Aluno: Hugo Martins de Lima
Professora: Josiane Rodrigues
Matéria: Projetos e Análise de Algoritmos.
Data: 26/09/2016

Introdução

A análise de algorítmo estuda os algorítmos desenvolvidos para resolver os mais diversos tipos de problemas que existem hoje em dia na área da computação. Análise de algoritmos responde perguntas do tipo: Este algoritmo resolve o meu problema? Quanto tempo o algoritmo consome para processar uma 'entrada' de tamanho *n*?.

Experimentos

Neste trabalho realizaremos alguns experimentos de ordenação de vetores. Abordaremos 2 algorítmos o HeapSort e o QuickSort. Os experimentos foram feitos no computador com as seguintes configurações: Intel(R) Core(TM) i3-2370M CPU @ 2.40GHz, Sistema Operacional Linux distribuição Fedora24, 8 Gb de memória ram. No momento dos experimentos o computador estava a disposição total para rodar os experimentos.

1. HEAPSORT

Tem um desempenho em tempo de execução muito bom em conjuntos ordenados aleatoriamente, tem um uso de memória bem comportado e o seu desempenho em pior cenário é praticamente igual ao desempenho em cenário médio. Alguns algoritmos de ordenação rápidos têm desempenhos extremamente ruins no pior cenário, quer em tempo de execução, quer no uso da memória.

O heapsort utiliza uma estrutura de dados chamada heap, para ordenar os elementos à medida que os insere na estrutura. Assim, ao final das inserções, os elementos podem ser sucessivamente removidos da raiz da heap, na ordem desejada, lembrando sempre de manter a propriedade de max-heap.

A heap pode ser representada como uma árvore (uma árvore binária com propriedades especiais) ou como um vetor. Para uma ordenação decrescente, deve ser construída uma heap mínima (o menor elemento fica na raiz). Para uma ordenação crescente, deve ser construído uma heap máxima (o maior elemento fica na raiz).

- **Propriedades do heap:**

Pai(i) = i/2

f_Esq(i) = 2i

f_Dir(i) = 2i+1

A[Pai(i)] = A[i]

- **Análise do código**

```
void  HEAPFY(int *A, int i, int tam_heap) {
  int l, r, m;
  l = LEFT(i);
  r = RIGHT(i);
  m = i;
  if((l <= tam_heap) && (A[l] > A[i])) {
            m = l;
  }
  if((r <= tam_heap) && (A[r] > A[m])) {
            m = r;
  }
  if(m != i) {
```

```
            TROCAR(i, m, A);
            HEAPFY(A, m, tam_heap);
  }
}
```

No HEAPFY(Essa função é responsável por deixar o vetor com uma estrutura heap, obedecendo as propriedades citadas acima) calculamos o pior caso pelo tamanho da árvore, que é calculado por h = |logn| logo o custo do heapfy é T(n) = O(logn)

```
void BUILD_HEAP(int *A) {
  for(int i = (SIZEOFVETOR/2); i >= 1; i--) { ----------------------------------n/2 vezes
              HEAPFY(A, i, SIZEOFVETOR);---------------------------------------O(log n)
  }-------------------------------------------------------------------------------T(n) = O(nlog n)
}

void HEAPSORT(int *A) {
  int TAM_HEAP = SIZEOFVETOR;
  BUILD_HEAP(A);---------------------------------------O(nlogn)
 for(int i = SIZEOFVETOR; i >= 2; i--) {---------------n -1 vezes
    TROCAR(1, i, A);-------------------------------------O(1)
    TAM_HEAP = (TAM_HEAP – 1);---------------------O(1)
    HEAPFY(A, 1, TAM_HEAP);------------------------O(logn)
  }
}
```

T(n) = O(nlogn)

- **vejamos alguns experimentos realizados:**

HEAPSORT(Aleatório)				
TEMPO 1 (MINUTOS)	**TEMPO 2 (MINUTOS)**	**TEMPO 3 (MINUTOS)**	**TAMANHO**	**TEMPO(MINUTOS)**
0,1166666667	0,1166666667	0,1166666667	10.000.000	0,1166666667
1,3	1,3	1,3	100.000.000	1,3
14,2	14,4333333333	14,2166666667	1.000.000.000	14,2833333333

HEAPSORT(Ordenado crescente)				
TEMPO 1 (MINUTOS)	**TEMPO 2 (MINUTOS)**	**TEMPO 3 (MINUTOS)**	**TAMANHO**	**TEMPO(MINUTOS)**
0,1166666667	0,1166666667	0,1	10.000.000	0,1111111111
1,25	1,2333333333	1,2333333333	100.000.000	1,2388888889
14,2833333333	14,3	14,2666666667	1.000.000.000	14,2833333333

HEAPSORT(90% dos elementos repetidos)				
TEMPO 1 (MINUTOS)	**TEMPO 2 (MINUTOS)**	**TEMPO 3 (MINUTOS)**	**TAMANHO**	**TEMPO(MINUTOS)**
0,0333333333	0,0333333333	0,0333333333	10.000.000	0,0333333333
1,3	1,2833333333	1,2833333333	100.000.000	1,2888888889
14,4	14,35	14,3666666667	1.000.000.000	14,3722222222

Com isso, concluo que o Heapsort trabalha no lugar ou seja “in place” e o tempo de execução em pior cenário para ordenar *n* elementos é de O(*n*log*n*). Para valores de *n*, razoavelmente grandes, o termo log*n* é quase constante, de modo que o tempo de ordenação é quase linear com o número de itens a ordenar.

2. QUICKSORT

A ideia do quicksort é **dividir, conquistar, combinar.**

Dividir o problema de ordenar um conjunto com n itens em dois problemas menores e esses problemas menores são divididos em outros problemas menores até restar apenas 1 elemento no vetor que pela definição um vetor com apenas um elemento já está ordenado. No final os vetores são combinadas para produzir a solução final.

- **Análise de algoritmo**

```
int PARTITION(int *A, int p, int r) {
  int x, i;
  x = A[r];-----------------------------------------------THETA(1)
  i = p-1;------------------------------------------------THETA(1)
  for(int j = p; j <= (r-1); j++) {----------------------(N-1 VEZES)
    if(A[j] <= x ) {---------------------------------------THETA(1)
      i = i+1;---------------------------------------------THETA(1)
      TROCAR(i, j, A);  ------------------------------ THETA(1)
    }
  }
  TROCAR((i+1), r, A); -----------------------------THETA(1)
  return i+1;
}
```

T(n) = THETA(n)

```
void QUICKSORT(int *A, int p, int r) {
  int q;
  if(p < r) {
    q = PARTITION(A, p, r);
    QUICKSORT(A, p, q-1);
    QUICKSORT(A, q+1, r);
  }
}
```

- **Pior caso:**

Vetor está ordenado, onde os subvetores estão mal balanceados

$T(n) = THETA(n^2)$

- **Melhor caso**

Os subvetores estão mais balanceados possíveis

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

- **Caso médio**

Os subvetores estão bem balanceado em alguns casos e em outros casos estão mal balanceados

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

- **vejamos alguns experimentos realizados:**

QUICKSORT(Aleatório)				
TEMPO 1 (MINUTOS)	**TEMPO 2 (MINUTOS)**	**TEMPO 3 (MINUTOS)**	**TAMANHO**	**TEMPO(MINUTOS)**
0,05	0,0666666667	0,0666666667	10.000.000	0,0611111111
0,6666666667	0,6666666667	0,7	100.000.000	0,6777777778
7,75	7,65	7,6666666667	1.000.000.000	7,6888888889

QUICKSORT(Ordenado crescente)				
TEMPO 1 (MINUTOS)	**TEMPO 2 (MINUTOS)**	**TEMPO 3 (MINUTOS)**	**TAMANHO**	**TEMPO(MINUTOS)**
0,9	0,8833333333	0,8833333333	100.000	0,8888888889
1,9833333333	1,9833333333	1,9833333333	150.000	1,9833333333
2,2833333333	2,3166666667	2,2833333333	160.000	2,2944444444

QUICKSORT(90% dos elementos repetidos)				
TEMPO 1 (MINUTOS)	**TEMPO 2 (MINUTOS)**	**TEMPO 3 (MINUTOS)**	**TAMANHO**	**TEMPO(MINUTOS)**
0,7333333333	0,7166666667	0,7166666667	100.000	0,7222222222
1,6166666667	1,6166666667	1,6333333333	150.000	1,6222222222
1,85	1,8666666667	1,8666666667	160.000	1,8611111111

Com isso, concluo que o QuickSort é uma técnica que funciona bem para números distintos e aleatórios, já com vetores ordenados ou com muitos números repetidos não funciona muito bem e em alguns casos, cai no pior caso. Ou seja, uma função quadrática. Nessa experiência não foi possível avaliar o QuickSort com vetor ordenado ou com 90% dos elementos repetidos com n > 160.00.

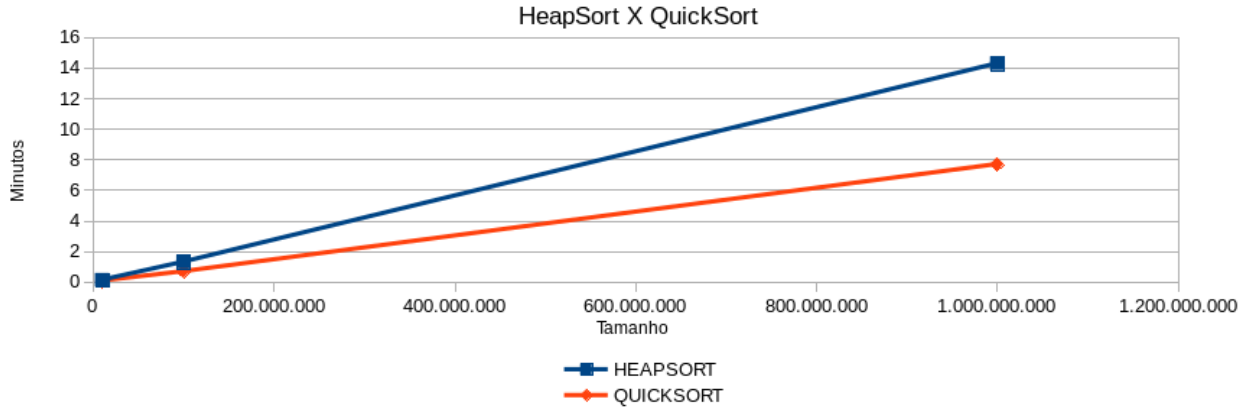
Conclusão

Concluimos que o QuickSort se sai um pouco melhor que o HeapSort no caso médio. Porém nos demais experimentos o HeapSort se sai muito melhor em relação ao QuickSort como mostram as tabelas no final de cada experimento. Nessa conclusão separamos duas tabelas(1.0 , 2.0), na 1.0 os valores são aleatórios de tamanho 10.000.000, 100.000.000, 1.000.000.000 .

1.0

Tempos em minutos

ALEATÓRIO			
	10.000.000	**100.000.000**	**1.000.000.000**
HEPSORT	0,1166666667	1,3	14,2833333333
QUICKSORT	0,0611111111	0,7	7,6888888889



2.0

tempos em minutos

90% VALORES REPETIDOS			
	10.000.000	**100.000.000**	**1.000.000.000**
HEPSORT	0,0333333333	1,2888888889	14,3722222222
QUICKSORT	Não consegui experimentar	Não consegui experimentar	Não consegui experimentar