Lista 2 de EDA 2

Arthur Luis Komatsu Aroeira 13/0102750

Pedro Kelvin de Castro M Batista 13/0129674

12 de setembro de 2017

Perguntas e Respostas

1. **Imagine um vetor onde o menor elemento está na última posição. Explique (use um diagrama de um vetor com 11 posições) como o Shellsort garante que esse elemento conseguirá chegar à sua posição final através de trocas sucessivas baseadas no gap.**

   O algoritmo se basea em particionar o vetor em vários segmentos de modo a varrer esse vetor e ir dividindo os grupos maiores em menores. No caso de um vetor com 11 posições, temos:

0	1	2	3	4	5	6	7	8	9	10
10	2	9	7	6	3	11	4	8	5	1

   O primeiro passo é comparar o elemento na sua posição atual com o elemento que dista N/2 da posição dele e compará-los para inserir o maior ao lado mais a direita. Ex.: o vetor possui tamanho 11, a metade inteira é 5. Então se compara o valor na posição zero ao elemento na posição 5; se compara o elemento na posição 1 ao elemento na posição 6; e assim por diante. Quando se chega na posição 10, o vetor está da seguinte forma:

0	1	2	3	4	5	6	7	8	9	10
3	2	4	7	5	10	11	9	8	6	1

   A próxima comparação a ser feita é o elemento na posição 5 com o elemento na posição 10. É feita a troca e o elemento da ultima posição é inserido no meio. Nesse ponto o algoritmo não se encerra, ele ainda faz mais uma comparação com o novo elemento na posição 5 e o primeiro elemento do vetor(posição 0). Esse é o momento em que o menor valor será inserido em sua posição correta.

0	1	2	3	4	5	6	7	8	9	10
1	2	4	7	5	3	11	9	8	6	10

   Ou seja, assim que se encontra um valor na sua posição errada, o vetor vai sendo varrido de trás para frente saltando nos gaps que ele já estava utilizando (no caso anterior o gap era 5). Ao acabar esse passo, de novo é divido pela metade o valor usado anteriormente. Agora a metade inteira vale $5/2 = 2$. As próximas comparações se darão entre o elemento em sua posição atual e o elemento na posição do elemento anterior mais 2. O vetor ficará da seguinte forma nesse passo:

0	1	2	3	4	5	6	7	8	9	10
1	2	4	3	5	6	8	7	10	9	11

Por fim, o novo valor a ser usado na partição do vetor será 1. Com isso, é fácil ver que serão feitas comparações sempre com elementos que estão um ao lado do outro, deixando todo o vetor ordenado. Em suma, o Shell Sort se comporta fazendo comparações entre pares de números, e esses pares são definidos através do tamanho do vetor. Quando em uma comparação é comprovado um valor em sua posição errada, as trocas são feitas e o vetor vai voltando saltando gaps e fazendo novas comparações, de modo que o menor valor sempre seja "jogado"para o começo.

2. **Implemente o algoritmo Quicksort escolhendo a mediana como pivô. Utilize o algoritmo QuickSelect + Mediano of Medians visto em sala.**

A seguir, o algoritmo implementado em C++ utilizando o algoritmo das medianas do professor:

```
#include <bits/stdc++.h>
using namespace std;

// sorts a small group with insertion sort
// group has 5 itens or less
void sort_group(vector<int>& numbers, int i, int group_size) {

        for (int j = i + 1; j < i + group_size; j++) {
                int k = j;
                while ((k > i) && (numbers[k] < numbers[k-1])) {
                        int aux = numbers[k-1];
                        numbers[k-1] = numbers[k];
                        numbers[k] = aux;
                        k--;
                }
        }
}

// breaks numbers into groups of five itens
// last group may be not complete
void divide_groups(vector<int>& numbers, int size) {

        int i = 0;

        /*  if size = 14
         *  two groups of 5 itens
         *  last group has 4 itens
         */

        for (i = 0; i <= (size - 5); i = i + 5) {
                sort_group(numbers, i, 5);
        }

        int last_size = size % 5; // last group size

        if (last_size > 0) {
                sort_group(numbers, i, last_size);
        }
        else {
                // do nothing, there is no incomplete group
        }
```

```
}


// copy median of each group to a second array
// caution with last group
void medians_copy(vector<int>& source, int source_size,vector<int>&
    ↪ destiny, int destiny_size) {

        int number_of_complete_groups = source_size / 5;
    int i = 0;

        // copy medians of complete groups
        for (i = 0; i < number_of_complete_groups; i++) {
                destiny[i] = source[i * 5 + 2];
        }

        // from now on, deals with last group

        int last_group_size = source_size % 5;

        if (last_group_size > 0) {

            // last group first position
            int first_position = i * 5;

                int last_median = 0;

                // defines position of last median accordingly to group
                    ↪ size
                // extract its value
                switch (last_group_size) {
                        case 1:
                        case 2:
                                last_median = source[first_position];
                                break;
                        case 3:
                        case 4:
                                last_median = source[first_position + 1];
                                break;
                        //case 5:
                                //last_median = source[first_position +
                                    ↪ 2];
                                //break;
                        default: // last group is not bigger than 4 itens
                                //assert(FALSE);
                                break;
                }

                // copy last median to second array
                destiny[destiny_size - 1] = last_median;
        }
        else {
                // do nothing, has only complete groups
        }
}

// copies left or right partition to a new array
vector<int>  partition_copy(vector<int> &source, int start, int end) {
```

```
        int destiny_size = (end + 1) - start;
        vector<int> destiny(destiny_size);

        for (int i = start; i <= end; i++) {
                destiny[i - start] = source[i];
        }

        return destiny;
}

// simple swap of two numbers of array
void swap(      vector<int> & input, int first_index, int second_index) {
        int aux = input[first_index];
        input[first_index] = input[second_index];
        input[second_index] = aux;
}

// finds first occurence of value
int find_index(vector<int>& input, int size, int value) {

        int value_index = -1;

        for (int i = 0; i < size; i++) {
                if (input[i] == value) {
                        value_index = i;
                        break;
                }
        }

        return value_index;
}

// smallers numbers to the left, bigger numbers to the right
// [small numbers][mom][bigger numbers]
// has problems with very small input, 1 or 2 itens
void partition_numbers_around_mom (vector<int>& input_numbers,
                                   int input_numbers_size,
                                                                   int
                                       ↪ median_of_median
                                       ↪ )
                                       ↪
                                       ↪ {
                                       ↪


    // short-circuit, fix me!
        if (input_numbers_size == 2) {
                if (input_numbers[0] > input_numbers[1]) {
                        swap(input_numbers, 0, 1);
                }
                return;
        }
        else if (input_numbers_size == 1) {
                return;
        }

        int mom_index = find_index(input_numbers, input_numbers_size,
            ↪ median_of_medians);
        swap(input_numbers, 0, mom_index);

        int i = 1;
```

```
        int j = input_numbers_size - 1;

        while (i < j) {

                // finds a greater in the left side
                while (input_numbers[i] < median_of_medians) {
                        i++;
                }

                // finds a smaller in the right side
                while (input_numbers[j] > median_of_medians) {
                        j--;
                }

                swap(input_numbers, i, j);
        }

        // both numbers on the middle are inverted
        // mom is at index 0
        swap(input_numbers, 0, i-1);
        swap(input_numbers, 0, i);

}

// oracle implements Median of Medians algorithm
int ask_oracle_for_median(vector<int> &input_numbers, int
    ↪ input_numbers_size) {

        divide_groups(input_numbers, input_numbers_size);

        int last_medians_size = input_numbers_size / 5;

        if ((input_numbers_size % 5) > 0) {
                last_medians_size = last_medians_size + 1;
        }
        else {
                // do nothing
        }

    vector<int> last_medians(last_medians_size);

        medians_copy(input_numbers, input_numbers_size, last_medians,
            ↪ last_medians_size);

        while (last_medians_size > 1) {


                divide_groups(last_medians, last_medians_size);


                int current_medians_size = 0;

                if ((last_medians_size % 5) > 0) {
                        current_medians_size = (last_medians_size / 5) +
                            ↪ 1;
                }
                else {
                        current_medians_size = last_medians_size / 5;
                }
```

```
            vector<int> current_medians (current_medians_size);

                medians_copy(last_medians, last_medians_size,
                    ↪ current_medians, current_medians_size);

                //free(last_medians);

                last_medians = current_medians;
                last_medians_size = current_medians_size;
        }

        return last_medians[0]; // this is the Median of Medians
}

int exoteric_select (vector<int>&input_numbers, int input_numbers_size,
    ↪ int k_esimo) {


        int mom = ask_oracle_for_median(input_numbers, input_numbers_size)
            ↪ ;

        partition_numbers_around_mom(input_numbers, input_numbers_size,
            ↪ mom);
        int mom_index = find_index(input_numbers, input_numbers_size, mom)
            ↪ ;

        int expected_size = k_esimo;
        int median = -1;

        if (mom_index > expected_size) {

                // oracle guessed too high

                int left_size = mom_index;

                vector<int> left_partition = partition_copy(input_numbers,
                    ↪  0, left_size - 1);

                median = exoteric_select(left_partition, left_size,
                    ↪ k_esimo);
        }
        else if (mom_index < expected_size) {

                // oracle guessed too low


                int right_size = (input_numbers_size - 1) - mom_index;
                int partition_start = mom_index + 1;

                vector<int> right_partition = partition_copy(input_numbers
                    ↪ , partition_start, input_numbers_size - 1);

                k_esimo = k_esimo - (mom_index + 1);
                median = exoteric_select(right_partition, (
                    ↪ input_numbers_size - 1) - mom_index, k_esimo);
        }
        else {

                // oracle is correct!
```

```
                median = mom;
        }

        return median;
}

void quickSort(vector<int>& A, int lo = 0, int hi = -1)
{
        if(hi == -1)
                hi = A.size();
        if(lo < hi)
        {
                vector<int> AA = A;
            vector<int> aux = partition_copy(AA, 0, AA.size() - 1);
                sort_group(aux, 0, AA.size()); // sort_group also works
                    ↪ with large groups
                int k_esimo = (AA.size() - 1) / 2; // median is at middle
                int median = exoteric_select(AA, AA.size(), k_esimo);
                int i = lo;

                for(int j = lo + 1; j < hi; j++)
                        if(A[j] <= A[lo])//A[lo] = pivot
                        {
                                i++;
                                swap(A[i], A[j]);
                        }
                swap(A[i], A[lo]);
                quickSort(A, lo, i);
                quickSort(A, i+1, hi);
        }
}

int main(){

    vector<int> v =
        ↪ {2,5,9,19,24,54,5,87,9,10,44,32,18,13,2,4,23,26,16,19,25,39,47,56,71};
        ↪
    quickSort(v);

    for(int i = 0; i < v.size(); ++i)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

3. **Implemente o Heapsort das duas formas: o modo normal, que utiliza um único vetor, e a segunda forma, como uma estrutura separada onde a ordenação é feita inserindo todos os dados e depois removendo todos. Cronometre a execução e trace um gráfico. Qual a mais rápida? A taxa de crescimento é a mesma?**

   No código a seguir, a função CriarHeap cria a estrutura heap do vetor v do começo, inserindo-se elemento por elemento. A cada inserção, a estrutura é ajustada olhando-se os pais e realizando trocas. Este processo faz $O(log(n))$ operações (pois o pai está na metade do índice) e, como são inseridos n elementos, a estrutura é montada em $O(nlog(n))$. Para retirar os elementos, retira-se o pai de todos (menor) e o substitui por uma sentinela (INF) e ajusta todos os filhos ($O(log(n))$). Como n elementos são

retirados, a complexidade é $O(nlog(n))$.

Já o HeapSort() implementa a ordenação no local do vetor e realiza as mesmas operações descritas acima. A complexidade é a mesma.

Foram testadas vetores com tamanhos com múltiplos de 2. xA seguir, o código implementado em C++:

```
#include <bits/stdc++.h>
using namespace std;

#define debug(x) cerr << fixed << #x << " is " << x << endl;
#define INF 2147483647 //2^31-1

void CriarHeap(vector<int> v)
{
    int n = v.size();
    vector<int> heap(n);

    for(int i = 0 ; i < n ; i++)
    {
        heap[i] = v[i];
        int pai = (i - 1)/2;
        int atual = i;

        while (heap[pai] > heap[atual] && atual != pai)
        {
            swap(heap[pai], heap[atual]);
            int aux = pai;
            pai = (pai - 1)/2;
            atual = aux;
        }
    }

    //Printar em ordem
    for(int i = 0 ; i < n ; i++)
    {
        //cout << heap[0] << " "; //-> printar primeiro elemento (menor)
        int atual = 0;
        heap[0] = INF;//sentinela: infinito

        do//remover primeiro elemento:descer primeiro elemento e subindo
            ↪ os menores
        {
            int l = 2 * atual + 1;
            int r = 2 * atual + 2;
            if(l == n - 1)
            {
                swap(heap[atual], heap[l]);
                atual = l;
            }
            else if(l > n)
                break;
            else if(heap[l] < heap[r])
            {
                swap(heap[atual], heap[l]);
                atual = l;
            }
            else if(r < n)
            {
                swap(heap[atual], heap[r]);
                atual = r;
```

```
            }
        }while(atual < n || heap[atual] != INF);
    }
}

void HeapifyNoVetor(vector<int> &v, int n, int i)
{
    int atual = i;
    int l = 2*i + 1;
    int r = 2*i + 2;

    if (n >l && v[l] > v[atual])
        atual = l;
    if (n >r && v[r] > v[atual])
        atual = r;
    if (atual != i)
    {
        swap(v[i], v[atual]);
        HeapifyNoVetor(v, n, atual);
    }
}

void HeapSort(vector<int> &v, int n)
{
    //Montar o heap
    for (int i = n / 2 - 1; i >= 0; i--)
        HeapifyNoVetor(v, n, i);

    //Extrair um por um
    for (int i = n - 1; i >= 0; i--)
    {
        swap(v[0], v[i]);
        HeapifyNoVetor(v, i, 0);
    }
}

int main()
{
    int tam = 2; //tamanho dos elementos
    for(int j = 0 ; j < 25 ; j++)
    {
        vector<int> v;
        for(int i = 0 ; i < tam ; i++)
            v.push_back(rand() % 100000);

        auto begin = chrono::high_resolution_clock::now(); //comeca o
            ↪ clock
        CriarHeap(v); //Criando a estrutura Heap e depois removendo em
            ↪ ordem
        auto end = chrono::high_resolution_clock::now();//termina o clock
        cout << tam << " elementos (Criando o Heap): " << (long double)
            ↪ chrono::duration_cast<chrono::nanoseconds>(end-begin).count
            ↪ ()/1000000000 << " s" << endl;

        begin = chrono::high_resolution_clock::now(); //comeca o clock
        HeapSort(v, v.size()); //Heap Sort no Vetor
        end = chrono::high_resolution_clock::now();//termina o clock
        cout << tam << " elementos (Heap no vetor): " << (long double)
            ↪ chrono::duration_cast<chrono::nanoseconds>(end-begin).count
            ↪ ()/1000000000 << " s" << endl;
```

```
        tam *= 2; //dobrar elemento
    }
    return 0;
}
```

O Output do programa foi:

```
2 elementos (Criando o Heap): 7.08e-07 s
2 elementos (Heap no vetor): 1.27e-07 s
4 elementos (Criando o Heap): 4.21e-07 s
4 elementos (Heap no vetor): 2.18e-07 s
8 elementos (Criando o Heap): 5.75e-07 s
8 elementos (Heap no vetor): 3.54e-07 s
16 elementos (Criando o Heap): 9.26e-07 s
16 elementos (Heap no vetor): 5.96e-07 s
32 elementos (Criando o Heap): 2.218e-06 s
32 elementos (Heap no vetor): 1.334e-06 s
64 elementos (Criando o Heap): 3.661e-06 s
64 elementos (Heap no vetor): 2.568e-06 s
128 elementos (Criando o Heap): 8.106e-06 s
128 elementos (Heap no vetor): 5.244e-06 s
256 elementos (Criando o Heap): 1.7624e-05 s
256 elementos (Heap no vetor): 1.1695e-05 s
512 elementos (Criando o Heap): 3.7352e-05 s
512 elementos (Heap no vetor): 2.4383e-05 s
1024 elementos (Criando o Heap): 8.1938e-05 s
1024 elementos (Heap no vetor): 5.2307e-05 s
2048 elementos (Criando o Heap): 0.000176682 s
2048 elementos (Heap no vetor): 0.000115217 s
4096 elementos (Criando o Heap): 0.000378025 s
4096 elementos (Heap no vetor): 0.000292694 s
8192 elementos (Criando o Heap): 0.000805768 s
8192 elementos (Heap no vetor): 0.000565352 s
16384 elementos (Criando o Heap): 0.0017496 s
16384 elementos (Heap no vetor): 0.00125204 s
32768 elementos (Criando o Heap): 0.0038121 s
32768 elementos (Heap no vetor): 0.00279048 s
65536 elementos (Criando o Heap): 0.00817947 s
65536 elementos (Heap no vetor): 0.00638157 s
131072 elementos (Criando o Heap): 0.0147599 s
131072 elementos (Heap no vetor): 0.0111779 s
262144 elementos (Criando o Heap): 0.0312989 s
262144 elementos (Heap no vetor): 0.027346 s
524288 elementos (Criando o Heap): 0.0665442 s
524288 elementos (Heap no vetor): 0.0632309 s
1048576 elementos (Criando o Heap): 0.141685 s
1048576 elementos (Heap no vetor): 0.144281 s
2097152 elementos (Criando o Heap): 0.431237 s
2097152 elementos (Heap no vetor): 0.365186 s
4194304 elementos (Criando o Heap): 1.01298 s
4194304 elementos (Heap no vetor): 1.0986 s
8388608 elementos (Criando o Heap): 2.31939 s
8388608 elementos (Heap no vetor): 3.00324 s
16777216 elementos (Criando o Heap): 5.20068 s
16777216 elementos (Heap no vetor): 8.2275 s
33554432 elementos (Criando o Heap): 12.8349 s
33554432 elementos (Heap no vetor): 21.1269 s
```

O gráfico do número de operações x tempo ficou:

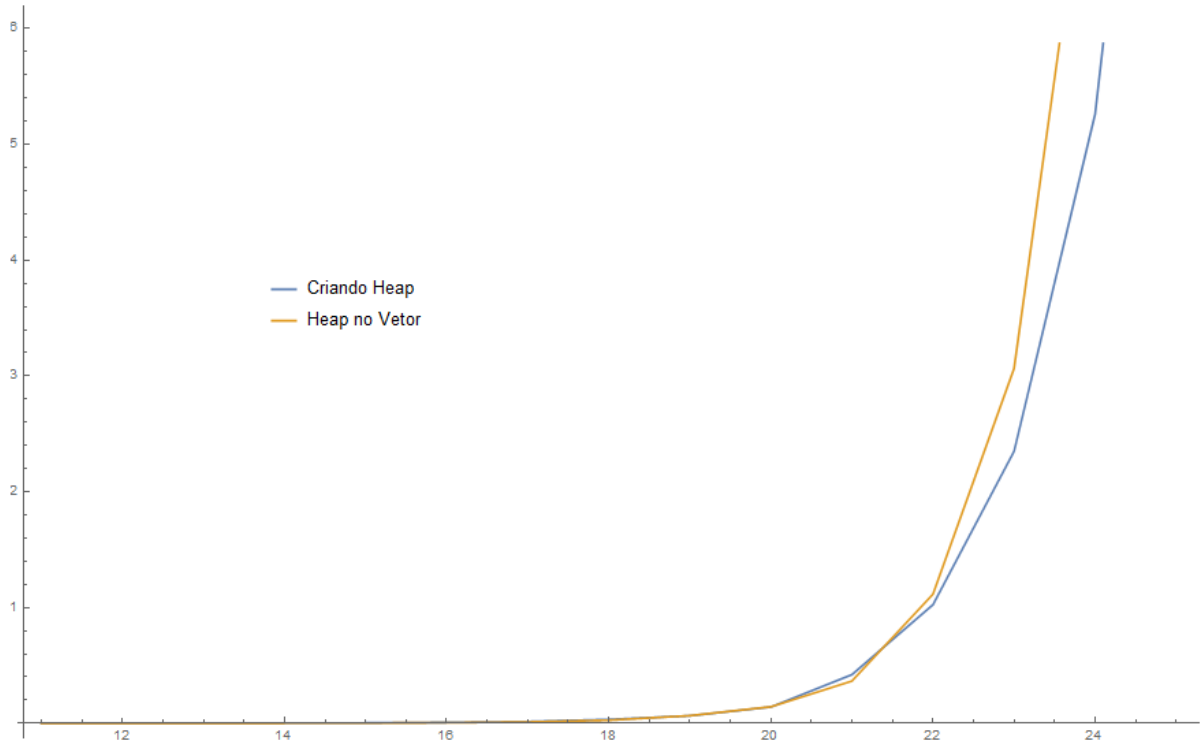

Figura 1 – Gráfico (tamanho x tempo) heap sort

Analisando os dados e o gráfico, nota-se que, com menos de 1000000 elementos, o heapsort no local é mais eficiente do que implementar separadamente. Já a partir de 1000000 de elementos, criando-se a estrutura Heap separado e retirando os elementos 1 por 1 é mais rápido do que fazer no vetor. Porém, gasta-se memória a mais.

4. **Implemente um algoritmo que misture o Counting sort e o Radix sort para ordenar um vetor de nomes. Utilize qualquer lista pública de nomes. Cronometre a execução e trace um gráfico. A taxa de crescimento é mesmo linear?**

   O código foi implementado e testado com vetores de tamanhos de potência de 2 até 8000000. A diferença na implementação com strings para inteiros, foi a adição de um mapa para realizar o histograma. Para o o caractere mínimo do histograma, utilizou-se o (' ' - 1), o qual foi o menor caractere imprimível da tabela ASCII e o máximo foi ('z') pelo mesmo motivo.

   A seguir, o código implementado para ordenar um vetor de strings usando radix sort:

```
#include <bits/stdc++.h>
using namespace std;

#define debug(x) cerr << fixed << #x << " is " << x << endl;

void CountSort(vector<string> &v, int ind)
{
        int n = v.size();
        vector<string> result(n);
        map<char, int> hist; //histograma
```

```
        for (int i = 0; i < n; i++)
                if(ind < v[i].size())
                        hist[v[i][ind]]++;
                else
                        hist[' ' - 1]++;

        for (char c = ' ' - 1; c <= 'z'; c++)
                hist[c] += hist[c - 1];

        for (int i = n - 1 ; i >= 0 ; i--)
        {
                if(ind < v[i].size())
                {
                        result[hist[ v[i][ind] ] - 1] = v[i];
                        hist[ v[i][ind] ]--;
                }
                else
                {
                        result[hist[ ' '-1  ] - 1] = v[i];
                        hist[ ' '-1 ]--;
                }
        }

        for (int i = 0 ; i < n ; i++)
                v[i] = result[i];
}

void RadixSort(vector<string> &v)
{
        int n = v.size();
        int maior = v[0].size();
        for (int i = 1; i < v.size(); i++)
                if (v[i].size() > maior)
                        maior = v[i].size();

        for (int ind = maior - 1; ind >= 0; ind--)
                CountSort(v, ind);
}


int main()
{
        string s;
        vector<string> v;
        while(cin >> s)
                v.push_back(s);
        for(int i = 1 ; i < 10000000 ; i *= 2)
        {
                vector<string> aux;
                for(int j = 0 ; j < i ; j++)
                        aux.push_back(v[i]);
        auto begin = chrono::high_resolution_clock::now(); //
            ↪ comeca o clock
        RadixSort(aux);
        auto end = chrono::high_resolution_clock::now();//termina
            ↪ o clock
        cout << i << " elementos (Radix Sort): " << (long double)
            ↪ chrono::duration_cast<chrono::nanoseconds>(end-
            ↪ begin).count()/1000000000 << " s" << endl;
```

```
        }

        return 0;
    }
```

O Output do programa foi:

```
1 elementos (Radix Sort): 6.3192e-05 s
2 elementos (Radix Sort): 0.000136185 s
4 elementos (Radix Sort): 5.5001e-05 s
8 elementos (Radix Sort): 0.000159653 s
16 elementos (Radix Sort): 0.000138099 s
32 elementos (Radix Sort): 9.0459e-05 s
64 elementos (Radix Sort): 0.000145646 s
128 elementos (Radix Sort): 0.000142812 s
256 elementos (Radix Sort): 0.000152485 s
512 elementos (Radix Sort): 0.000184636 s
1024 elementos (Radix Sort): 0.000316686 s
2048 elementos (Radix Sort): 0.000619167 s
4096 elementos (Radix Sort): 0.00091275 s
8192 elementos (Radix Sort): 0.00278447 s
16384 elementos (Radix Sort): 0.00353023 s
32768 elementos (Radix Sort): 0.00611093 s
65536 elementos (Radix Sort): 0.0211001 s
131072 elementos (Radix Sort): 0.0451137 s
262144 elementos (Radix Sort): 0.0614839 s
524288 elementos (Radix Sort): 0.152673 s
1048576 elementos (Radix Sort): 0.277234 s
2097152 elementos (Radix Sort): 0.638358 s
4194304 elementos (Radix Sort): 0.466352 s
8388608 elementos (Radix Sort): 2.81867 s
```

Foi montado uma tabela tamanho do vetor x tempo de execução mostrada a seguir:

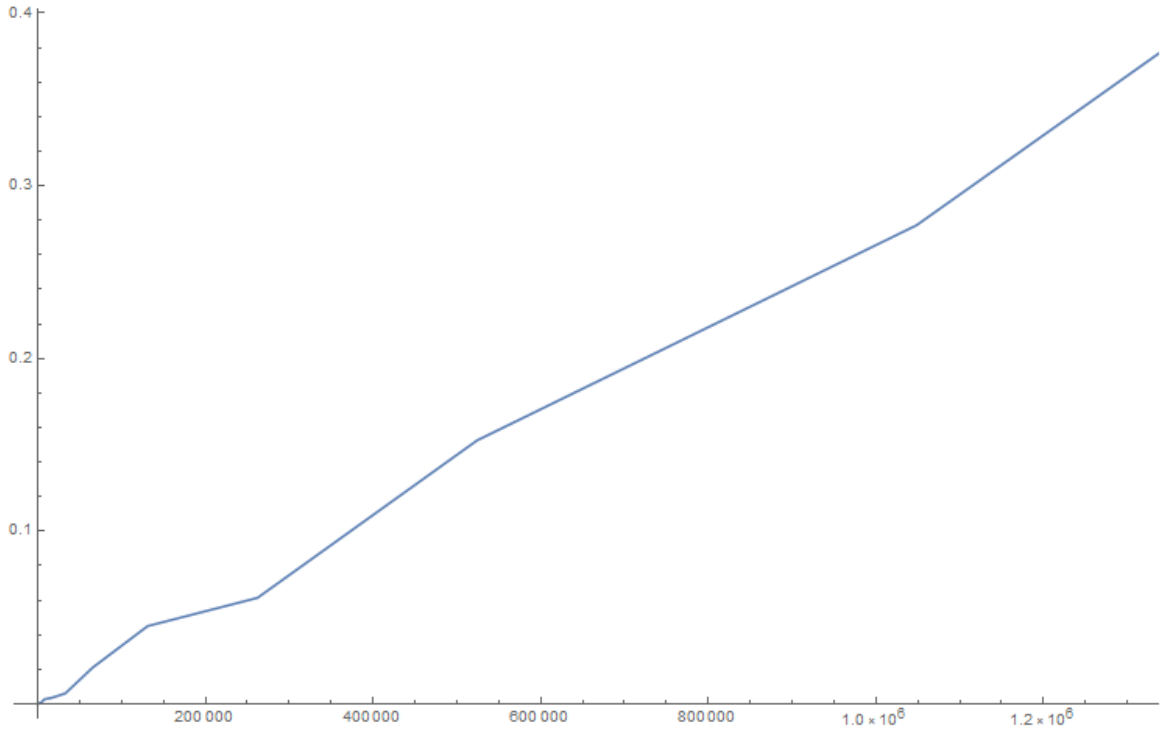

Figura 2 – Gráfico (tamanho x tempo) do radix sort

Observa-se que, como esperado, o tempo de execução foi aproximadamente linear.