Universidade Federal de São João del - Rei

Ciência da Computação – 2º período

Algoritmos e Estruturas de Dados II

ANÁLISE EXPERIMENTAL DO ALGORITMO DE ORDENAÇÃO QUICKSORT

Gabriel Spada Ramos

Isabella Vieira Ferreira

Mônica Neli de Resende

Novembro / 2011

Sumário

Lista de figuras

Lista de tabelas

1. Introdução

QuickSort é o algoritmo de ordenação interna mais rápido para uma variedade de situações. O algoritmo foi inventado por C. A. R. Hoare em 1960 e publicado em 1962 após uma série de refinamentos.

O seu funcionamento utiliza o princípio de dividir recursivamente problema principal (de ordenação) em problemas menores a partir de um pivô, ou seja, trocando os elementos de posição de modo que a parte esquerda contenha valores menores que o pivô e a parte direita valores maiores que o pivô.

QuickSort é extremamente eficiente na ordenação de dados. O método necessita apenas de uma pilha como memória auxiliar e requer em média $n \log n$ operações para ordenar n itens. Vale ressaltar que para arquivos já ordenados a escolha do pivô é muito importante para evitar o pior caso. Escolher os extremos de um arquivo, por exemplo, torna as partições muito desiguais eliminando apenas um item a cada chamada recursiva aumentando o número de comparações e o tamanho da pilha.

Ele é o mais usado para a maioria das aplicações, porém é necessário que se consiga uma implementação consistente. Neste trabalho apresentaremos discussões feitas sobre a análise experimental de quatro diferentes formas de melhorar o seu desempenho.

2. Proposta de trabalho

Conforme nos foi proposto temos como objetivo realizar alterações na forma tradicional do QuickSort e verificar se o comportamento previsto na análise teórica corresponde à realidade de um processo sendo executado sobre um sistema, definindo uma base de testes (comparabilidade) e realizando-os para cada caso (confiabilidade). Dessa forma, para que este se comporte de maneira melhor, ou seja, com menor probabilidade de ocorrer o pior caso que é $O(n^2)$, analisamos as seguintes possibilidades:

- Escolher aleatoriamente o pivô;
- Verificar se o trecho atual já está ordenado, interrompendo a recursão caso estiver;
- Escolher como pivô a mediana de três elementos aleatórios;
- Ordenar diretamente o conjunto de dados para instâncias pequenas utilizando outro algoritmo (InsertionSort);

3. Descrição da solução

3.1 Associação dos nomes dos algoritmos às modificações

- *Quicksort tradicional:* quicksort.c
- *Quicksort com a escolha aleatória do pivô:* pivo_aleatorio.c
- *Quicksort com interrupção da recursão para "trechos" já ordenados:* para_recursao.c
- *Quicksort com o pivô sendo a mediana de três elementos aleatórios:* medianade3.c
- *Quicksort com ordenação direta (parando a recursão) para instâncias "pequenas" utilizando o InsertionSort:* uso_insertionsort.c

3.2 Implementação

Através de um *menu* os algoritmos oferecem a opção de gerar os valores da entrada em ordem crescente, decrescente e pseudo-aleatória, armazenando-os em um arquivo denominado *Entrada.txt*. Os dados são ordenados de forma crescente e a saída estará em outro arquivo cujo nome é *Saida.txt*. Os arquivos serão criados a cada nova execução do algoritmo e salvos na mesma pasta que o programa.

Foi utilizado uma implementação com alocação estática de memória, onde o usuário insere o número de valores que deseja ordenar. Os elementos do vetor são gerados aleatoriamente da seguinte forma:

```
srand(12);
     for (i = 1; i <= max; i++){
     //Gerando números pseudo-aleatorios e guardando
no vetor.
          vetor[i].chave = rand() % max;
     }
```

Onde:

- s*rand (12)* é o gerador de semente que definimos como sendo 12, gerando assim a mesma sequência de números;
- *max* é o tamanho máximo do vetor informado pelo usuário;
- *rand () é* a função que gera os números a partir da semente.

Estrutura de dados utilizada:

```
typedef struct item {
     int chave;
} Item;
```

3.3 A função “particionar” e a escolha do pivô

O principal foco das otimizações é a escolha do pivô na função “particionar”, que quanto mais otimizada for, mais rápida será a ordenação. A escolha do pivô determina o quão balanceado será o particionamento. Um particionamento é otimamente balanceado quando o elemento escolhido como pivô é o elemento central da sequência ordenada, ou seja, a mediana de todos os elementos. E é pessimamente balanceado quando o elemento escolhido como pivô é o maior ou o menor elemento da sequência. O melhor caso do Quicksort ocorre quando todos os particionamentos em todos os níveis de recursão forem ótimos, ou seja, quando toda vez que a função “particionar” for

chamada, ela divida a sequência exatamente ao meio (no caso do QuickSort tradicional). Isso pode ser visto como uma árvore binária onde a profundidade de todas as folhas é a mesma.

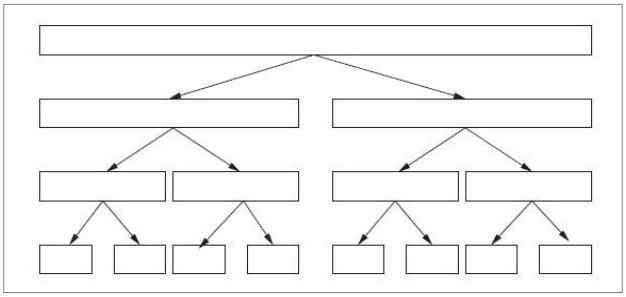

Figura 1: Árvore binária representando o melhor caso do QuickSort

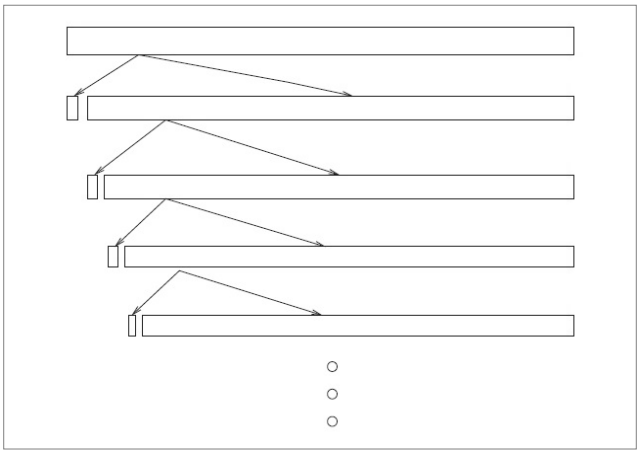

Figura 2: Árvore binária representando o pior caso do QuickSort

As otimizações na função “particionar” ocasionam melhorias relevantes no tempo médio de resposta e não alteram a escala de complexidade do algoritmo.

3.4 Descrição de rotinas

Os algoritmos estão modularizados da seguinte forma:

- **void particionar (int esq, int dir, int *i, int *j, Item a [max+1])**

Contém a escolha do pivô (que será alvo das modificações) e o particionamento do vetor. O método consiste em percorrer a parte esquerda até encontrar algum elemento maior que o pivô, em seguida percorre a direita procurando por algum elemento menor que o pivô, trocando-os de posição caso sejam encontrados. Repete-se o procedimento até que os apontadores se cruzem.

- **void ordenar (int esq, int dir, Item a [max+1])**

Responsável por chamar recursivamente a função “particionar” até que cada parte se encontre ordenada.

- **void quicksort (Item a[max+1], int *n)**

Responsável por chamar a função “ordenar”.

3.5 Ordem de execução

O algoritmo consiste em:

- Ler o número de valores a serem ordenados (informado pelo usuário);
- Preencher o vetor com números aleatórios, em ordem crescente ou decrescente;
- Armazenar os dados gerados em um arquivo;

- Ordenar a sequência a partir da chamada à função “QuickSort” (a função “QuickSort” chama a função “ordenar” que, por sua vez, chama recursivamente a função “particionar” até que os dados se encontrem todos ordenados).
- Armazenar os dados ordenados em um arquivo;

3.6 Descrições das otimizações

3.6.1 QuickSort tradicional

O pivô é obtido a partir da mediana dos ponteiros esquerda e direita (da partição).

Trecho do código:

```
// obtem o pivô x. O pivô é o elemento do meio.
     x = a[(*i + *j) / 2];
```

3.6.2 Escolha aleatória do pivô

Foi utilizado a função *srand (time (NULL))*, de modo que a cada chamada da função “particionar” gerasse um pivô aleatório entre os valores contidos naquela partição.

Trecho do código onde foi feita a modificação:

```
//Escolhendo aleatoriamente o pivô
     srand(time(NULL));
     h = rand() % (*j-*i) + *i;
     x = a[h];
```

3.6.3 Escolha da mediana de três elementos aleatórios

É alocado um vetor auxiliar de três posições e preenchidos com números pseudo-aleatórios gerados entre os valores naquela partição. Em seguida ordena-os e atribui a mediana ao pivô.

Foi utilizada a seguinte estrutura de dados:

```
typedef struct auxilia{

     int chave;

     int valor;

} Auxilia;
```

Trecho do código onde foi feita a modificação:

```
//Escolhendo tres elementos aleatorios e guardando-os no
vetor.
     for(l=0;l<3;l++){
           aux = rand()%(dir - esq) + esq;
           z[l].valor = a[aux].chave;
           z[l].chave = rand()%(dir - esq) + esq;
     }
     //Ordenando os tres elementos que foram gerados.
     for(l=0;l<3;l++){
          for(m=1;m<3;m++){
               if (z[l].chave > z[m].chave){
                    n = z[l];
                    z[l]=z[m];
                    z[m]=n;
               }
          }
     }
     //Pivô.
          x = a[z[1].chave];
```

3.6.4 Ordenação direta de dados para instâncias pequenas utilizando outro algoritmo

O pivô foi escolhido fazendo a mediana dos índices onde estão os valores da esquerda e da direita daquela partição. O algoritmo segue com as chamadas recursivas até termos um número de valores maiores que 20. Para instâncias menores ou iguais a 20 chama-se o algoritmo de ordenação interna InsertionSort, que consiste em verificar qual posição é a adequada para o elemento, comparando-o com os anteriores e inserindo-o na posição certa utilizando uma sentinela (posição auxiliar no vetor) .

Trecho do código onde foi feita a modificação:

```
// Algoritmo de ordenacao: InsertionSort
void insercao(Item a[max+1],int *k ,int *n) {
     int i, j,aux;
     Item x;

     aux = a[*k-1].chave;

     for (i = *k+1; i <= *n; i++) {
          x = a[i];
          j = i - 1;
          a[*k-1] = x;

          while (x.chave < a[j].chave) {
               a[j+1] = a[j];
               j--;
          }
          a[j+1] = x;
     }
     a[*k-1].chave = aux;
}

//Na função ordenar

// Se o numero de elementos da particao for menor ou igual
a 20, utiliza o algoritmo de insercao.

if ((dir - esq) <= 20){

     insercao(a,&esq,&dir);

}
```

3.6.5 Verifica se o trecho atual já está ordenado, interrompendo a recursão caso estiver

Para instâncias menores que 100 elementos, o algoritmo verifica se aquela partição já está ordenada, caso estiver a recursão é interrompida.

Trecho do código onde foi feita a modificação:

```
void pararecursao(int *verifica,int esq,int dir,Item
a[max+1]){
     int i;

     for(i=esq;i<=dir-1;i++){
          if (a[i+1].chave < a[i].chave){
               *verifica = 1;
          }
     }
          }



//Na função ordenar

if ((dir - esq) < 100){
          verifica = 0;
          pararecursao (&verifica,esq,dir,a);
          if (verifica==0) return;
               }
```

4. Ilustração do funcionamento do algoritmo (QuickSort Tradicional)

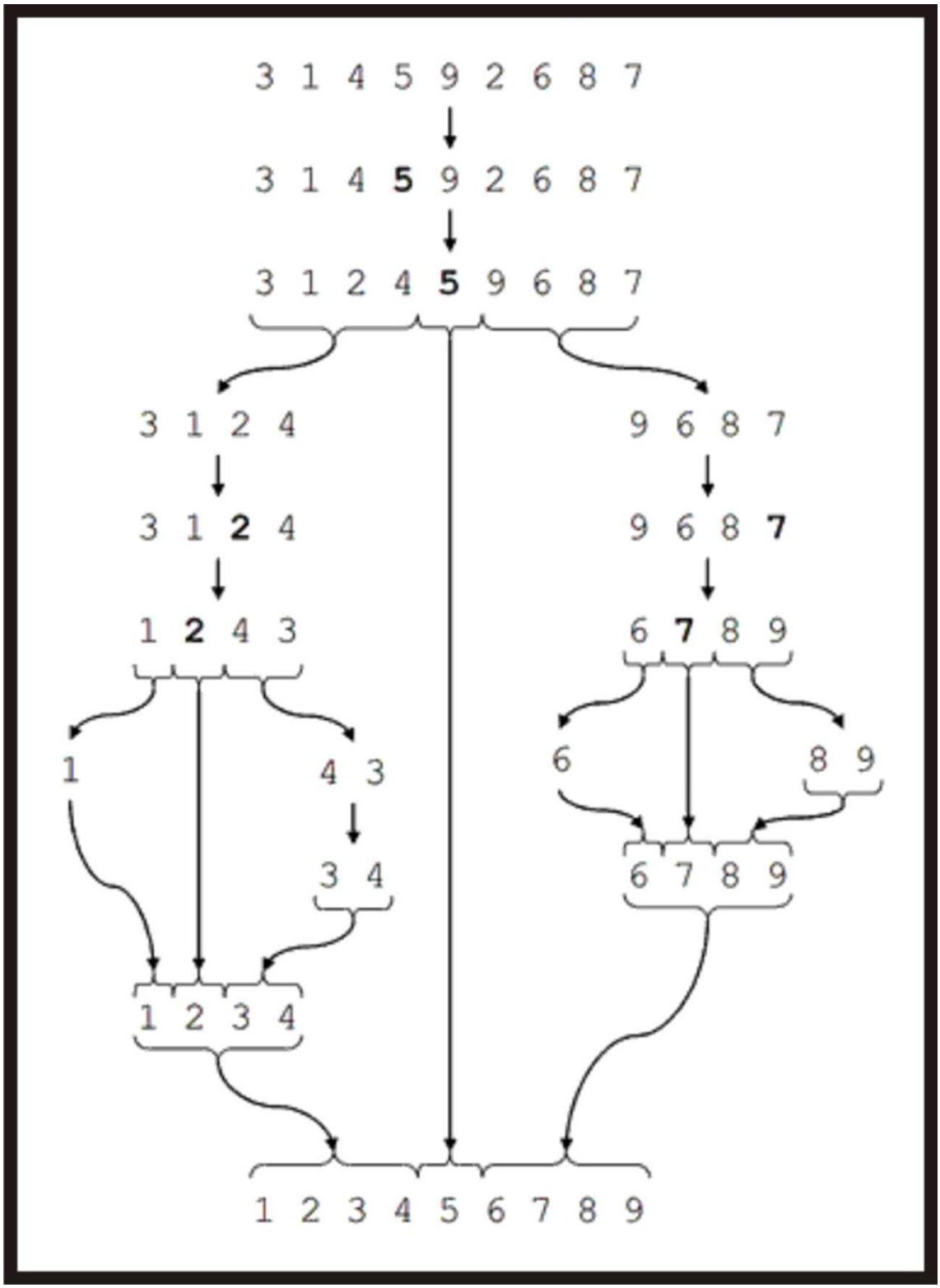

Figura 3 – Ilustração do funcionamento do QuickSort

5. Análises e discussões

Foram feitos 30 testes para cada tipo de entrada (crescente, decrescente e pseudo-aleatória) analisando entradas com 10.000, 100.000 e 2.000.000 elementos. Os tempos foram calculados a partir do comando *time ./prog* ao executar o programa. A tabela a seguir apresenta a média dos tempos (em segundos) de cada modificação:

Crescente			
Algoritmos	**Elementos**		
	10.000	**100.000**	**2.000.000**
Tradicional	**0,00290**	**0,03820**	0,76120
Pivô Aleatório	0,02940	0,27540	5,49580
Mediana de 3	0,00700	0,06240	1,09760
Pára recursão	**0,00290**	0,04400	**0,69720**
InsertionSort	0,00770	0,04880	0,71920

Tabela 1: Média dos tempos com entrada em ordem crescente

Decrescente			
	Elementos		
Algoritmos	**10.000**	**100.000**	**2.000.000**
Tradicional	0,00410	**0,04090**	0,77580
Pivô Aleatório	0,03220	0,29440	5,49800
Mediana de 3	0,06660	0,06090	1,09570
Pára recursão	**0,00380**	0,04120	**0,66960**
InsertionSort	0,00800	0,04520	0,72000

Tabela 2: Média dos tempos com entrada em ordem decrescente

Pseudo-Aleatória			
	Elementos		
Algoritmos	10.000	100.000	2.000.000
Tradicional	0,00780	0,06220	1,25660
Pivô Aleatório	0,04640	0,42410	8,45940
Mediana de 3	0,01050	0,08350	1,60690
Pára recursão	**0,00680**	0,07010	1,28160
InsertionSort	0,01250	**0,06120**	**1,13000**

Tabela 3: Média dos tempos com entrada em ordem pseudo-aleatória

Os valores de tempo marcados em negrito são os menores valores observados para aquele tipo de entrada e quantidade de elementos pré-determinada. Podemos perceber que as modificações “Pára recursão” e “InsertionSort” foram as que demonstraram melhores resultados, e em alguns casos o QuickSort na forma tradicional prevaleceu.

As outras modificações, como “Mediana de 3” e “Pivô Aleatório” , tem como objetivo evitar o pior caso do QuickSort, tendo assim valores maiores de tempo para os testes, mas com a vantagem de evitar casos onde o tempo seria bem pior.

6. Conclusão

Foram apresentados neste trabalho diferentes formas nas quais a idéia inicial do QuickSort pode ser aperfeiçoada.Verificamos quão importante é a escolha do pivô no desempenho do algoritmo, tal que quanto mais central for, melhor será, tornando as partições bem divididas e evitando o pior caso do algoritmo.

Percebemos que as otimizações utilizando o algoritmo InsertionSort para instâncias pequenas e verificando se o trecho atual já está ordenado interrompendo a recursão apresentaram uma melhora significativa no tempo de execução, pois o QuickSort tradicional não é muito eficiente para seqüências de tamanho pequeno (em relação ao tempo de resposta) e com relação a interrupção das chamadas recursivas há uma economia de memória auxiliar.

As modificações de escolher como pivô a mediana de três elementos aleatórios e escolher aleatoriamente o pivô não apresentaram uma melhora no tempo de execução, mas evitam o pior caso.

7. Bibliografia

ZIVIANI, N. . Projeto de Algoritmos com Implementações em Pascal e C. 2a.. ed. São Paulo: Thompson Learning, 2004. v. 1. 572 p.

PRADO, J. A. S. Análise experimental do *Quicksort* probabilístico com gerador de números pseudo-aleatórios penta-independente; 2005; Curitiba.