Universidade Federal de Pelotas
Bacharelado em Ciência da Computação
Estrutura de Dados II – 2008/02
Prof. Ricardo Araújo
Trabalho 1

Bruno Martins Rodrigues – 06101898

Especificação

Desenvolver os algoritmos de ordenação (Bubblesort, Insertsort, Heapsort, Shellsort e Quicksort) e fazer testes, aumentando gradativamente o número de elementos, com o objetivo de traçar gráficos e fazer a análise de eficiência dos mesmos.

Desenvolvimento

Durante o trabalho, cada algoritmo rodou testes para listas ordenadas em ordem crescente, inversa e números aleatórios. Para cada vetor, os valores variavam entre 0 e o número de elementos do vetor, valor este que aumentava em 5 mil a cada teste, até o limite de 100mil elementos. A princípio esse valor era muito maior, porém, em alguns algoritmos o tempo de cada execução pode demorar 30min para cada passo. Algoritmos mais velozes, como o Quicksort não tem esse problema, porém, o tempo para testes foi curto, não podendo fazer testes com um maior número de elementos.

Cada valor foi testado 20 vezes, a fim de se obter o tempo de teste mais preciso, pois em alguns casos há uma grande variação entre o menor e o maior tempo na execução do algoritmo em um mesmo vetor. Isso se dá por diversos fatores, tais como: uso do processador por outra aplicação, posição dos elementos no vetor e diversos outros fatores. Também foi salvo o melhor e o pior tempo de cada passo.

Configuração do Computador:

Processador: AMD Turion(tm) 64 X2 Mobile Technology TL-50 – 1.6Ghz
Memória: 1.7 Gb de RAM
Sistema Operacional: GNU/Linux Ubuntu 8.04

Bubblesort

O bubblesort, ou, ordenação por método bolha é um dos algoritmos mais simples e menos eficientes de ordenação. Ele simplesmente varre cada posição do vetor, trocando os elementos de posição caso ele seja menor que o anterior. O melhor caso, é um vetor já ordenado, e o seu pior caso, é um vetor ordenado inversamente.

Algoritmo

```
void bubblesort (int *v, int size)
{
	int i, j;

	for (i = size - 1; i > 0; i--)
		for (j = 0; j < i; j++)
			if (v[j] > v[j+1])
			{
				int temp;

				temp = v[j];
				v[j] = v[j+1];
				v[j+1] = temp;
			}
}
```

Insertsort

Ordenação por inserção, pode ser comparado como a maneira que as pessoas ordenam cartas de baralho, pois ele age do início (esquerda) para o fim, colocando os menores elementos no início do vetor. O melhor caso é o vetor ordenado, e o pior, ordenado inversamente.

Algoritmo

```
void insertsort(int *v, int size)
{
	int i, j, item;

	for (i = 1; i < size; ++i)
	{
		item = v[i];
		for (j = i - 1; j >= 0 && v[j] > item; --j)
			v[j + 1] = v[j];
		v[j + 1] = item;
	}
}
```

Heapsort

O algoritmo cria uma estrutura de dados chamada heap, que ordena os elementos a medida que os elementos são adicionados na estrutura.

Algoritmo

```
void heapsort(int *v, int size)
{
	int i, pai, filho, t;

	i = size / 2;

	while (1)
	{
		if (i > 0)
		{
			i--;
			t = v[i];
		}

		else
		{
			size--;
			if (size == 0) return;
			t = v[size];
			v[n] = v[0];
		}

		pai = i;
		filho = i * 2 + 1;

		while (filho < size)
		{
			if ((filho + 1 < size) && (v[filho + 1] > v[filho]))
				filho++;

			if (v[filho] > t)
			{
				v[pai] = v[filho];
				pai = filho;
				filho = pai * 2 + 1;
			}
			else break;
		}
		v[pai] = t;
	}
}
```

Shellsort

Este algoritmo divide o vetor sucessivamente em partes menores para depois utilizar o método Insertsort sobre ele e finalmente unir as partes. Costuma ser bastante eficiente em teoria, porém, para a memória do computador, é bastante complicado inserir vetores no meio de outros vetores, pois, para isto é necessário deslocar os elementos na memória.

Algoritmo

```
void shellsort(int *v, int size) {
	int i, j, increment, temp;
	increment = size / 2;

	while (increment > 0) {
	for (i = increment; i < size; i++)
	{
		j = i;
		temp = v[i];
		while ((j >= increment) && (a[j-increment] > temp))
		{
			v[j] = a[j - increment];
			j = j - increment;
		}
		v[j] = temp;
	}

	if (increment == 2)
		increment = 1;
	else
		increment = (int) (increment / 2.2);
	}
}
```

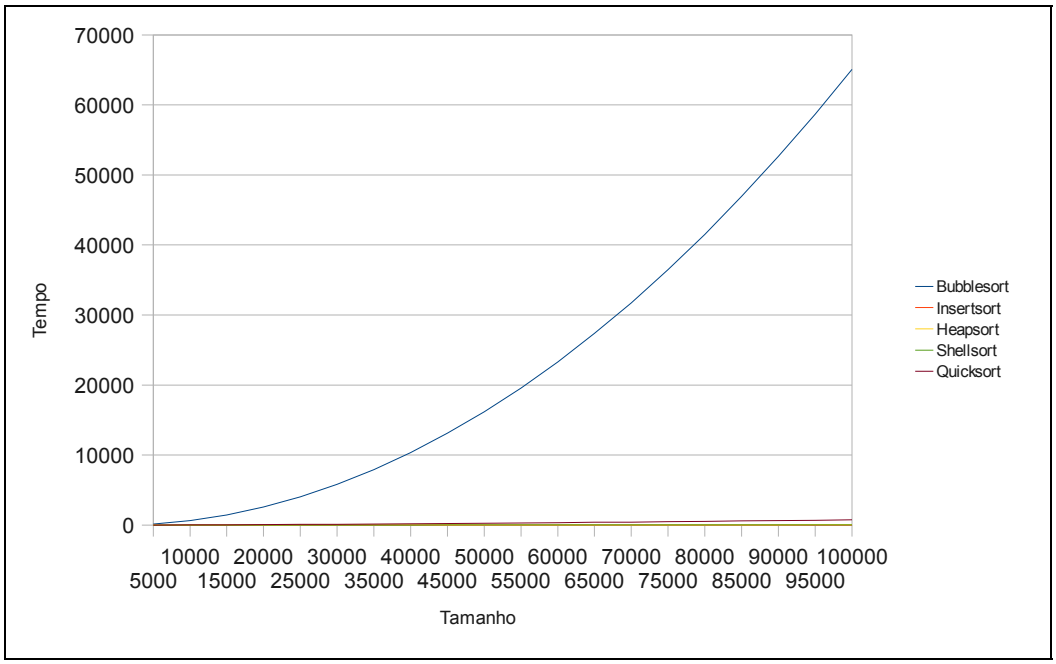
Quicksort

O algoritmo de ordenação mais utilizado atualmente. É extremamente eficiente em vetores bem aleatórios, o seu pior caso é bastante raro, e normalmente se dá quando a escolha dos pivôs é ruim. Normalmente age melhor em vetores desordenados do que em vetores já ordenados (crescentemente e inverso), pois o seu método de ordenação não se dá varrendo o vetor, mas sim dividindo o vetor em partes menores.

Algoritmo

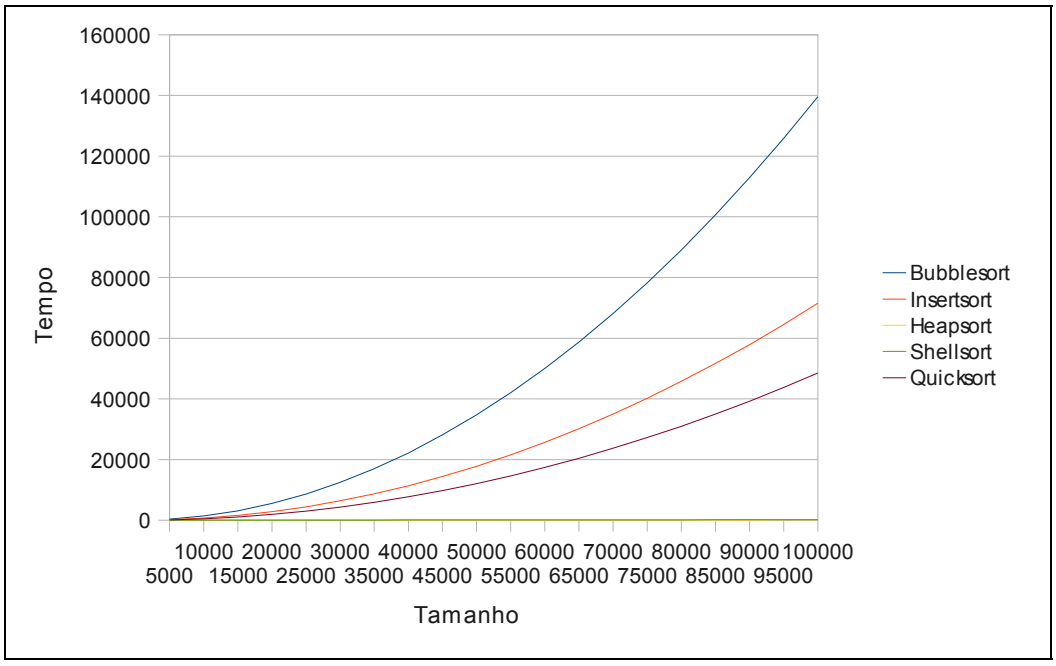
```
void quicksort(int *array, int begin, int end) {
	int x;

	if (end - begin > 1)
	{
		int pivot = array[begin];
		int l = begin + 1;
		int r = end;

		while (l < r) {
			if (array[l] <= pivot)
				l++;
			else
			{
				x = array[l];
				array[l] = array[r];
				array[r] = x;
				r--;
			}
		}
		l--;
		x = array[begin];
		array[begin] = array[l];
		array[l] = x;
		quicksort(array, begin, l);
		quicksort(array, r, end);
	}
}
```

Gráficos
Vetores Ordenados



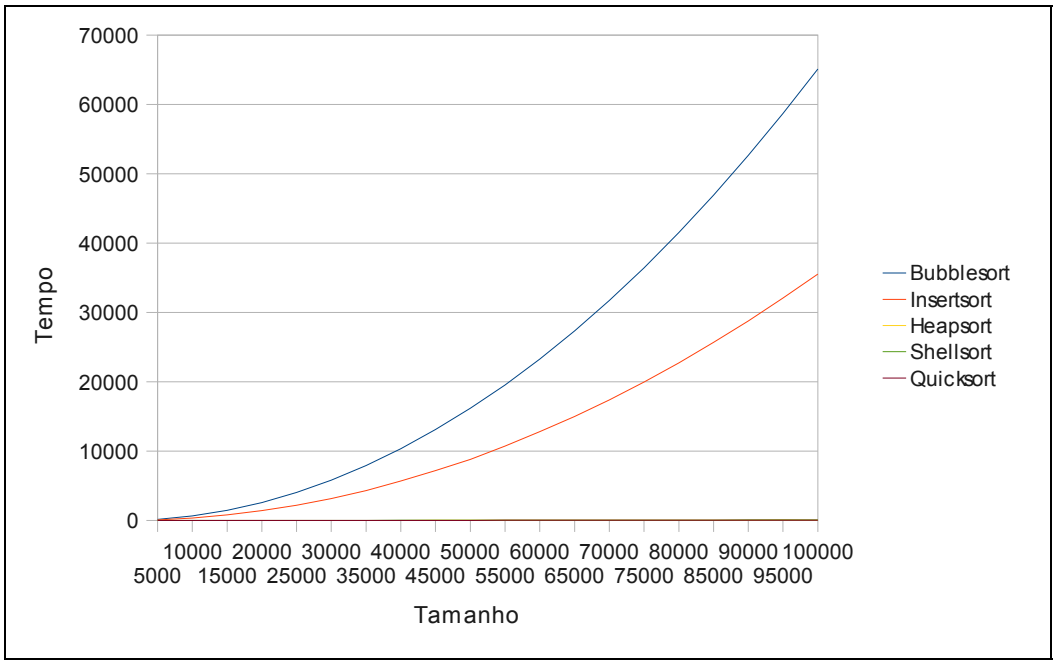
Em vetores ordenados, a diferença de tempo entre os algoritmos é mínima, com excessão do Bubblesort, que de qualquer maneira, varre o vetor de uma maneira pouco prática.

Vetores Inversos

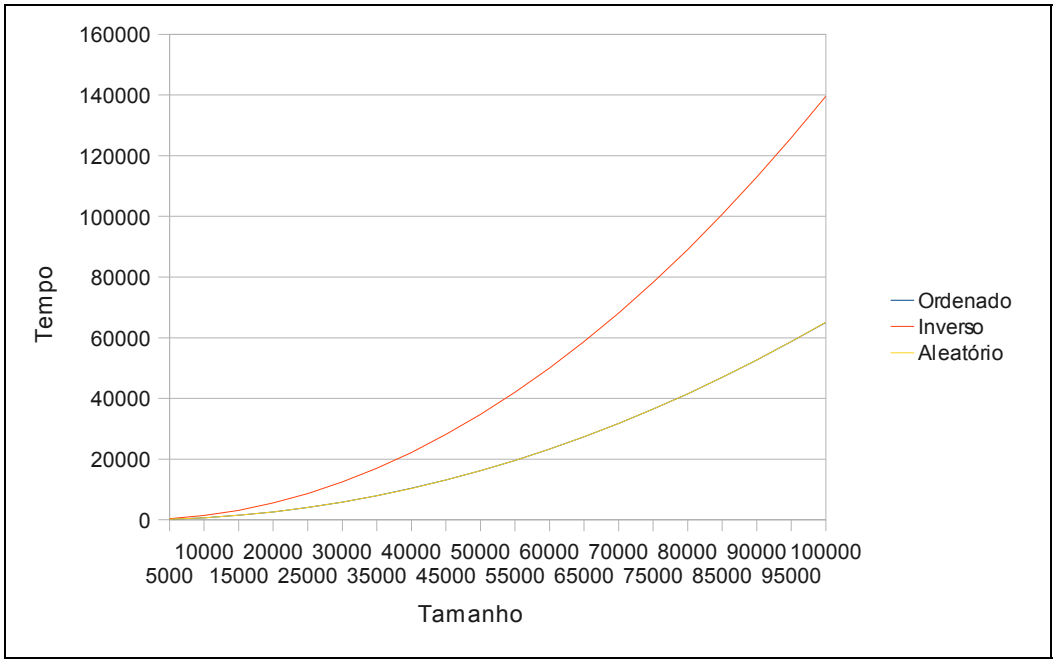


Em vetores inversamente ordenados, o HeapSort e o ShellSort são os que apresentam o melhor resultado, bem superior aos outros.

Vetores Aleatórios

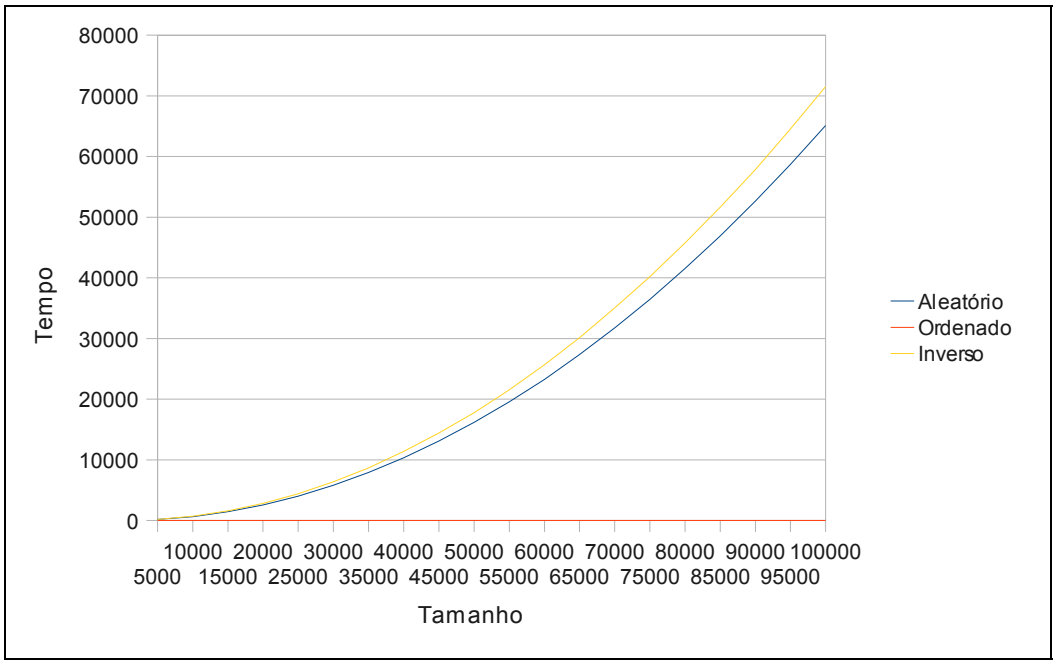


BubbleSort

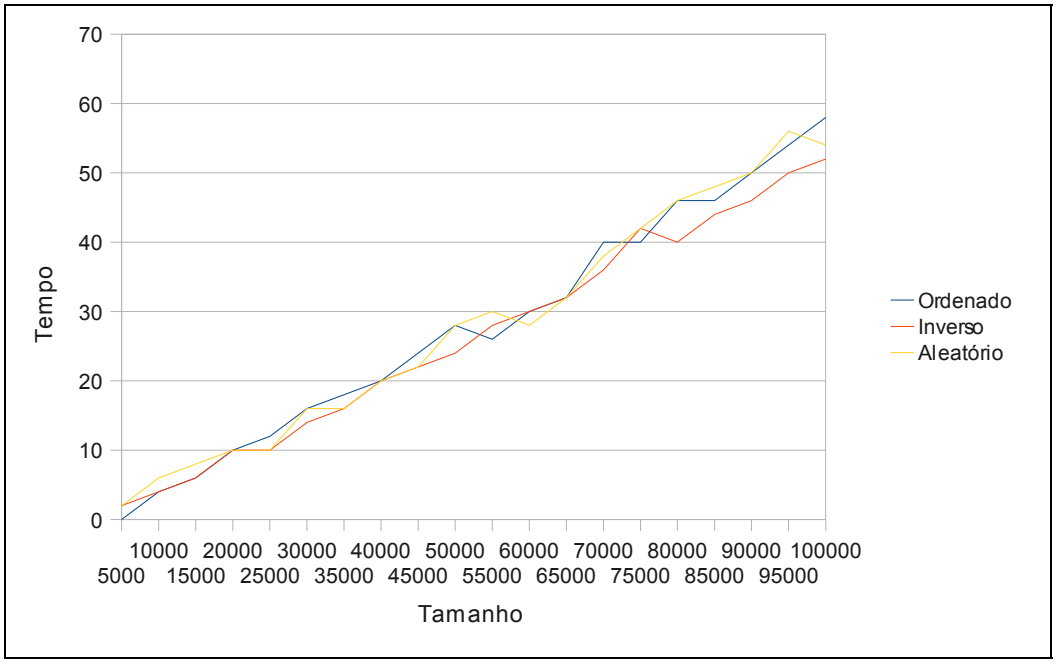


O interessante no Bubblesort é que não houve praticamente nenhuma variação no tempo de ordenação de um vetori ordenado para um aleatório. Já um inverso, tem uma curva de crescimento mais alta.

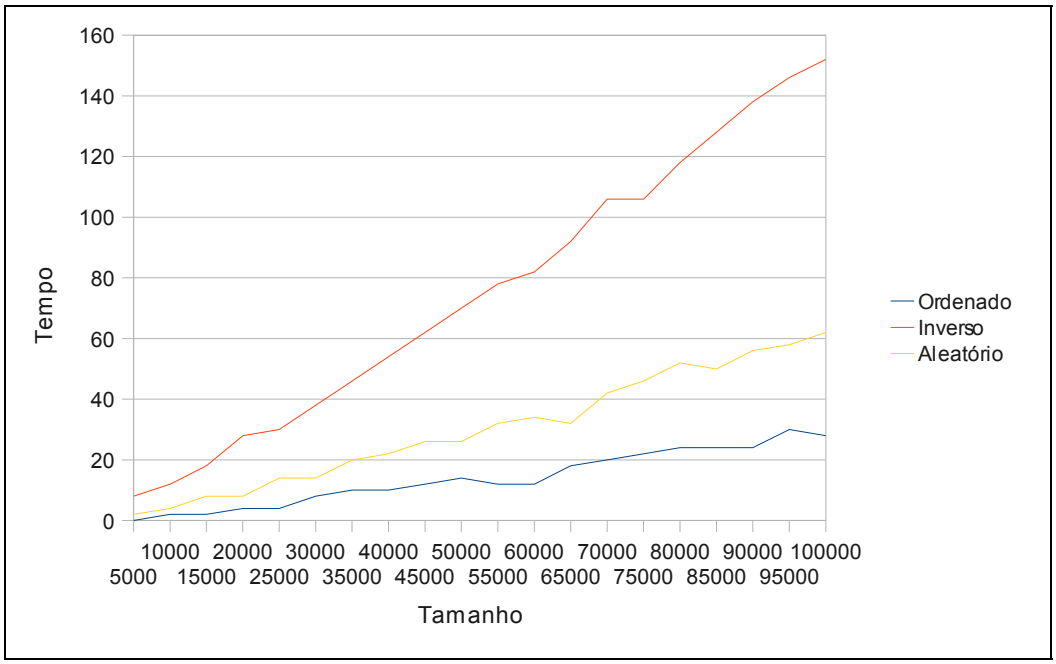
InsertSort



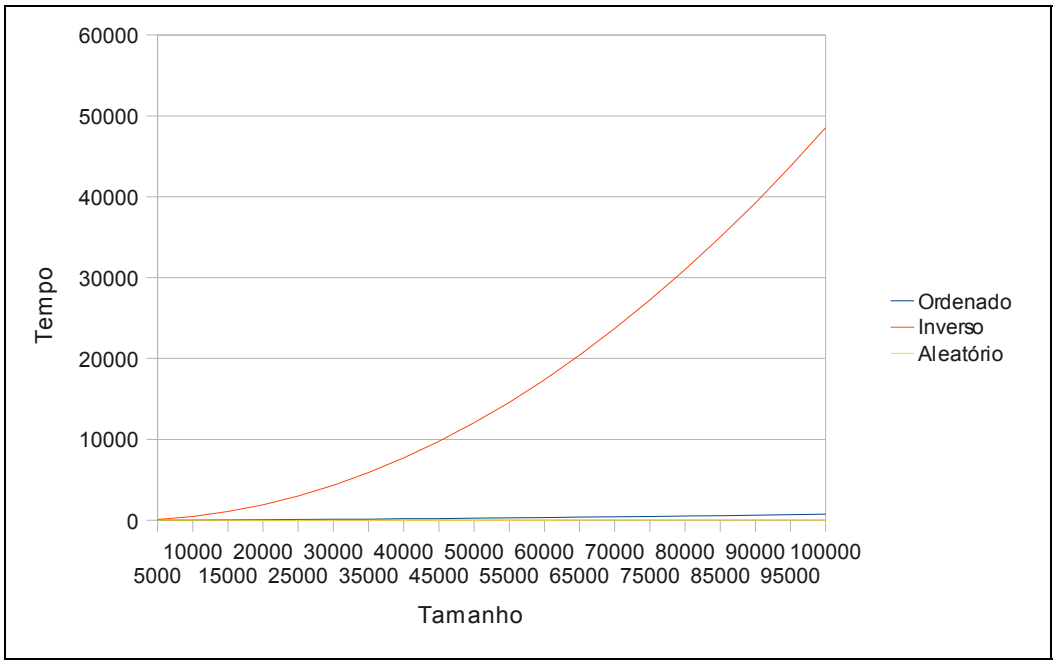
Heapsort



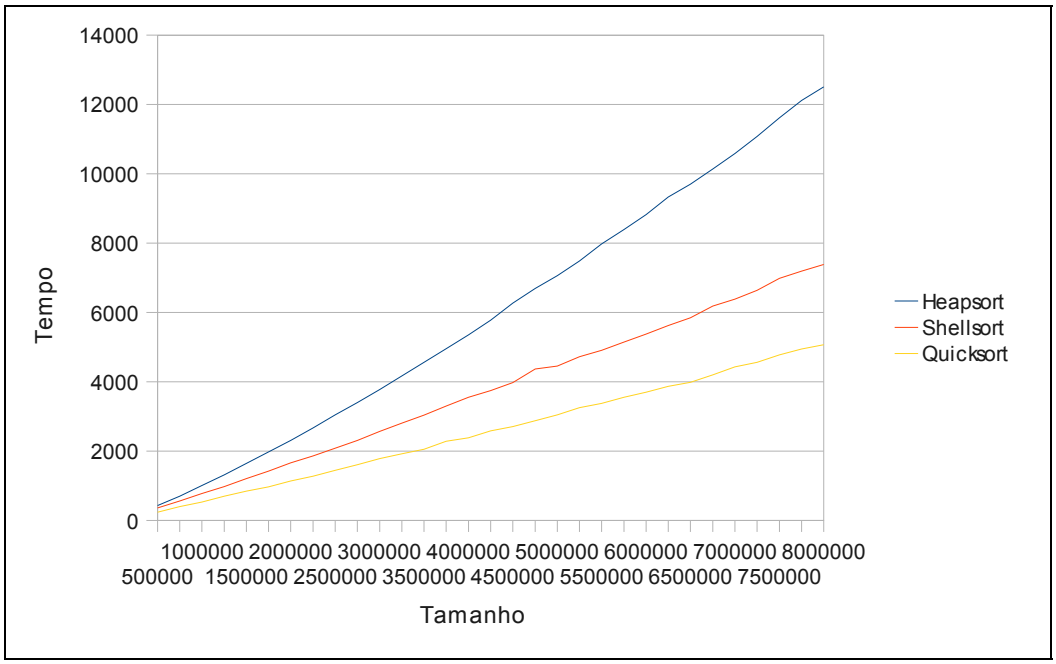
Shellsort



Quicksort



Heap x Shell x Quick – Valores Aleatórios (500mil – 8milhões)



Utilizando um grande número de elementos nos 3 algoritmos mais velozes, é possível verificar a superioridade do Quicksort.