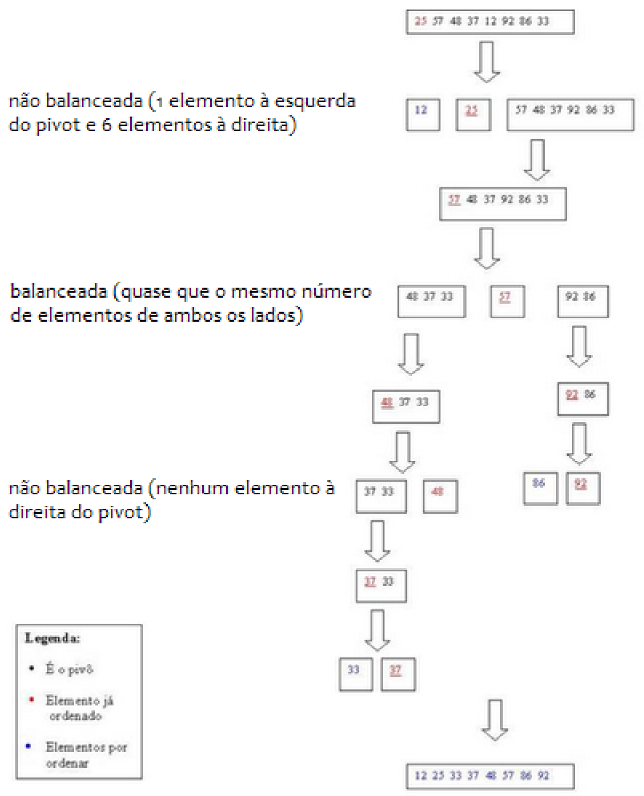
Em seguida, você precisa conhecer o algoritmo Quicksort. Para isto, você pode seguir o **Tutorial AED 004**, desta série, e/ou ler o capítulo referente no livro-texto.

Se você estiver lendo este tutorial tenha certeza de ter seguido o Tutorial AED 004. Agora que você seguiu todos os passos até aqui, está pronto para prosseguir com este tutorial.

2 O Algoritmo de Ordenação Quicksort

2.1 Análise de Complexidade

O tempo de execução do Quicksort varia conforme a partição é balanceada[1] ou não.



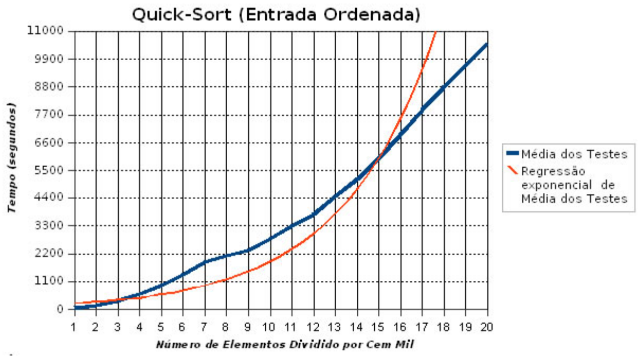
A figura mostra sucessivas chamadas recursivas do Quicksort usando como *pivot* sempre o primeiro elemento. Notar os constantes desbalanceamentos.

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).

[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



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.

Mas como podemos chegar a valores como $n \ln n$ e n^2? E o que significam as notações O (ômicron), Ω (ômega) e Θ (teta), em $O(n \ln n)$ ou $O(n^2)$?

2.1.1 O que é a Complexidade?

A Complexidade de um algoritmo consiste na quantidade de "trabalho" necessária para a sua execução, expressa em função das operações fundamentais, as quais variam de acordo com o algoritmo, e em função do volume de dados.

2.1.2 Porquê o estudo da Complexidade?

Muitas vezes as pessoas quando começam a estudar algoritmos perguntam-se qual a necessidade de desenvolver novos algoritmos para problemas que já têm solução. A performance é extremamente importante na informática pelo que existe uma necessidade constante de melhorar os algoritmos. Apesar de parecer contraditório, com o aumento da velocidade dos computadores, torna-se cada vez mais importante desenvolver algoritmos mais eficientes, devido ao aumento constante do "tamanho" dos problemas a serem resolvidos.

Devido a este fator surge a Complexidade Computacional, pois e através dela que se torna possível determinar se a implementação de determinado algoritmo é viável.

A importância da complexidade pode ser observada no exemplo abaixo, que mostra 5 algoritmos A_1 a A_5 para resolver um mesmo problema, de complexidades diferentes. Supomos que uma operação leva 1 ms para ser efetuada. A tabela seguinte dá o tempo necessário por cada um dos algoritmos.

$T_k(n)$ é a complexidade do algoritmo.

n	A_1	A_2	A_3	A_4	A_5
	$T_1(n) = n$	$T_2(n) = n \log n$	$T_3(n) = n^2$	$T_4(n) = n^3$	$T_5(n) = 2n$
16	0,016 s	0,064 s	0,256 s	4 s	1 m 4 s
32	0,032 s	0,160 s	1 s	33 s	46 d
512	0,512 s	9 s	4 m 22 s	1 d 13 h	10^{137} séc.

2.1.3 Tipos de Complexidade.

Espacial Este tipo de complexidade representa o espaço de memória usado para executar o algoritmo, por exemplo.

Temporal Este tipo de complexidade é o mais usado podendo dividir-se em dois grupos:

- Tempo (real) necessário à execução do algoritmo.
- Número de instruções necessárias à execução.

Para o estudo da complexidade são usadas três perspectivas: pior caso, melhor caso e o caso médio.

2.1.3.1 Pior Caso

Este método é normalmente representado por O( ), por exemplo se dissermos que um determinado algoritmo é representado por $g(x)$ e a sua complexidade **Pior Caso** é n, será representada por $g(x) = O(n)$.

Consiste basicamente em assumir o pior dos casos que pode acontecer, sendo muito usado e sendo normalmente o mais fácil de determinar.

Exemplo:
Se por exemplo existirem cinco baús, sendo que apenas um deles tem algo dentro e os outros estão vazios a complexidade no *pior caso* será O(5) pois eu, no pior dos casos, encontro o baú correto na quinta tentativa.

2.1.3.2 Melhor Caso
Representa-se por Ω( ).

Método que consiste em assumir que vai acontecer o melhor. Pouco usado. Tem aplicação em poucos casos.

Exemplo:
Se tivermos uma lista de números e quisermos encontrar algum deles assume-se que a complexidade **Melhor Caso** é Ω(1), pois assumimos que o numero estaria logo na cabeça da lista.

2.1.3.3 Caso Médio
Representa-se por Θ( ).

Este método é dos três, o mais difícil de determinar, pois necessita de análise estatística e como tal muitos testes. No entanto é muito usado, pois é também o que representa mais corretamente a complexidade do algoritmo.

2.1.4 Upper Bound
Seja dado um problema, por exemplo, multiplicação de duas matrizes quadradas de ordem n ($n \times n$). Conhecemos um algoritmo para resolver este problema (pelo método trivial) de complexidade O(n^3). Sabemos assim que a complexidade deste problema não deve superar O(n^3), uma vez que existe um algoritmo desta complexidade que o resolve. Um limite superior (*upper bound*) deste problema é O(n^3). O limite superior de um algoritmo pode mudar se alguém descobrir um algoritmo melhor. Isso de fato aconteceu com o algoritmo de *Strassen* que é $O(n \log 7)$. Assim o limite superior do problema de multiplicação de matrizes passou a ser $O(n \log 7)$. Outros pesquisadores melhoraram ainda este resultado. Atualmente o melhor resultado é o de *Coppersmith* e *Winograd* de $O(n^{2,376})$.

O limite superior de um algoritmo é parecido com o recorde mundial de uma modalidade de atletismo. Ela é estabelecida pelo melhor atleta (algoritmo) do momento. Assim como o recorde mundial o limite superior pode ser melhorado por um algoritmo (atleta) mais veloz.

2.1.5 Lower Bound
Às vezes é possível demonstrar que para um dado problema, qualquer que seja o algoritmo a ser usado, o problema requer pelo menos certo número de operações. Essa complexidade é chamada limite inferior (*lower bound*). Veja que o limite inferior depende do problema, mas não do algoritmo em particular. Usamos a letra Ω em lugar de O para denotar um limite inferior.

Para o problema de multiplicação de matrizes de ordem *n*, apenas para ler os elementos das duas matrizes de entrada leva O(n^2). Assim uma cota inferior trivial é Ω(n^2).

Na analogia anterior, um limite inferior de uma modalidade de atletismo não dependeria mais do atleta. Seria algum tempo mínimo que a modalidade exige, qualquer que seja o atleta. Um limite inferior trivial para os 100 metros seria o tempo que a velocidade da luz leva a percorrer 100 metros no vácuo.

Se um algoritmo tem uma complexidade que é igual ao limite inferior do problema, então o algoritmo é ótimo.

O algoritmo de *Coppersmith* e *Winograd* é de $O(n^{2,376})$, mas o limite inferior é de Ω(n^2). Portanto, não é ótimo. Pode ser que este limite superior possa ainda ser melhorado.

2.1.6 Complexidade de Tempo
Pode ser medida empiricamente, com funções para o cálculo do tempo.

Exemplo:
```
#include <time.h>
time_t t1, t2;

time(&t1);
/* Algoritmo entra aqui */
time(&t2);

double diferenca = difftime(t2, t1);
//diferença em segundos
```

2.1.6.1 ANÁLISE ASSINTÓTICA
Deve-se preocupar com a eficiência de algoritmos quando o tamanho de *n* for grande.

A eficiência assintótica de um algoritmo descreve a eficiência relativa dele quando *n* torna-se grande. Portanto, para comparar 2 algoritmos, determinam-se as taxas de crescimento de cada um. O algoritmo com menor taxa de crescimento rodará mais rápido quando o tamanho do problema for grande.

2.1.6.1.1 Calculando o tempo de execução

Supondo que as operações simples demoram uma unidade de tempo para executar, considere o código abaixo para calcular $soma = \sum_{i=1}^{n} i^3$.

```
int soma;

soma = 0;
for(i = 1; i <= n; i++)
    soma = soma + i * i * i;
printf("%d\n", soma);
```

Temos:

```
int soma;

soma = 0; // 1 unidade de tempo
for(i = 1; i <= n; i++) // 1 + n unidades
    soma = soma + i * i * i; // 1 + 1 + 2 unidades
printf("%d\n", soma); // 1 unidade
```

A linha **soma = 0;** custa 1 unidade de tempo. A atribuição **i = 1** no comando for custa 1 unidade de tempo. A linha **soma = soma + i * i * i;** custa 4 unidades de tempo e é executada n vezes pela linha **for(i = 1; i <= n; i++)**, portanto, tem custo de tempo $4n$. A impressão do resultado, realizada pela linha **printf("%d\n", soma);** custa 1 unidade de tempo. Somando tudo, temos: $1 + 1 + 4n + 1$, resultado num custo total para o código executar $4n + 3$, portanto, dizemos que o custo é $0(n)$. Isto é, no pior caso, o algoritmo tem custo <u>linear</u>.

Em geral, como se dá a resposta em termos do *big* O, costuma-se desconsiderar as constantes e elementos menores dos cálculos. Por exemplo, o que realmente deu a grandeza de tempo desejada no algoritmo anterior foi a repetição na linha **for(i = 1; i <= n; i++)**.

Em geral, não consideramos os termos de ordem inferior da complexidade de um algoritmo, apenas o termo predominante.

Por exemplo, um algoritmo tem complexidade $T(n) = 3n^2 + 100n$. Nesta função, o segundo termo tem um peso relativamente grande, mas a partir de $n_0 = 11$, é o termo n^2 que *dá o tom* do crescimento da função: uma parábola. A constante 3 também tem uma influência irrelevante sobre a taxa de crescimento da função após um certo tempo. Por isso dizemos que este algoritmo é da ordem de n^2 ou que tem complexidade $O(n^2)$.

2.1.6.1.2 Regras para o cálculo

Repetições

O tempo de execução de uma repetição é o tempo dos comandos dentro da repetição (incluindo testes) vezes o número de vezes que é executada.

Repetições aninhadas

A análise é feita de dentro para fora. O tempo total de comandos dentro de um grupo de repetições aninhadas é o tempo de execução dos comandos multiplicado pelo produto do tamanho de todas as repetições.

O exemplo abaixo é $O(n^2)$:

```
for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
        k = k + 1;
```

Comandos consecutivos

É a soma dos tempos de cada um bloco, o que pode significar o máximo entre eles.

O exemplo abaixo é $O(n^2)$, apesar da primeira repetição ser $O(n)$:

```
for(i = 0; i < n; i++)
    k = 0;

for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
        k = k + 1;
```

Se... então... senão

Para uma cláusula condicional, o tempo de execução nunca é maior do que o tempo do teste mais o tempo do maior entre os comandos relativos ao então e os comandos relativos ao senão.

O exemplo abaixo é $O(n)$:

```
if(i < j)
    i = i + 1;
else
    for(k = 1; k <= n; k++)
        i = i * k;
```

Exercício
Estime quantas unidades de tempo são necessárias para rodar o algoritmo abaixo.

```
int i, j;
int A[n];

i = 0;
while(i < n) {
   A[i] = 0;
   i = i + 1;
}
for(i = 0; i < n; i++)
   for(j = 0; j < n; j++)
      A[i] = A[i] + i + j;
```

Chamadas a sub-rotinas
Uma subrotina deve ser analisada primeiro e depois ter suas unidades de tempo incorporadas ao programa/subrotina que a chamou.

Sub-rotinas recursivas
Analise a recorrência (equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores).

Caso típico: algoritmos de dividir-e-conquistar, ou seja, algoritmos que desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas menores em tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução para o problema original.

Exemplo:
Calculo de *n*!

```
int fatorial (int n) {
   if(n == 1)
      return n;
   return fatorial(n - 1) * n;
}
```

Quando passamos um número n > 1 para a função fatorial, ela nos devolve $T(n) = T(n-1) \times n$. A chamada recursiva **fatorial(n – 1)**, chama novamente a função fatorial, mas desta vez n está uma unidade menor. O que acontece se n ainda é maior que 1? A função fatorial recebe o valor n - 1 como valor de n e, portanto, a equação de recorrência ficará $T(n-1) = T(n-1-1) \times (n-1) \Rightarrow T(n-1) = T(n-2) \times (n-1)$. Depois de algumas chamadas recursivas, n finalmente chegará a 1, assim teremos as seguintes relações de recorrência:

$$T(n) = T(n-1) \times n$$

$$T(n-1) = T(n-1-1) \times (n-1) \Rightarrow$$
$$T(n-1) = T(n-2) \times (n-1)$$

$$T(n-2) = T(n-3) \times (n-2)$$

$$\vdots$$

$$T(3) = T(3-1) \times 3 \Rightarrow T(2) \times 3$$

$$T(2) = T(2-1) \times 2 \Rightarrow T(1) \times 2$$

$$T(1) = 1$$

Sabendo que T(1) = 1, basta substituirmos na equação de cima para descobrirmos que $T(2) = 1 \times 2$, e portanto, tem custo de 1 unidade de tempo. Substituindo T(2) na equação de cima, temos $T(3) = 2 \times 3$, e também tem custo de 1 unidade de tempo. Continuando assim, descobriremos todos os custos de tempo até chegarmos a *n*. Ao somarmos todas as unidades de tempo consumidas, obtemos 1 + 1 + ... + 1, *n* vezes, de onde concluímos que a função tem custo O(*n*), ou seja, custo linear.

2.2 Melhorias e Implementações

2.2.1 Melhorias

O Quicksort opera escolhendo inicialmente um valor do *array* como elemento *pivot*, ou seja, um elemento que deveria, a priori, dividir o *array* em duas partes, na melhor das hipóteses, de tamanhos iguais.

Dependendo da ordem inicial dos elementos do *array* e do método de escolha do *pivot*, geralmente as partes à esquerda e à direita do *pivot* não serão iguais, mas desbalanceadas. Quando o desbalanceamento é pequeno, o algoritmo geralmente é o mais rápido entre todos os algoritmos de ordenação, mas quando ocorre um grande desbalanceamento, tais como uma quantidade muito pequena de elementos de um dos lados do *pivot* e uma quantidade muito grande do outro lado, ou um *array* vazio de um dos lados, o algoritmo degenera.

Você deve escolher com cuidado um método para definir o valor do comparando. O método é frequentemente determinado pelo tipo de dado que está sendo ordenado. Em listas postais muito grandes, onde a ordenação é frequentemente feita através do código CEP, a seleção é simples, porque os códigos são razoavelmente distribuídos

– e uma simples função algébrica pode determinar um comparando adequado. Porém, em certos bancos de dados, as chaves podem ser iguais ou muito próximas em valor e uma seleção randômica é frequentemente a melhor. Um método comum e satisfatório é tomar uma amostra com três elementos de uma partição e utilizar o valor médio.

Dicas:

- Escolher um *pivot* usando a mediana de três elementos, pois evita o pior caso.
- Depois da partição, trabalhar primeiro no subvetor de menor tamanho, pois diminui o crescimento da pilha.
- Utilizar um algoritmo simples (seleção, inserção) para partições de tamanho pequeno.
- Usar um Quicksort não recursivo, o que evita o custo de várias chamadas recursivas.

2.2.1.2 Um Quicksort Não Recursivo

```
void QuickSortNaoRec(Vetor A, Indice n) {
   TipoPilha pilha;
   TipoItem item;
   int esq, dir, i, j;

   // campos esq e dir
   FPVazia(&pilha);
   esq = 0;
   dir = n-1;
   item.dir = dir;
   item.esq = esq;
   Empilha(item, &pilha);
   do
      if (dir > esq) {
         Particao(A,esq,dir,&i, &j);
         if ((j-esq)>(dir-i)) {
            item.dir = j;
            item.esq = esq;
            Empilha(item, &pilha);
            esq = i;
         }
         else {
            item.esq = i;
            item.dir = dir;
            Empilha(item, &pilha);
            dir = j;
         }
      }
      else {
         Desempilha(&pilha,&item);
         dir = item.dir; esq = item.esq;
      }
   } while (!Vazia(pilha));
}
```

2.2.2 O Quicksort Nativo

Função pertencente à biblioteca **stdlib.h** que ordena um vetor usando o algoritmo Quicksort.

Exemplo:

```
qsort(tipo vetor, numElem, sizeof(tipo), compare);
```

como em

```
int values[6];
...
qsort(values, 6, sizeof(int), compare);
```

O último parâmetro passado para esta função é o nome da função que irá determinar o critério de comparação. Sendo assim podemos ordenar qualquer tipo de dados, até mesmo structs através do qsort, basta "ensinar" para ele como fazer isso.

Exemplo de compare para inteiros:

```
int compare (const void * a, const void * b){
   return ( *(int*)a - *(int*)b );
}
```

Exemplo de compare para um struct:

```
struct Times{
   char nome[21];
   int vitorias,sets,pontosmarcados;
};

int compare(const void *a, const void *b) {
   Times *x = (Times*) a;
   Times *y = (Times*) b;

   // criterios
   //1 - vitorias, maior primeiro
   if(x->vitorias!=y->vitorias)
      return y->vitorias-x->vitorias;

   //2 - sets, maior primeiro
   if(x->sets!=y->sets) return y->sets-x->sets;

   //3 - pontos marcados, maior primeiro
   if(x->pontosmarcados != y->pontosmarcados)
     return y->pontosmarcados – x - > pontosmarcados;

   //ordem alfabetica
   return (strcmp(x->nome,y->nome));
}
```

Exemplo simplificado de qsort:

```
#include <stdio.h>
#include <stdlib.h>

int values[] = {40, 10, 100, 90, 20, 25};

int compare(const void * a, const void * b) {
    return(*(int*)a - *(int*)b);
}

int main () {
    int n;
    qsort(values, 6, sizeof(int), compare);
    for(n=0; n<6; n++)
        printf ("%d ", values[n]);
    return 0;
}
```

Exemplo completo de qsort:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* função de comparação de inteiros para qsort */
int compara_inteiros(const void *a, const void *b) {
    const int *ia = (const int *)a; // casting de ponteiro
    const int *ib = (const int *)b;
    return *ia  - *ib;
    /* comparação de inteiros: retorna negativo se b > a
    e positivo se a > b */
}

/* função de impressão de vetor */
void imprime_vetor_inteiros(const int *array, size_t qtd) {
    size_t i;

    for(i=0; i<qtd; i++)
        printf("%3d | ", array[i]);

    putchar('\n');
}

/* exemplo de ordenação de inteiros usando qsort() */
void exemplo_ordena_inteiros() {
    int numeros[] = { 7, 3, 4, 1, -1, 23, 12, 43, 2, -4, 5 };
    size_t qtd_numeros = sizeof(numeros)/sizeof(int);


    puts("*** Ordenando Inteiros...");

    /* Imprime o vetor original */
    imprime_vetor_inteiros(numeros, qtd_numeros);

    /* Ordena o vetor usando qsort */
    qsort(numeros, qtd_numeros, sizeof(int), compara_inteiros);

    /* Imprime o vetor ordenado */
    imprime_vetor_inteiros(numeros, qtd_numeros);
}

/* função de comparação de C-string */
int compara_cstring(const void *a, const void *b) {
    const char **ia = (const char **)a;
    const char **ib = (const char **)b;
    return strcmp(*ia, *ib);
    /* strcmp trabalha exatamente como esperado de uma
    função de comparação */
}
```

```
/* função de impressão de vetor C-string */
void imprime_vetor_cstring(char **array, size_t qtd) {
    size_t i;

    for(i=0; i<qtd; i++)
        printf("%17s | ", array[i]);

    putchar('\n');
}

/* exemplo de ordenação de vetor C-strings usando qsort() */
void exemplo_ordena_cstrings() {
    char *strings[] = { "Vez", "Cesta", "Ruco", "Profetiza", "Asso", "Aco", "Assessorio", "Acessorio", "Ascender",
"Acender", "Assento", "Acento" };
    size_t qtd_strings = sizeof(strings) / sizeof(char *);

    /** STRING */
    puts("*** Ordenando String...");

    /* Imprime o vetor original */
    imprime_vetor_cstring(strings, qtd_strings);

    /* Ordena o vetor usando qsort */
    qsort(strings, qtd_strings, sizeof(char *), compara_cstring);

    /* Imprime o vetor ordenado */
    imprime_vetor_cstring(strings, qtd_strings);
}

/* um exemplo de struct */
struct st_ex {
    char produto[16];
    float preco;
};

/* função de comparação para struct (campo preco do tipo float) */
int compara_struct_por_preco(const void *a, const void *b) {
    struct st_ex *ia = (struct st_ex *)a;
    struct st_ex *ib = (struct st_ex *)b;
    return (int)(100.f*ia->preco - 100.f*ib->preco);
    /* comparação de float: retorna negativo se b > a
    e positivo se a > b. Nós multiplicamos o resultado por 100.0
    para preservar a frações decimais */

}

/* função de comparação para struct (campo produto do tipo C-string) */
int compara_struct_por_produto(const void *a, const void *b) {
    struct st_ex *ia = (struct st_ex *)a;
    struct st_ex *ib = (struct st_ex *)b;
    return strcmp(ia->produto, ib->produto);
    /* strcmp trabalha exatamente como esperado de uma
    função de comparação */
}
```

```
/* exemplo de função para impressão de vetor de struct */
void imprime_vetor_de_struct(struct st_ex *array, size_t qtd) {
    size_t i;

    for(i=0; i<qtd; i++)
        printf("[ produto: %-16s preco: $%8.2f ]\n", array[i].produto, array[i].preco);

    puts("--");
}

/* exemplo de ordenação de structs usando qsort() */
void exemplo_ordena_structs(void) {
    struct st_ex structs[] = {{"tocador de mp3", 299.0f}, {"tv lcd", 2200.0f},
                              {"notebook", 1300.0f}, {"smartphone", 499.99f},
                              {"dvd player", 150.0f}, {"iPad", 499.0f }};

    size_t structs_qtd = sizeof(structs) / sizeof(struct st_ex);

    puts("*** Ordenando Struct (preco)...");

    /* Imprime vetor de struct original */
    imprime_vetor_de_struct(structs, structs_qtd);

    /* ordena vetor de struct usando qsort */
    qsort(structs, structs_qtd, sizeof(struct st_ex), compara_struct_por_preco);

    /* Imprime vetor de struct ordenado */
    imprime_vetor_de_struct(structs, structs_qtd);

    puts("*** Ordenando Struct (produto)...");

    /* reordena vetor de struct usando outra função de comparação */
    qsort(structs, structs_qtd, sizeof(struct st_ex), compara_struct_por_produto);

    /* Imprime vetor de struct ordenado */
    imprime_vetor_de_struct(structs, structs_qtd);
}

int main() {
    exemplo_ordena_inteiros();
    exemplo_ordena_cstrings();
    exemplo_ordena_structs();
    return 0;
}
```

3 Ordenando Outras Estruturas de Dados

Até agora, apenas arrays de inteiros ou strings foram ordenados. Entretanto, geralmente são os tipos complexos de dados que precisam ser ordenados.

3.1 Ordenação de Strings

A maneira mais fácil de ordenar strings é criar um array de ponteiros e caracteres para essas strings. Isso permite manter uma indexação fácil e conservar a base do algoritmo do Quicksort inalterada. A versão para string do Quicksort a seguir aceita um array de strings, cada uma de até 10 caracteres de comprimento.

Listagem 1:

```
// Um quicksort para strings
void quick_string(char item[][10], int count) {
  qs_string(item, 0, count – 1);
}

void qs_string(char item[][10], int left, int right) {
    register int i, j;
    char *x;
    char temp[10];

    i = left; j = right;
    x = item[(left + right) / 2];
    do {
       while((strcmp(item[i], x)) < 0 && (i < right)) i++;
       while((strcmp(item[j], x)) > 0 && (j > left)) j--;
        if(i <= j) {
            strcpy(temp, item[i]);
            strcpy(item[i], item[j]);
            strcpy(item[j], temp);
            i++; j--;
        }
    } while(i <= j);
    if(left < j) qs_string(item, left, j);
    if(I < right) qs_string(item, i, right);
}
```

Observe que o passo da comparação foi alterado para usar a função **strcmp()**. A função devolve um número negativo se a primeira string é lexicograficamente menor que a segunda, zero se as strings são iguais e um número positivo se a primeira string é lexicograficamente maior que a segunda.

O uso da função **strcmp()** diminui a velocidade da ordenação por duas razões. Primeiro ela envolve uma chamada a uma função, que sempre toma tempo. Segundo, **strcmp()** realiza diversas comparações para determinar a relação entre as duas strings. No primeiro caso, se a velocidade de execução for realmente crítica, pode-se colocar o código para **strcmp()** *inline*, dentro da rotina, duplicando seu código. No segundo caso, não há maneira de evitar a comparação entre as strings, visto que, por definição, essa é a tarefa que tem de ser executada.

3.2 Ordenação de Estruturas

Muitos dos programas aplicativos que requerem uma ordenação precisam ter um agrupamento de dados ordenados. Uma lista postal é um excelente exemplo, porque o nome, a rua, a cidade, o estado e o CEP estão todos relacionados. Quando essa unidade conglomerada de dados é ordenada, uma chave de ordenação é usada, mas toda a estrutura é trocada. Para entender como isso é feito, primeiro criemos uma estrutura. Uma estrutura conveniente é

Listagem 2:

```
struct address {
    char name[40];
    char street[40];
    char city[20];
    char state[3];
    char zip[10];
}
```

state tem extensão de três caracteres e **zip** de 10 caracteres, porque um *array* sempre precisa de um caractere a mais do que o comprimento máximo da string para armazenar o terminador nulo.

Como é razoável que uma lista postal possa ser arranjada como um *array* de estruturas, assuma, para esse exemplo, que a rotina ordenará um *array* de estruturas do tipo **address**. A rotina é mostrada aqui:

Listagem 3:

```
// Um quicksort para structs
void quick_struct(struct address item[], int count)
{
  qs_struct(item, 0, count – 1);
}

void qs_struct(struct address item[], int left, int
right) {
    register int i, j;
    char *x;
    struct address temp;

    i = left; j = right;
    x = item[(left + right) / 2].zip;
    do {
       while((strcmp(item[i].zip, x)) < 0 && (i < right)) i++;
       while((strcmp(item[j].zip, x)) > 0 && (j > left)) j--;
        if(i <= j) {
            strcpy(temp, item[i]);
            strcpy(item[i], item[j]);
            strcpy(item[j], temp);
            i++; j--;
        }
    } while(i <= j);
    if(left < j) qs_struct(item, left, j);
    if(i < right) qs_struct(item, i, right);
}
```

4 Exercícios Propostos

1. Crie um programa que faça uma comparação entre todos os métodos de ordenação estudados em aula com relação a estabilidade (preservar ordem lexicográfica), ordem de complexidade levando em consideração comparações e movimentações para um vetor de 100 inteiros contendo os 100 primeiros números naturais em ordem decrescente.
2. Escreva um programa para ordenar os 200 primeiros números da lista de 100000 números inteiros fornecida no blog, primeiro usando o *pivot* como o primeiro elemento, depois usando o *pivot* como o elemento central, depois usando um *pivot* escolhido aleatoriamente e por fim, um *pivot* usando a mediana entre três o primeiro, o último e elemento central dos valores.
3. Escreva um programa que leia compromissos para uma agenda, anotando o assunto, a data do compromisso (no formato dd/mm/aaaa) e a hora do compromisso (no formato hh:mm) e ordene os compromissos em ordem crescente de data, hora e assunto.

5 Terminamos

Terminamos por aqui.

Corra para o próximo tutorial.