Trabalho I – IC2

João Roberto de Moraes Neto nº14801159

Isabeli Gomes Oliveira nº14761812

a) Segue os algoritmos contidos nos arquivos .cpp:

Bubblesort:

```cpp
void bubbleSort(std::vector<std::string>& codes, int *com, int *mov){
    int n = codes.size();
    std::string aux;
    for(int i = 1; i < n; i++){
        for(int j = n - 1; j >= i; j--){
            if(codes[j - 1] > codes[j]){
                aux = codes[j - 1];
                codes[j - 1] = codes[j];
                codes[j] = aux;
                (*mov) += 3;
            }
            (*com)++;
        }
    }
}
```

Mergesort:

```cpp
void merge(std::vector<std::string>& a, int L, int h, int R, std::vector<std::string>& c, int* comp, int* mov){
   int i = L;
   int j = h + 1;
   int k = L - 1;

   while(i <= h && j <= R){
       k++;
       if(a[i] < a[j]){
           c[k] = a[i];
           (*mov)++;
           i++;
       }
       else{
           c[k] = a[j];
           (*mov)++;
           j++;
       }
       (*comp)++;

   }
```

```
    while(i <= h){
        k++;
        c[k] = a[i];
        (*mov)++;
        i++;
    }
    while(j <= R){
        k++;
        c[k] = a[j];
        (*mov)++;
        j++;
    }
}
```

```
void mpass(std::vector<std::string>& a, int n, int p, std::vector<std::string>& c, int* comp, int* mov){
    int i = 0;
    while(i <= n - 2 * p){
        merge(a, i, i + p - 1, i + 2 * p - 1, c, comp, mov);
        i = i + 2 * p;
    }
    if(i + p - 1 < n){
        merge(a, i, i + p - 1, n - 1, c, comp, mov);
    }
    else{
        for(int j = i; j <= n - 1; j++){
            c[j] = a[j];
            (*mov)++;
        }
    }
}
```

```
void mergeSort(std::vector<std::string>& a, int* comp, int* mov){
    std::vector<std::string> c;
    c.resize(a.size());
    int n = a.size();
    int p = 1;

    while(p < n){
        mpass(a, n, p , c, comp, mov);
        p *= 2;
        mpass(c, n, p, a, comp, mov);
        p *= 2;
    }
}
```

Heapsort:

```
void heapify(std::vector<std::string>& array, int L, int R, int* comp, int* mov){
    int i = L;
    int j = 2 * L;
    std::string x = array[L];

    if (j < R && array[j] < array[j + 1]) {
        j++;
    }
    (*comp)++;
    while ((j <= R) && (x < array[j])) {
        array[i] = array[j];
        (*mov)++;
        i = j;
        j = 2*j;
       if (j < R && array[j] < array[j + 1]) {
            j++;
        }
        (*comp)++;
    }

    array[i] = x;
    (*mov)++;
}
```

```
void heapSort(std::vector<std::string>& array, int n, int* comp, int* mov){
    for(int L = n/2; L >= 0; L--){
        heapify(array, L, n, comp, mov);
    }
    for(int R = n-1; R >= 1; R--){
        std::string w = array[0];
        array[0] = array[R];
        array[R] = w;
        (*mov) += 3;
        heapify(array, 0, R-1, comp, mov);
    }
}
```

Binary Insertionsort:

```
void insertBinarySort(std::vector<std::string>& codes, int* comp, int* mov){
    int n = codes.size();
    for(int i = 1; i < n; i++){
        std::string key = codes[i];
        int left = 0;
        int right = i;

        while(left < right){
            int mid = (right + left) / 2;

            if(codes[mid] <= key){
                left = mid + 1;
            }else{
                right = mid;
            }
            (*comp)++;
        }
        int j = i;
        while(j > right){
            codes[j] = codes[j-1];
            (*mov)++;
            j--;
        }
        codes[right] = key;
        (*mov)++;
    }
}
```

Insertionsort:

```
void insertionSort(std::vector<std::string>& codes, int* comp, int* mov){
    int n = codes.size();
    for(int i = 1; i < n; i++){
        std::string key = codes[i];
        int j = i - 1;

        while(j >= 0 && codes[j] > key){
            (*comp)++;
            codes[j + 1] = codes[j];
            (*mov)++;
            j = j - 1;
        }
        (*comp)++;
        codes[j + 1] = key;
        (*mov)++;
    }
}
```

Quicksort:

```
int qsort(std::vector<std::string>& array, int low, int high, int* comp, int* mov){
    int i = low, j = high;
    std::string pivo = array[(low+high)/2], aux;

    do{
        while(array[i] < pivo){
            (*comp)++;
            i++;
        }
        (*comp)++;
        while(pivo < array[j]){
            (*comp)++;
            j--;
        }
        (*comp)++;
        if(i<=j){
            aux = array[i];
            array[i] = array[j];
            array[j] = aux;
            (*mov) += 3;
            i++;
            j--;
        }
    }while(i<=j);
    if(low < j){
        qsort(array, low, j, comp, mov);
    }
    if(i < high){
        qsort(array, i, high, comp, mov);
    }

    }
```

```
void quickSort(std::vector<std::string>& array, int n, int* comp, int* mov){
    qsort(array, 0, n-1, comp, mov);
}
```

Selectionsort:

```
void selectionSort(std::vector<std::string>& codes, int *comp, int *mov){
    int n = codes.size();
    std::string aux;
    for(int i = 0; i < n - 2; i++){
        int minIndex = i;
        for(int j = i + 1; j < n - 1; j++){
            if(codes[j] < codes[minIndex]){
                minIndex = j;
            }
            (*comp)++;
        }
        aux = codes[i];
        codes[i] = codes[minIndex];
        codes[minIndex] = aux;
        (*mov) += 3;
    }
}
```

b) Os números de comparações e movimentações de cada algoritmo para cada arquivo mes_i.txt estão localizados no arquivo CompMov.txt

c)

Gráfico de comparações:

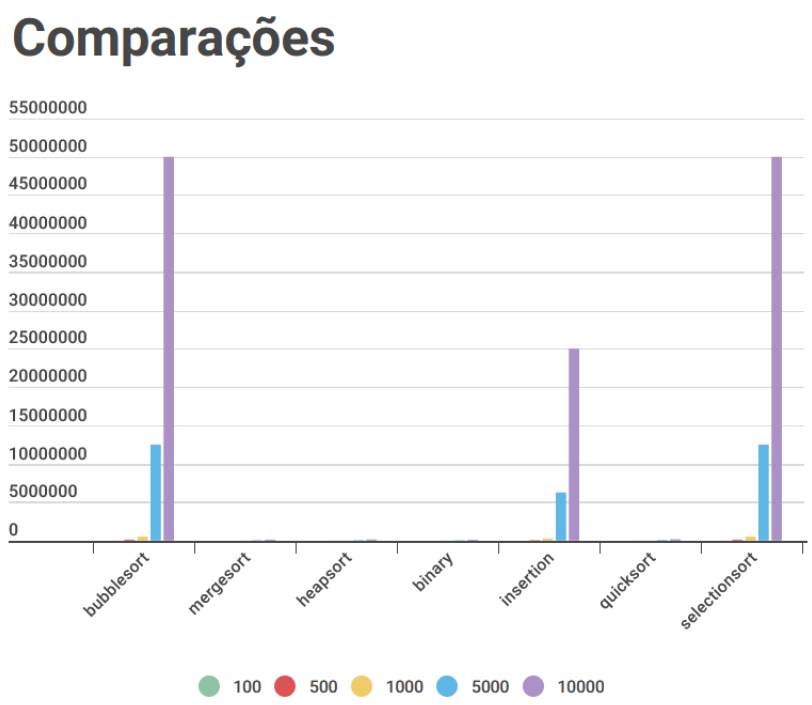
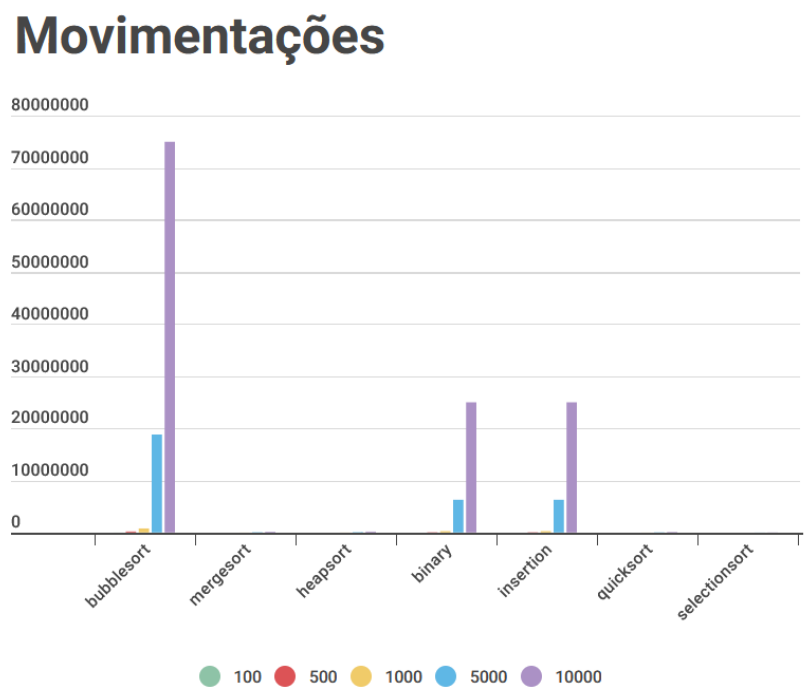


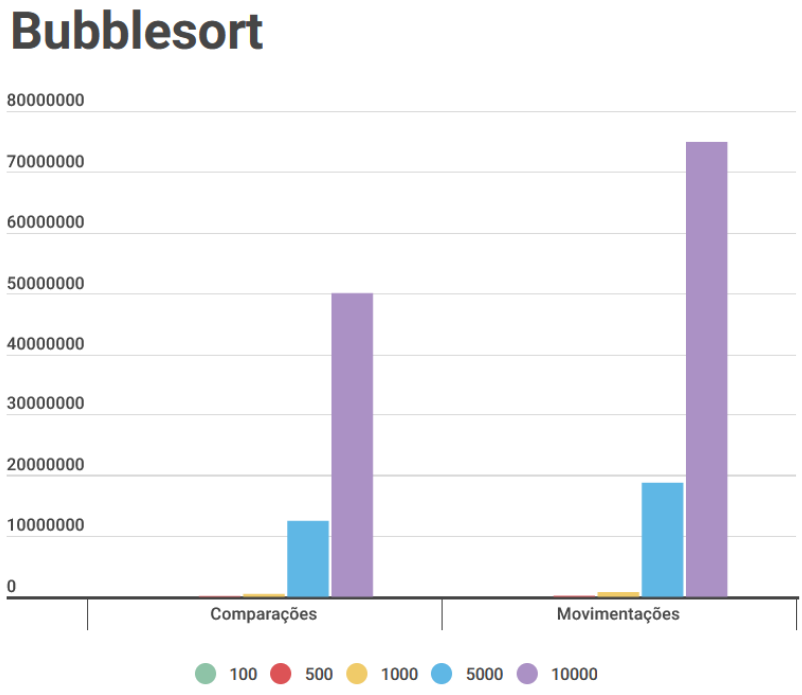
Gráfico de Movimentações:



Percebe-se que pela diferença visível entre as movimentações e comparações máximas dos algoritmos, alguns deles não possuem valores visíveis nos gráficos. Por isso fizemos também, gráficos separados para cada algoritmo.

d) Analise dos algoritmos:

Bubblesort:



Ao analisar o gráfico do algoritmo Bubblesort, percebe-se que o crescimento em suas comparações é semelhante ao de suas movimentações, isso pode ser justificado por causa de suas complexidades serem iguais, $O(N^2)$, exceto no caso mínimo, o que pode ser comprovado analisando o número exato de comparações e movimentações:

Ao dividir o número de comparações por N^2 deve-se obter um valor constante (com pequenas variações) que se repete pra todos os valores de N, o mesmo ocorre com as movimentações:

N = 100 => comp = 0,505 | Mov = 0,7257

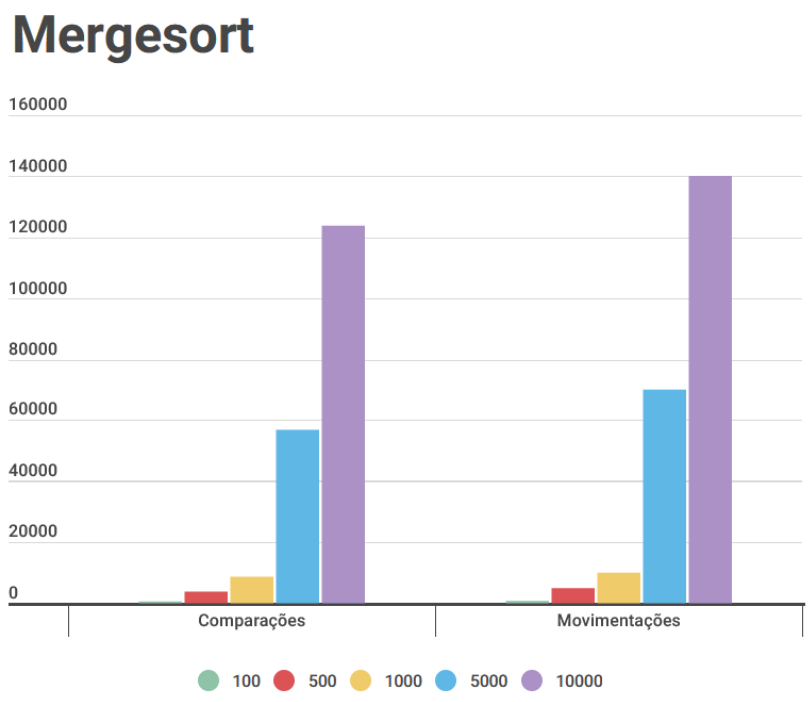
N = 500 => comp = 0,501 |Mov = 0,7586

N = 1000 => comp = 0,500 |Mov = 0,7676

N = 5000 => comp = 0,500 |Mov = 0,7515

N = 10000 => comp = 0,500 |Mov = 0,7491

Mergesort:



Ao analisar o gráfico do algoritmo Mergesort, percebe-se que o crescimento em suas comparações é semelhante ao de suas movimentações, assim como o bubblesort, porém sua a complexidade é $O(N*\log_2 N)$, o que pode ser comprovado analisando o número exato de comparações e movimentações:

Ao dividir o número de comparações por $N*\log_2 N$ deve-se obter um valor constante (com pequenas variações) que se repete pra todos os valores de N, o mesmo ocorre com as movimentações:

N = 100 => comp = 0,862 | Mov = 1,216

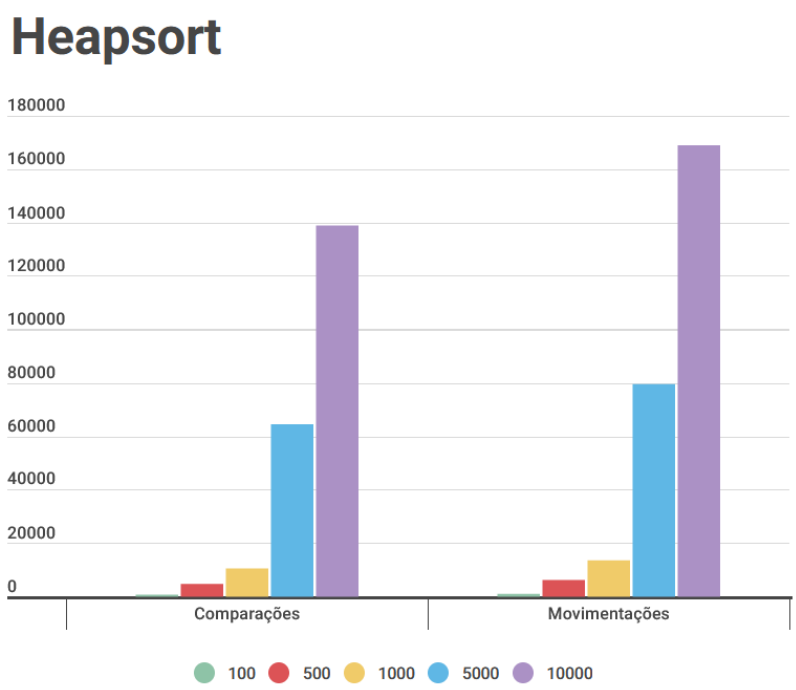
N = 500 => comp = 0,861 |Mov = 1,117

N = 1000 => comp = 0,874 |Mov = 1,004

N = 5000 => comp = 0,925 |Mov = 1,139

N = 10000 => comp = 0,931 |Mov = 1,053

Heapsort:



Ao analisar o gráfico do algoritmo Heapsort, percebe-se que o crescimento em suas comparações é semelhante ao de suas movimentações, assim como o Mergesort, e com a mesma complexidade, $O(N*\log_2 N)$, o que pode ser comprovado analisando o número exato de comparações e movimentações:

Ao dividir o número de comparações por $N*\log_2 N$ deve-se obter um valor constante (com pequenas variações) que se repete pra todos os valores de N, o mesmo ocorre com as movimentações:

N = 100 => comp = 1,115 | Mov = 1,566

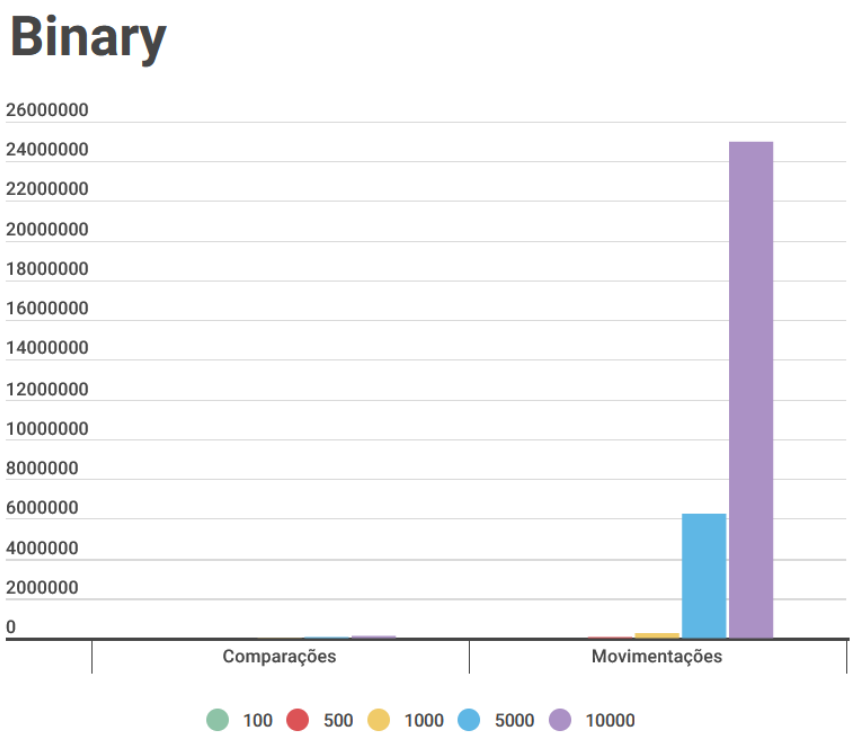
N = 500 => comp = 1,066 |Mov = 1,401

N = 1000 => comp = 1,065 |Mov = 1,366

N = 5000 => comp = 1,050 |Mov = 1,295

N = 10000 => comp = 1,045 |Mov = 1,271

Binary Insertionsort:



Ao analisar o gráfico do algoritmo Binary Insertionsort, percebe-se que o número de suas comparações é nitidamente menor que de o de suas movimentações, assim como o crescimento. Isso se deve porque a complexidade de comparações é $O(N*\log_2 N)$, enquanto a de movimentações é $O(N^2)$, o que pode ser comprovado analisando o número exato de comparações e movimentações:

Ao dividir o número de comparações por $N*\log_2 N$ deve-se obter um valor constante (com pequenas variações) que se repete pra todos os valores de N, o mesmo ocorre com as movimentações, ao dividir os valores por N^2:

N = 100 => comp = 0,812 | Mov = 0,251

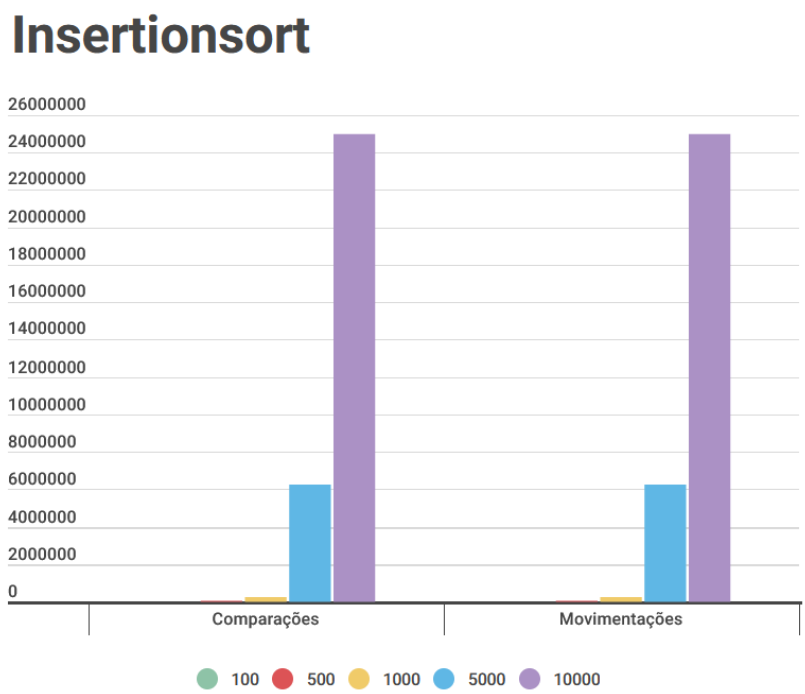
N = 500 => comp = 0,849 |Mov = 0,254

N = 1000 => comp = 0,861 |Mov = 0,256

N = 5000 => comp = 0,887 |Mov = 0,250

N = 10000 => comp = 0,895 |Mov = 0,249

Insertionsort:



Ao analisar o gráfico do algoritmo Insertionsort, percebe-se que o crescimento em suas comparações é igual ao de suas movimentações, logo com a mesma complexidade, $O(N^2)$, o que pode ser comprovado analisando o número exato de comparações e movimentações:

Ao dividir o número de comparações por N^2 deve-se obter um valor constante (com pequenas variações) que se repete pra todos os valores de N, o mesmo ocorre com as movimentações:

N = 100 => comp = 0,251 | Mov = 0,251

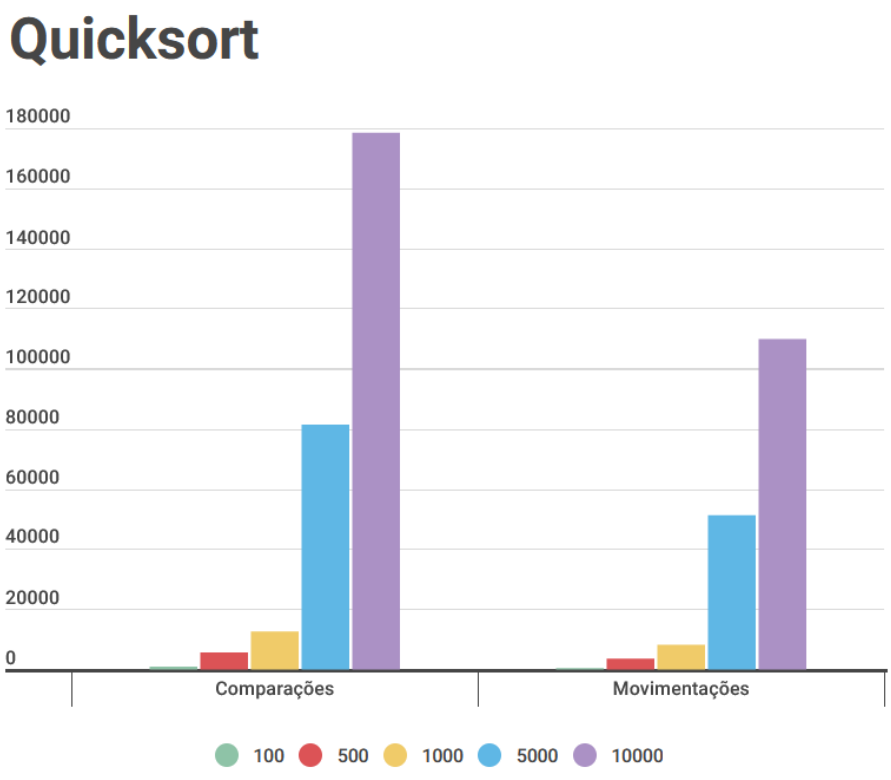
N = 500 => comp = 0,254 |Mov = 0,254

N = 1000 => comp = 0,256 |Mov = 0,256

N = 5000 => comp = 0,250 |Mov = 0,250

N = 10000 => comp = 0,249 |Mov = 0,249

Quicksort:



Ao analisar o gráfico do algoritmo Quicksort, percebe-se que o crescimento em suas comparações é semelhante ao de suas movimentações, assim como o Heapsort, e com a mesma complexidade em caso médio, $O(N*\log_2 N)$, o que pode ser comprovado analisando o número exato de comparações e movimentações:

Ao dividir o número de comparações por $N*\log_2 N$ deve-se obter um valor constante (com pequenas variações) que se repete pra todos os valores de N, o mesmo ocorre com as movimentações:

N = 100 => comp = 1,420 | Mov = 0,767

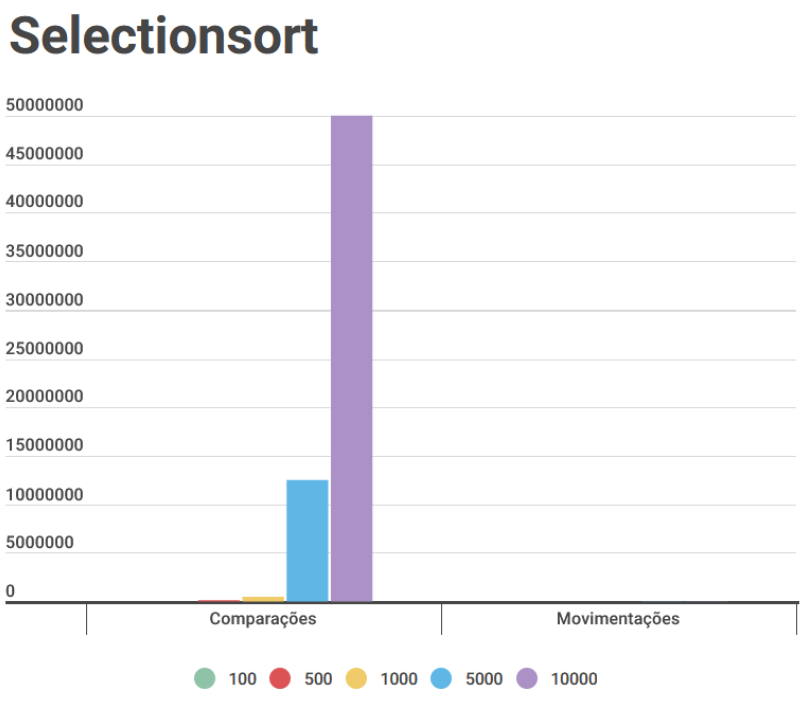
N = 500 => comp = 1,278 |Mov = 0,817

N = 1000 => comp = 1,271 |Mov = 0,824

N = 5000 => comp = 1,325 |Mov = 0,835

N = 10000 => comp = 1,343 |Mov = 0,826

Selectionsort:



Ao analisar o gráfico do algoritmo Selectionsort, percebe-se que o número de suas comparações é nitidamente maior que de o de suas movimentações, assim como o crescimento. Isso se deve porque a complexidade de comparações é $O(N^2)$, enquanto a de movimentações é O(N), o que pode ser comprovado analisando o número exato de comparações e movimentações:

Ao dividir o número de comparações por N^2 deve-se obter um valor constante (com pequenas variações) que se repete pra todos os valores de N, o mesmo ocorre com as movimentações, ao dividir os valores por N:

N = 100 => comp = 0,495 | Mov = 2,970

N = 500 => comp = 0,499 |Mov = 2,994

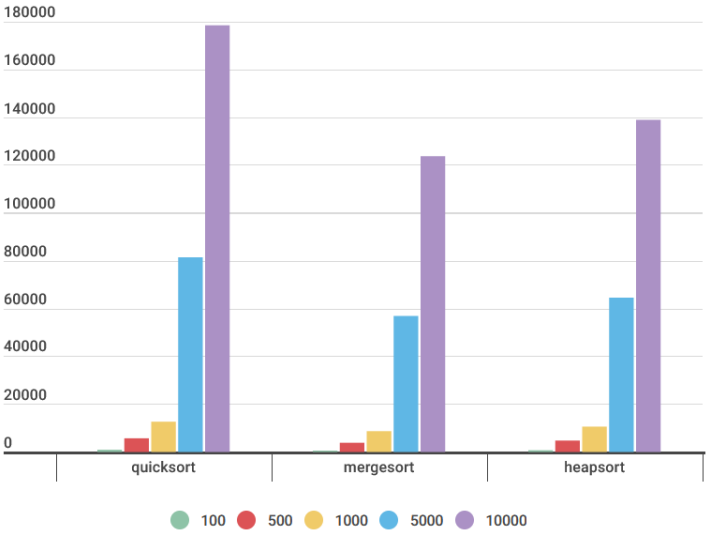
N = 1000 => comp = 0,499 |Mov = 2,997

N = 5000 => comp = 0,499 |Mov = 2,999

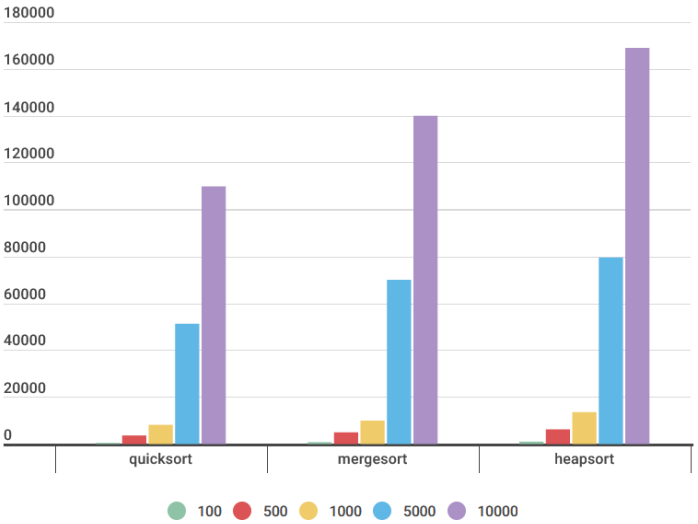
N = 10000 => comp = 0,499 |Mov = 2,999

e) Analisando novamente os gráficos de comparações e movimentações, percebe-se que três algoritmos possuem baixos valores em ambos os gráficos, eles são: Heapsort, Mergesort e Quicksort. Para conseguir analisá-los mais minuciosamente, fizemos um gráfico para analisar suas movimentações, um para analisar suas comparações, e um para analisar as somas das duas:

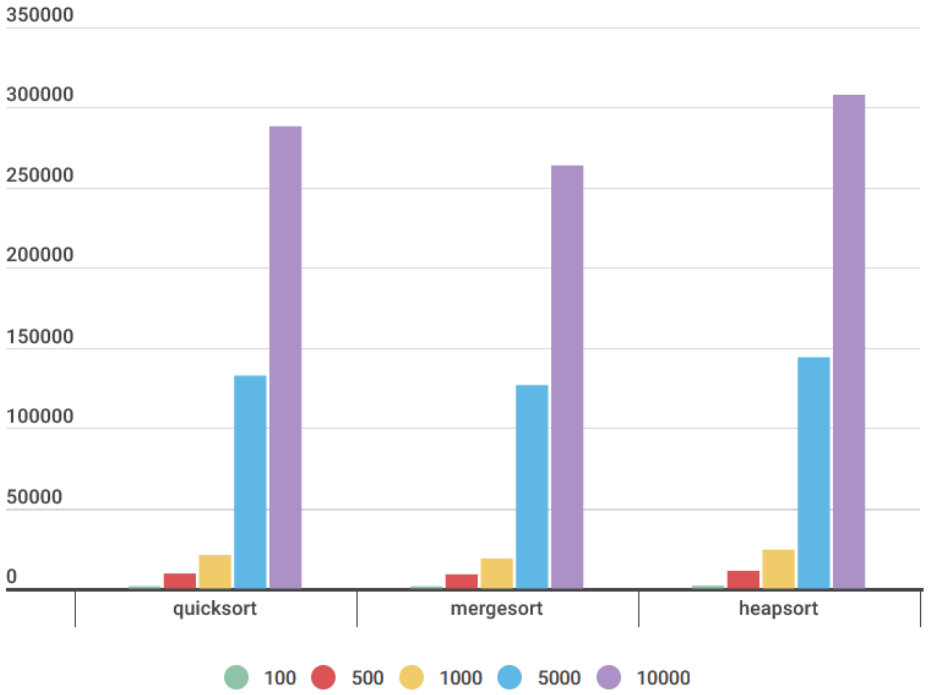
Comparações



Movimentações



Comp + Mov



Portanto, consegue-se analisar que no caso proposto pelo trabalho, devido suas proximidades em números de comparações e movimentações, decorrentes de possuírem a mesma complexidade em caso médio, os Algoritmos mais eficientes são: Quicksort, Mergesort e Heapsort.