Trabalho 1 de Estrutura de Dados II

Professora: Vânia de Oliveira Neves

2018.3

José Santos Sá Carvalho – 201665557AC

Brian Luís Coimbra Maia – 201665512B

Matheus de Oliveira Carvalho – 201665568C

Davi Rezende –

Universidade Federal de Juiz de Fora

Introdução

O objetivo deste trabalho é analisar os gastos dos deputados do Congresso brasileiro nos últimos anos. Para este fim, parte-se de um conjunto de dados de cerca de 3 milhões de registros de gastos. Para encontrar os deputados e partidos responsáveis pelos maiores gastos, serão comparados vários algoritmos de ordenação e de busca, a fim de utilizar o algoritmo que se sair melhor. Serão consideradas três métricas de desempenho dos algoritmos de ordenação: tempo de CPU gasto, número de comparações de chaves, e número de cópias de registros. Para os algoritmos de pesquisa, serão considerados o número de comparações de chaves e o gasto de memória. Todas as funções, rotinas e algoritmos foram implementados usando a linguagem C++11.

O código fonte do projeto pode ser encontrado em `https://github.com/avmhdt/Trabalho_ED2_Final.git`.

1 Análise dos Algoritmos

Nesta seção, foram comparados os desempenhos de diversos algoritmos de ordenação e pesquisa (resolução de colisões). Para tal, foi lido o arquivo `deputies_dataset_tratado.csv`, e os registros foram armazenados em um array `allDeputados` da classe `GastoDeputado`, especificada como segue. Cada objeto `GastoDeputado` corresponde a uma linha da tabela, e contém como atributos:

- `bugged_date`
- `receipt_date` – data do recibo;
- `deputy_id` – id do deputado;
- `political_party` – partido político;
- `state_code` – estado que o deputado representa;
- `deputy_name` – nome do deputado
- `receipt_social_security_number` – número do recibo;
- `receipt_description` – descrição do recibo;
- `establishment_name` – nome do estabelecimento;

- `receipt_value` – valor do recibo.

`deputy_id` e `receipt_value` são armazenados como inteiros (`int`). O resto dos atributos são armazanados como strings (biblioteca `<string>`).

A partir daí, são geradas 5 sementes aleatoriamente, a partir de `srand(1)`. Para cada semente, é aberto um arquivo `saida_seed_XXXX.txt` (para a seed `XXXX`, por exemplo), e a semente é passada como argumento para `srand` para a geração de números aleatórios. Em seguida, para cada valor de N lido do arquivo `entrada.txt` – $N = 1000, 5000, 10000, 50000, 100000, 500000$ –, é alocado um array do tipo `GastoDeputado` e um vetor de inteiros `conjuntoDeputados`, de tamanho N. O primeiro array receberá registros aleatoriamente escolhidos do array `allDeputados`, e o segundo será inicializado com os números `deputy_id` dos mesmos deputados. Desta forma, temos um array de registros, e um de inteiros. Esses arrays serão ordenados conforme o cenário requisitado. Ao final da ordenação por um determinado algoritmo, serão imprimidos no `saida_seed_XXXX.txt` um valor booleano (0/1 corresponente) que diz se o vetor foi ordenado ou não, o tempo de CPU gasto na ordenação em segundos, o número de comparações de chaves, e o número de cópias de registros. Para este fim, uma troca da posição entre dois registros é contabilizada como uma (1) cópia.

Para o cálculo dos números de comparações de chaves, e cópias de registros, em cada função de ordenação há dois argumentos `unsigned int *comp` e `unsigned int *copias`, que são ponteiros para inteiros positivos. Os valores `*comp` e `*copias` são inicializados para 0 no programa principal, e os ponteiros são passados para as funções de ordenação, e incrementados toda vez que há uma comparação entre chaves (`(*comp) += 1`), ou uma cópia de registros (`(*cópias) += 1`).

Funções auxiliares à implementação dos algoritmos foram implementadas conforme necessário. Como exemplos, uma função `swapPtr` para trocar dois elementos de um vetor, uma função `ordenado` que checa se um vetor está ordenado, uma wrapper para a função `rand` que retorna um inteiro (pseudo)aleatório entre dois valores especificados.

Ao término das tarefas requeridas em cada senário especificado, para cada semente, são computadas as médias de cada estatística, e essas são imprimidas no arquivo `saidaFinalCenarioX.txt`, assim como o algoritmo mais rápido dentre os comparados, para cada valor de N.

As computações foram realizadas em um notebook Lenovo Ideapad 710s, com processador Intel Core i7-6700HQ CPU @ 2.60GHz × 8, arquitetura x86_64, 16 GiB de memória RAM, placa de vídeo NVIDIA GeForce GTX 950M, rodando Ubuntu 16.04 64-bit.

1.1 Cenário I: Impacto de diferentes estruturas de dados

Neste cenário, foi avaliado o desempenho do algoritmo Quicksort recursivo ao ordenar:

1. Um vetor de inteiros, correspondentes a ID's de deputados aleatoriamente selecionados do conjunto de dados, como delineado acima;
2. Um vetor de registros de gastos de deputados, isto é, um vetor de estruturas `GastoDeputado`, especificadas acima. O ID dos deputados foi usado como chave.

O algoritmo Quicksort recursivo foi implementado de forma similar à vista em sala de aula. Os detalhes do uso das funções e rotinas seguem abaixo.

```
void quickSort(int *left, int *right, unsigned int *comp, unsigned int *copias)
```

```
void quickSortDeputyId(GastoDeputado *leftDep, GastoDeputado *rightDep,
                       unsigned int *comp, unsigned int *copias)
```

Argumentos:

`*left` ponteiro para o primeiro elemento do vetor a ser ordenado.

`*right` ponteiro para o último elemento do vetor a ser ordenado.

`*comp` ponteiro para contador de comparações de chaves.

`*copias` ponteiro para contador de cópias de registros.

Os algoritmos fazem uso de uma rotina para particionar o vetor, a partir de um pivô, que corresponde ao elemento mais à direita do vetor. A rotina está especificada como segue.

```
int *partitionIt(int *left, int *right, int pivot, unsigned int *comp,
                 unsigned int *copias)
```

```
GastoDeputado *partitionItDeputyId(GastoDeputado *leftDep,
                                   GastoDeputado *rightDep, int pivot,
```

`unsigned int *comp, unsigned int *copias)`

Argumentos:

`*left` ponteiro para o primeiro elemento do vetor a ser ordenado.

`*right` ponteiro para o último elemento do vetor a ser ordenado.

`pivot` pivô a partir do qual se faz o particionamento.

`*comp` ponteiro para contador de comparações de chaves.

`*copias` ponteiro para contador de cópias de registros.

As médias de cada estatística computada, para cada estrutura de dados, e cada valor de N, são exibidas nas tabelas e nos gráficos abaixo. Os eixos y de cada gráfico estão em escala logarítmica para facilitar a compreensão.

Tempo de CPU médio (segundos)

N	Inteiros	Registros de Gastos
1000	0.0001356	0.0048062
5000	0.0007382	0.030969
10000	0.0018682	0.07712
50000	0.0214904	1.04248
100000	0.064765	3.57413
500000	1.26934	83.2762

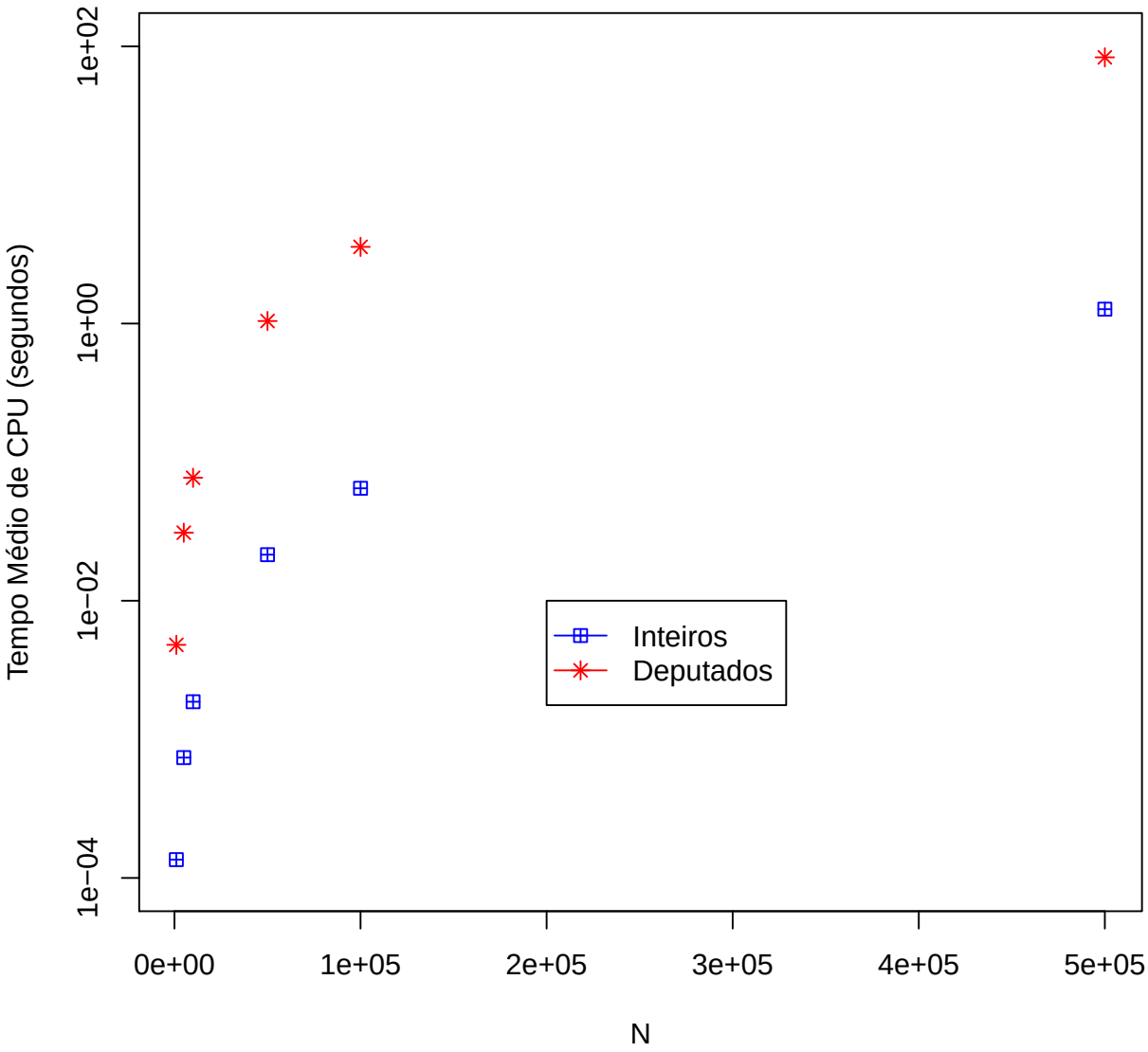
Número médio de comparações de chaves

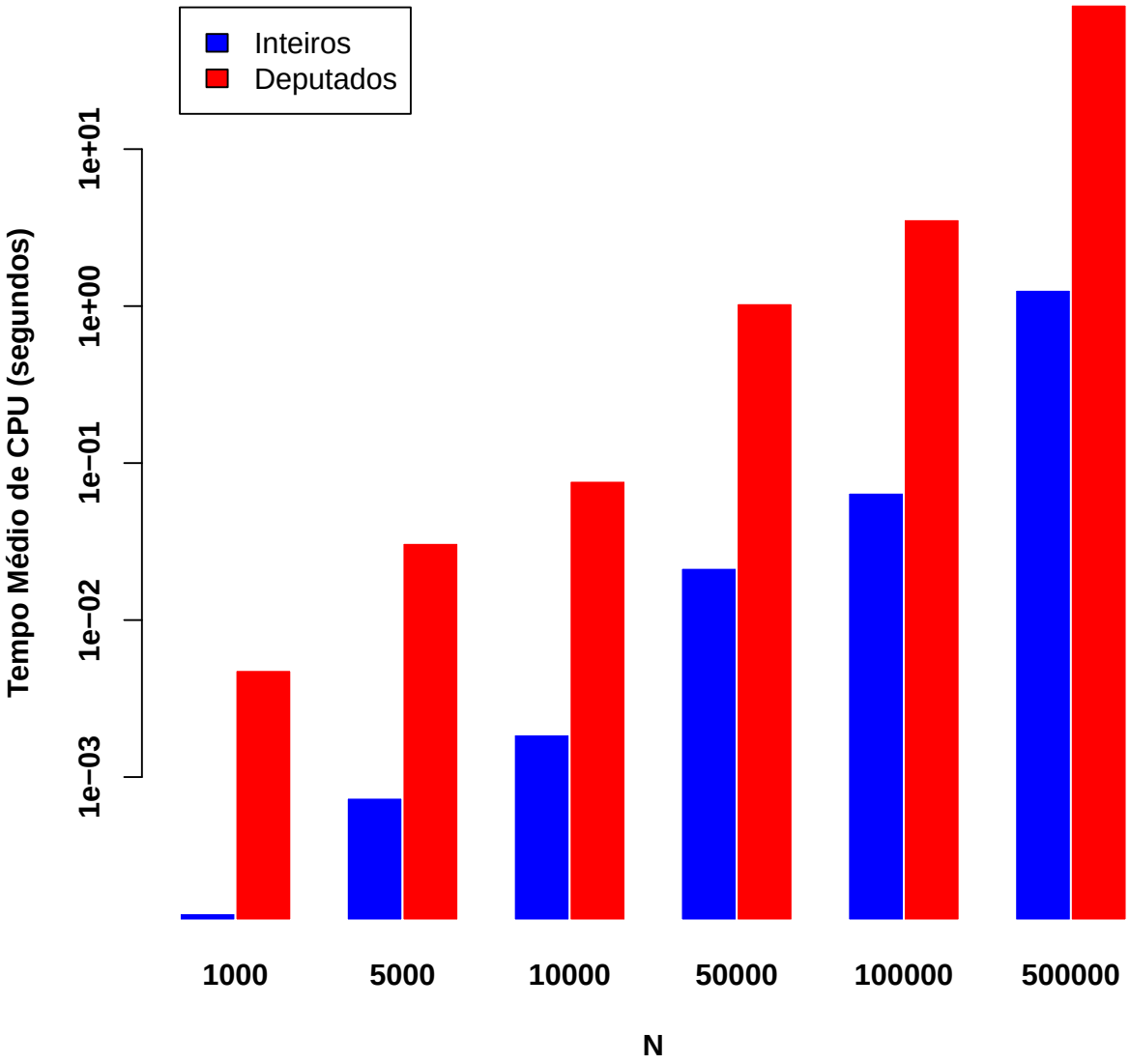
N	Inteiros	Registros de Gastos
1000	11261	11261
5000	79790	79790
10000	197640	197640
50000	2551915	2551915
100000	8899232	8899232
500000	197527310	197527310

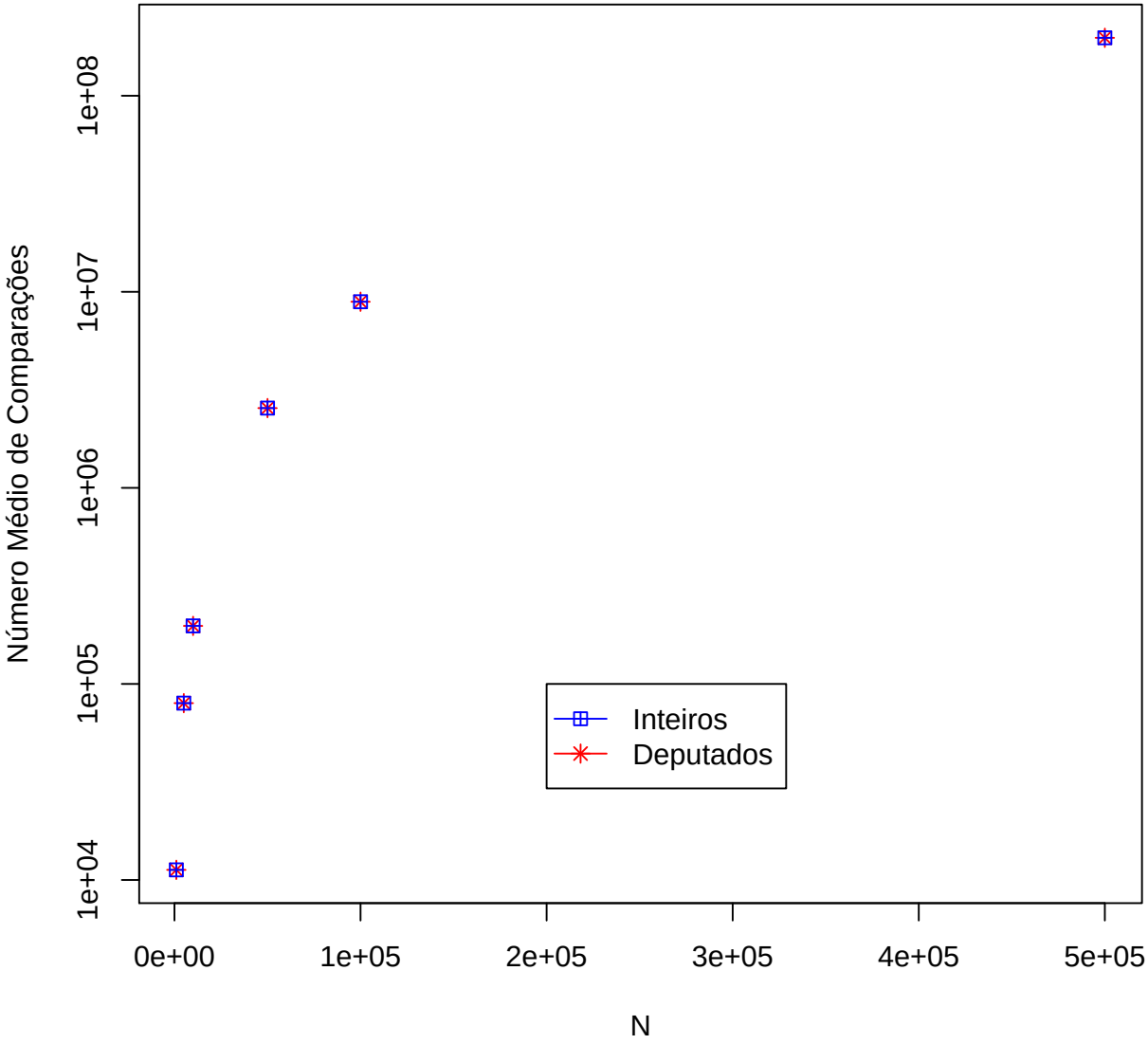
Número médio de cópias de registros

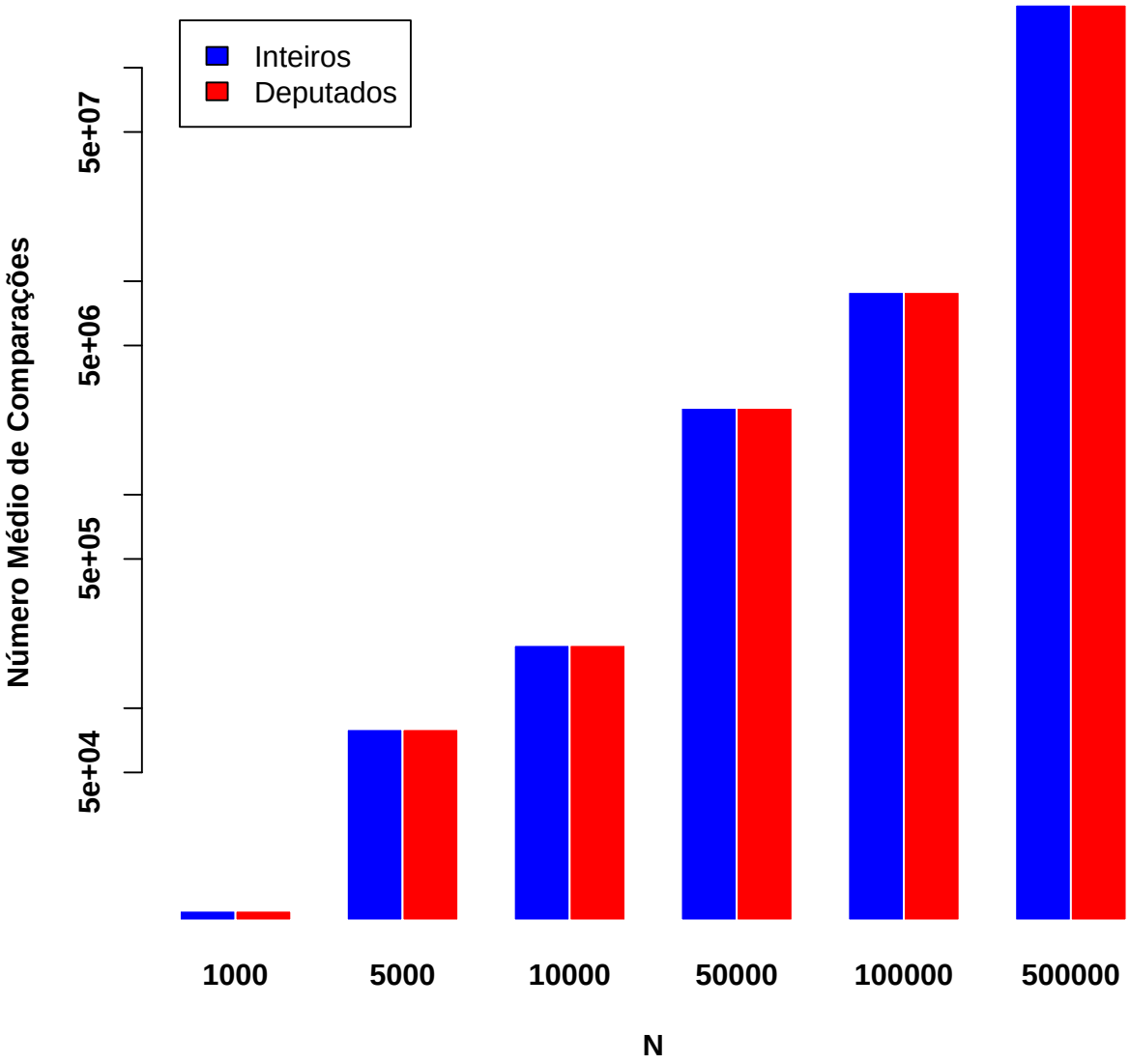
N	Inteiros	Registros de Gastos
1000	6652	6652
5000	52054	52054
10000	147580	147580
50000	2289060	2289060
100000	8360755	8360755
500000	194814933	194814933

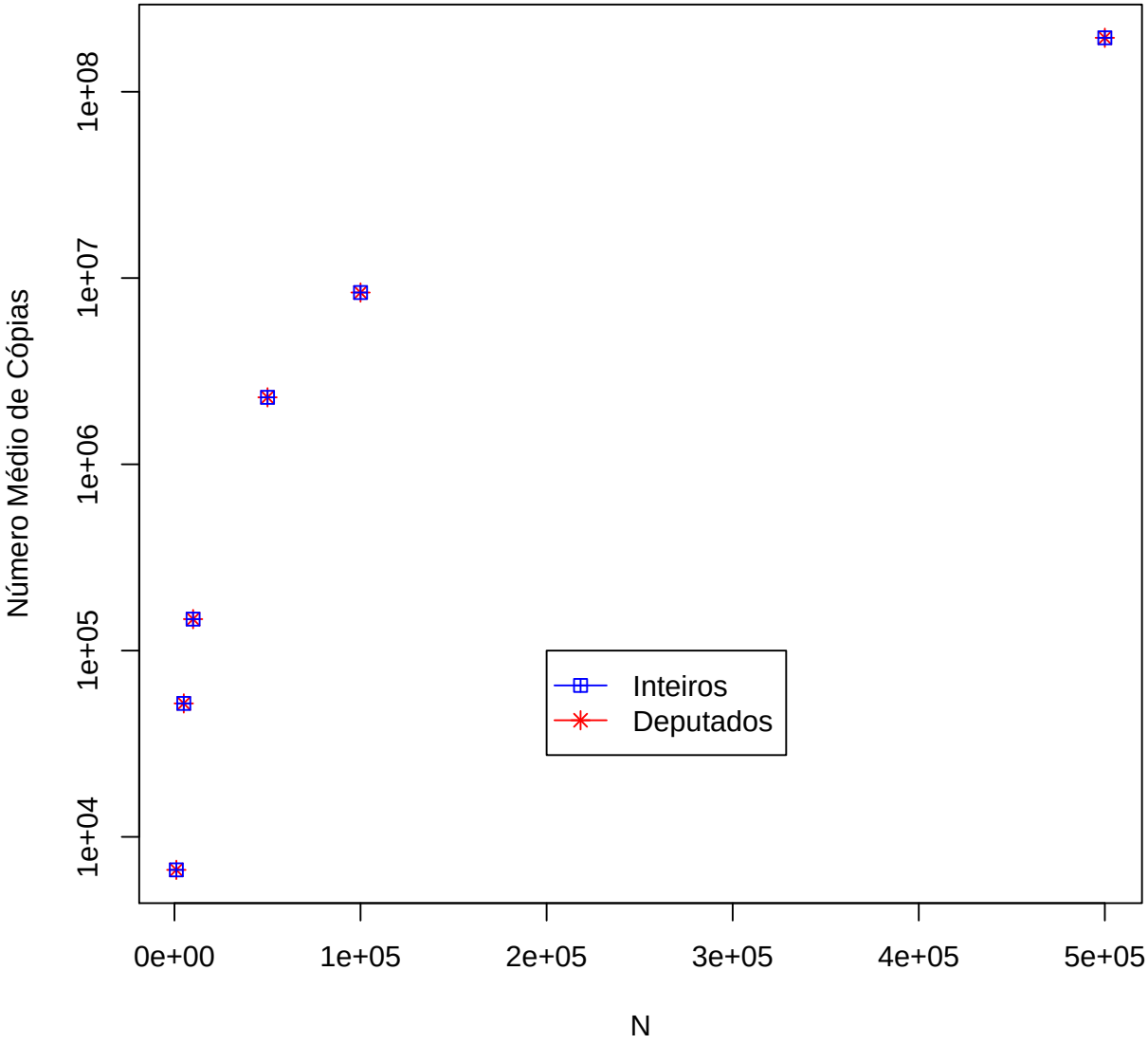
Como é possível verificar, o número de comparações e de cópias é igual para as duas estruturas de dados (afinal, é o mesmo algoritmo). Foi observada uma diferença significativa no tempo, como evidenciado pelos gráficos abaixo.

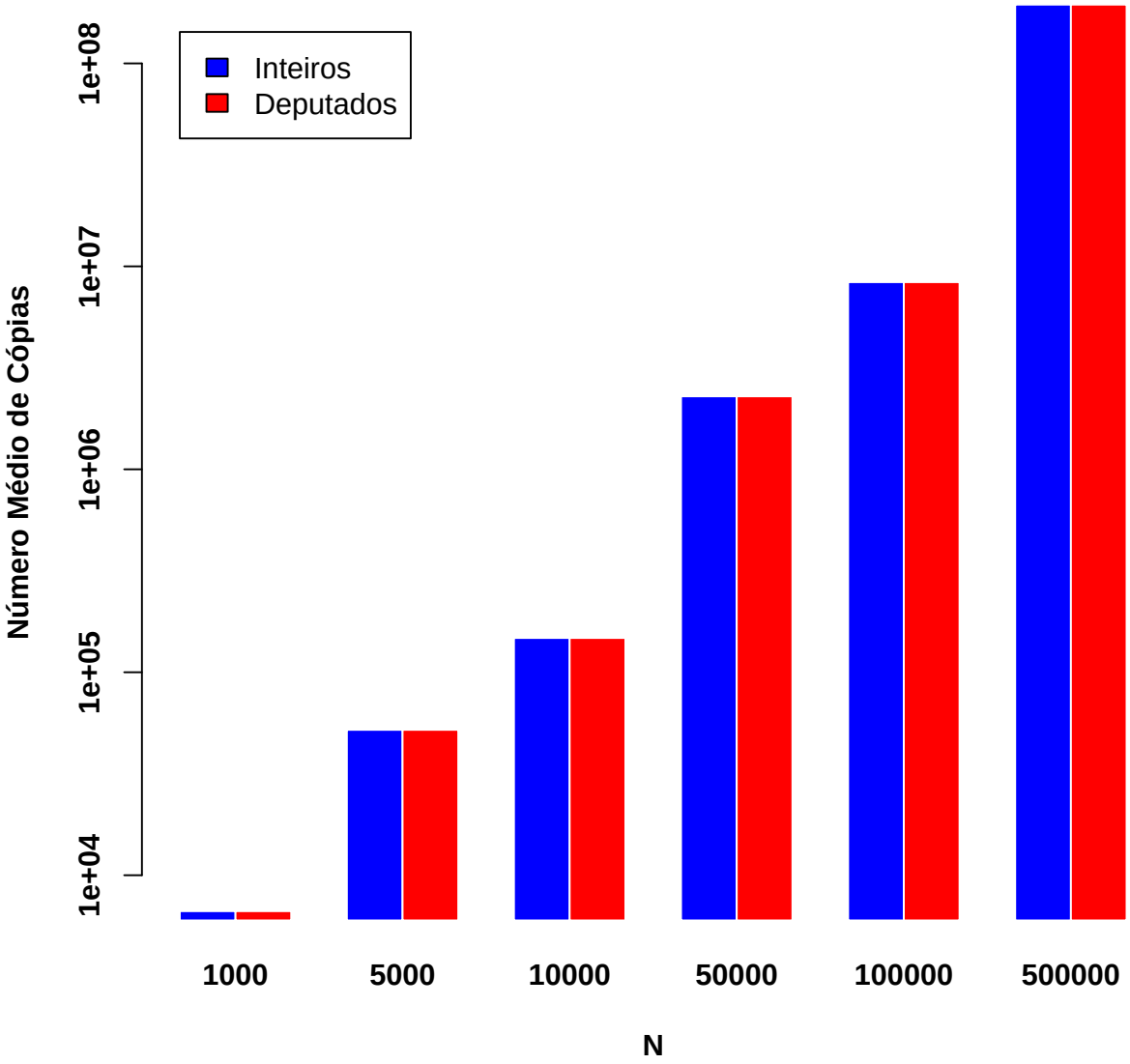

Tempo Médio de CPU (segundos)
1e+02
1e+00
1e−02
1e−04
0e+00
1e+05
2e+05
3e+05
4e+05
5e+05
N
Inteiros
Deputados


Inteiros
Deputados
Tempo Médio de CPU (segundos)
1e−03
1e−02
1e−01
1e+00
1e+01
1000
5000
10000
50000
100000
500000
N


Número Médio de Comparações
1e+04
1e+05
1e+06
1e+07
1e+08
0e+00
1e+05
2e+05
3e+05
4e+05
5e+05
N
Inteiros
Deputados


Inteiros
Deputados
Número Médio de Comparações
5e+04
5e+05
5e+06
5e+07
1000
5000
10000
50000
100000
500000
N


Número Médio de Cópias
1e+04
1e+05
1e+06
1e+07
1e+08
0e+00
1e+05
2e+05
3e+05
4e+05
5e+05
N
Inteiros
Deputados



Como é de se esperar, a ordenação de inteiros foi mais rápida que a ordenação de `GastoDeputado`, para todos os valores de N. Esse é um resultado bastante intuitivo, uma vez que a comparação entre dois aributos de um objeto `GastoDeputado` envolve a chamada de uma função `getDeputyId()`, e enquanto um elemento em um vetor de inteiros já está prontamente disponível para comparação. No entanto, é importante destacar que essa diferença no tempo de CPU gasto não se deve a uma diferença entre o número de operações de comparação ou cópia executadas, uma vez que esse número é idêntico para as duas estruturas de dados.

1.2 Cenário II: Impacto de variações do Quicksort

Neste cenário, foram avaliados os desempenhos de diferentes variações do Quicksort para ordenar um conjunto de N inteiros armazenados em um vetor. Cada elemento do vetor deve consiste em um valor de `deputy_id`, importados aleatoriamente da base de dados, como explicado acima. As variações usadas foram as seguintes:

1. Quicksort (Recursivo): Escolhe o pivô como sendo o elemento mais à direita do vetor.
2. QuickSort Mediana(k): Escolhe o pivô como sendo a mediana de k elementos aleatoriamente escolhidos. Foram testados os valores k = 3 e k = 5.
3. Quicksort Insertion(m): Utiliza o InsertionSort para ordenar partições de tamanho menor ou igual a m. Foram testados os valores m = 10 e m = 100.

A função `int partitionIt` foi usada, conforme feito no Cenário I, para o particionamento do vetor, dado um pivô. O Quicksort Mediana(k) faz uso de uma rotina auxiliar que seleciona aleatoriamente k elementos do vetor, calcula e retorna a mediana entre eles. Para calcular a mediana, os k elementos foram inseridos em um vetor de inteiros, este vetor foi ordenado, e foi retirado o valor na posição intermediária. A ordenação foi feita da seguinte forma: O menor elemento do vetor foi encontrado e colocado na posição mais à esquerda, depois o segundo menor dentre os elementos restantes, e assim por diante. Como isso foi feito somente para vetores pequenos (k = 3 e k = 5), é esperado que a ineficiência dessa estratégia não interfira de forma significativa no tempo levado pela ordenação, em relação a outros algoritmos que poderiam ser usados para ordenar o vetor de tamanho k.

A implementação do Quicksort recursivo foi a mesma delineada na Seção 1.1. Seguem as especificações dos outros algoritmos e rotinas utilizadas.

```
void quickSortMedian(int *left, int *right, int k, unsigned int *comp,
                     unsigned int *copias)
```

```
void quickSortInsertion(int *left, int *right, int m, unsigned int *comp,
                        unsigned int *copias)
```

Argumentos: Os argumentos são os mesmos especificado anteriormente para a rotina `quickSort`, com exceção de

`k` número de elementos a partir dos quais se calcula a mediana.

`m` tamanho a partir do qual se ordena o vetor usando `insertionSort`, conforme o escrito abaixo.

Para calcular a mediana, a rotina usada foi `medianOfK`, que segue especificada abaixo.

```
void medianOfK(int *left, int *right, int k, unsigned int *comp,
               unsigned int *copias)
```

Argumentos:

`*left` ponteiro para o primeiro elemento do vetor a ser ordenado.

`*right` ponteiro para o último elemento do vetor a ser ordenado.

`k` número de elementos a partir dos quais se calcula a mediana.

`*comp` ponteiro para contador de comparações de chaves.

`*copias` ponteiro para contador de cópias de registros.

```
void insertionSort(int *vec, int vecSize, unsigned int *comp,
                   unsigned int *copias)
```

Argumentos:

`*vec` ponteiro para o primeiro elemento do vetor a ser ordenado.

`vecSize` número de elementos do vetor a ser ordenado.

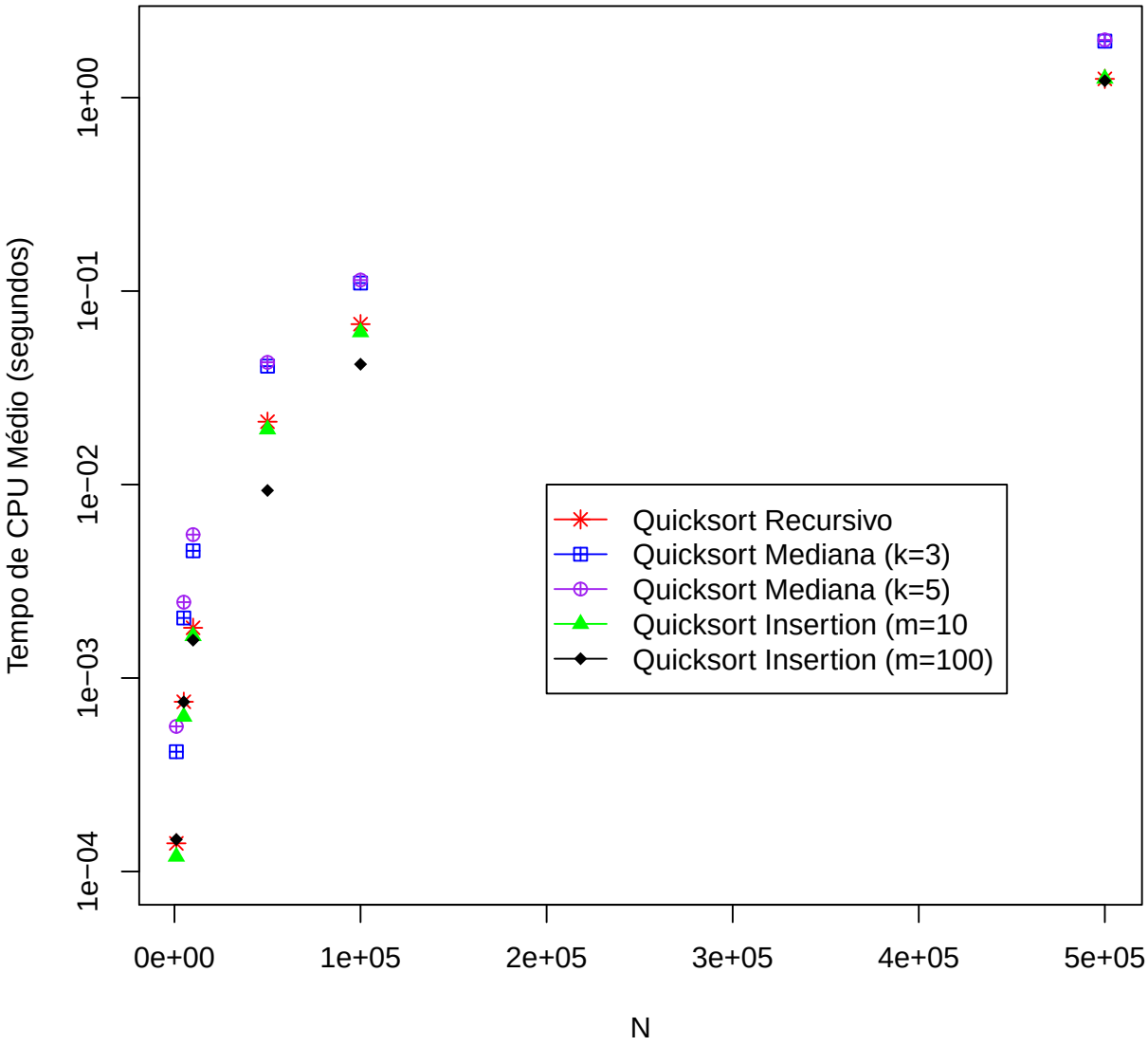
`*comp` ponteiro para contador de comparações de chaves.

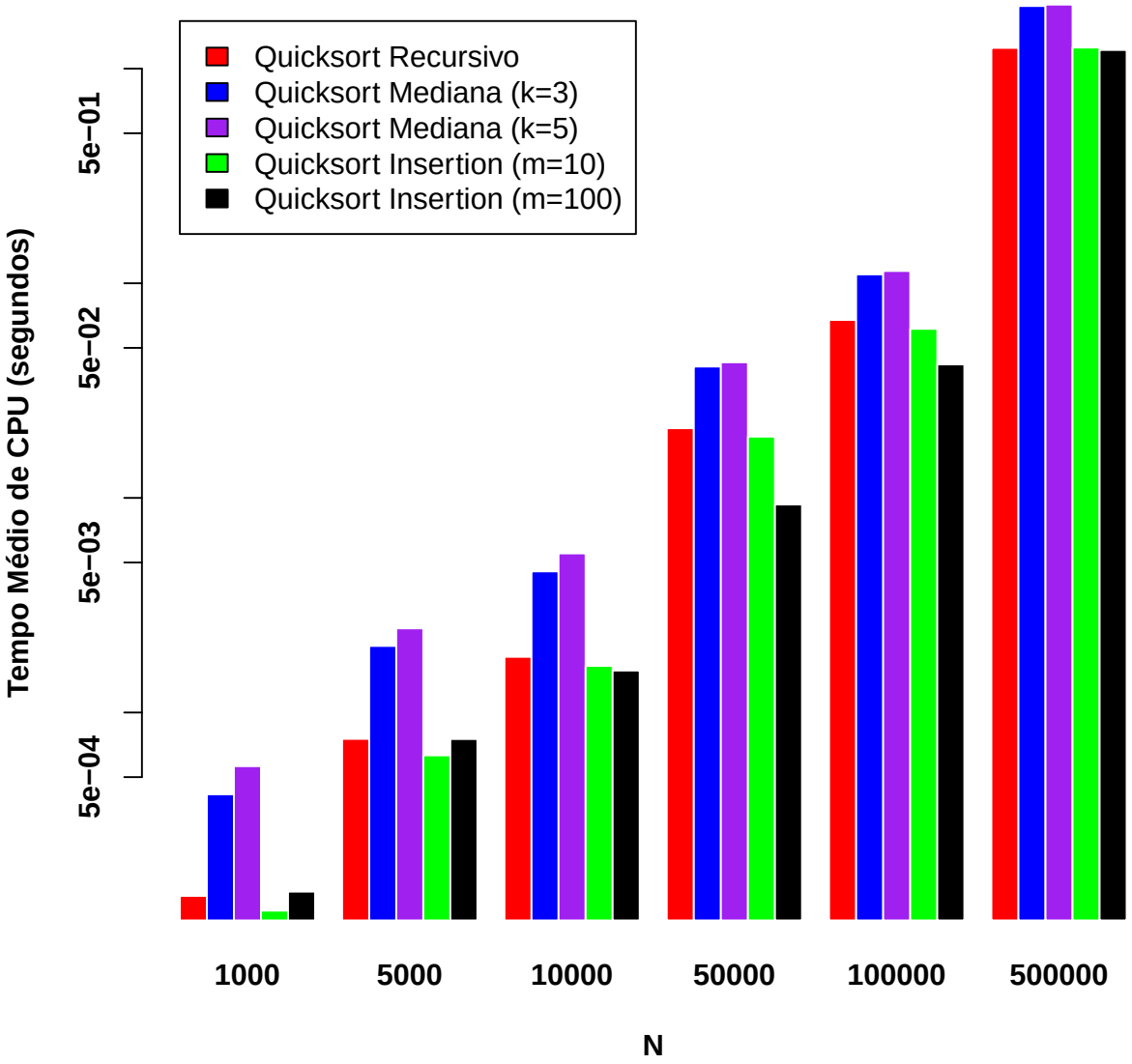
`*copias` ponteiro para contador de cópias de registros.

Os resultados da ordenação usando os três algoritmos, compostos pelas métricas delineadas anteriormente (tempo de CPU gasto, número de comparações de chaves, e número de cópias de registros), são exibidos nas tabelas e gráficos abaixo.

Tempo de CPU médio (segundos)

N	Recursivo	Mediana (k = 3)	Mediana (k = 5)	Insertion (m = 10)	Insertion (m = 100)
1000	0.0001398	0.000416	0.0005624	0.0001194	0.0001464
5000	0.0007538	0.0020432	0.0024672	0.0006316	0.0007524
10000	0.0018178	0.004545	0.0055122	0.0016488	0.001569
50000	0.0211194	0.040929	0.0428264	0.0192932	0.0093214
100000	0.0674578	0.110016	0.114062	0.0613296	0.0418638
500000	1.24946	1.95785	1.98743	1.25393	1.22184



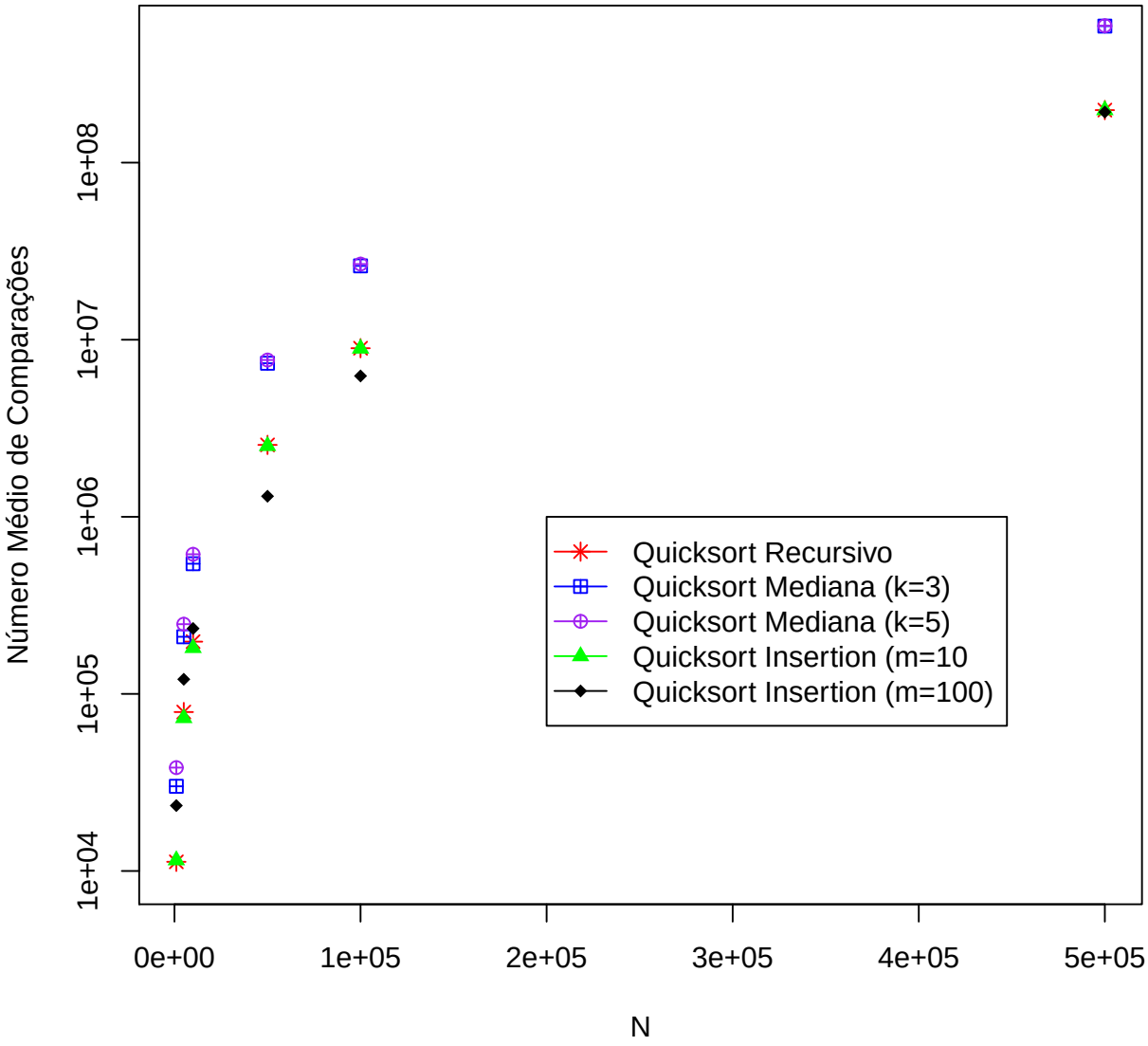


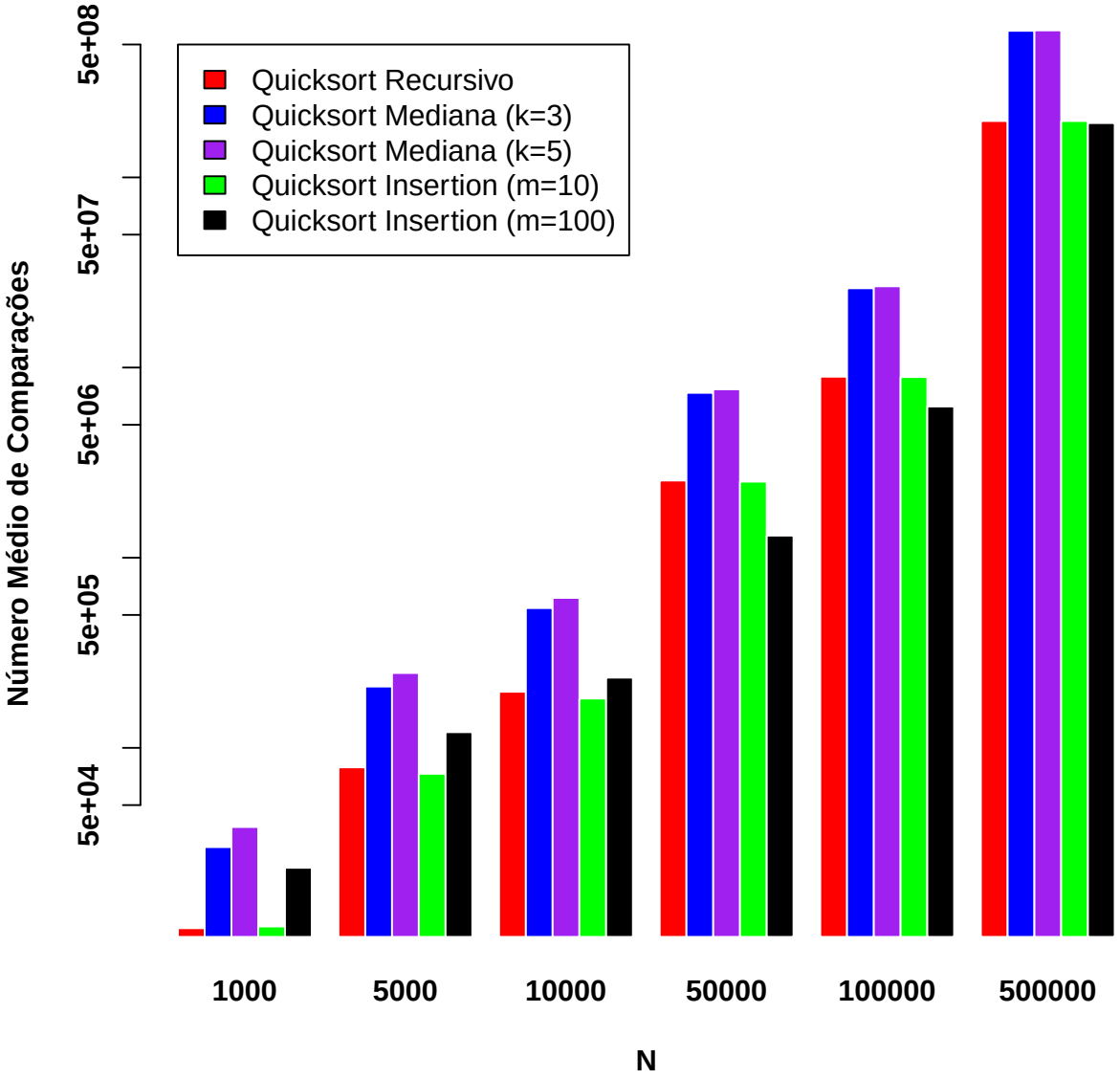
Observando os gráficos, pode-se verificar que, para todos os valores de N, alguma variação do Quicksort Insertion foi o algoritmo mais rápido. Para valores pequenos de N, o algoritmo com m = 10 foi o mais rápido. É interessante em particular o resultado do teste com N = 1000, no qual o Quicksort Recursivo foi ligeiramente mais rápido do que o Quicksort Insertion (m=100). No entanto, para valores maiores de N (a partir de 10000), o parâmetro m = 100 para o Quicksort Insertion levou a uma ordenação mais rápida do que os seus competidores. O Quicksort Mediana desempenhou uma ordenação mais lenta do que os outros algoritmos, com

k = 5 sendo um pouco mais lento do que k = 3 para todos os valores de N.

Número médio de comparações de chaves

N	Recursivo	Mediana (k = 3)	Mediana (k = 5)	Insertion (m = 10)	Insertion (m = 100)
1000	11261	26412	26812	11498	23405
5000	78997	193766	194593	73087	120946
10000	197264	511912	511331	181764	234127
50000	2547158	7213632	7178297	2512564	1306843
100000	8954647	25818958	25743366	8917656	6241590
500000	198011055	588659385	588035471	197971795	193055236



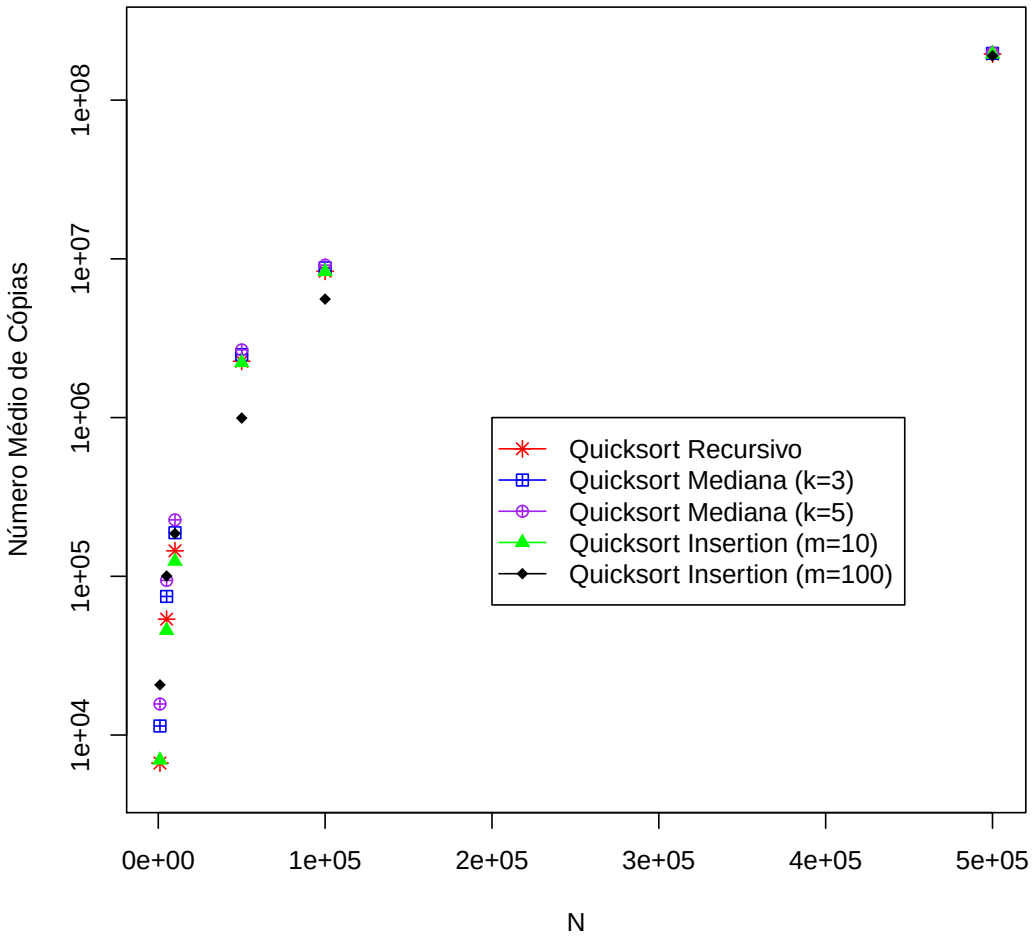


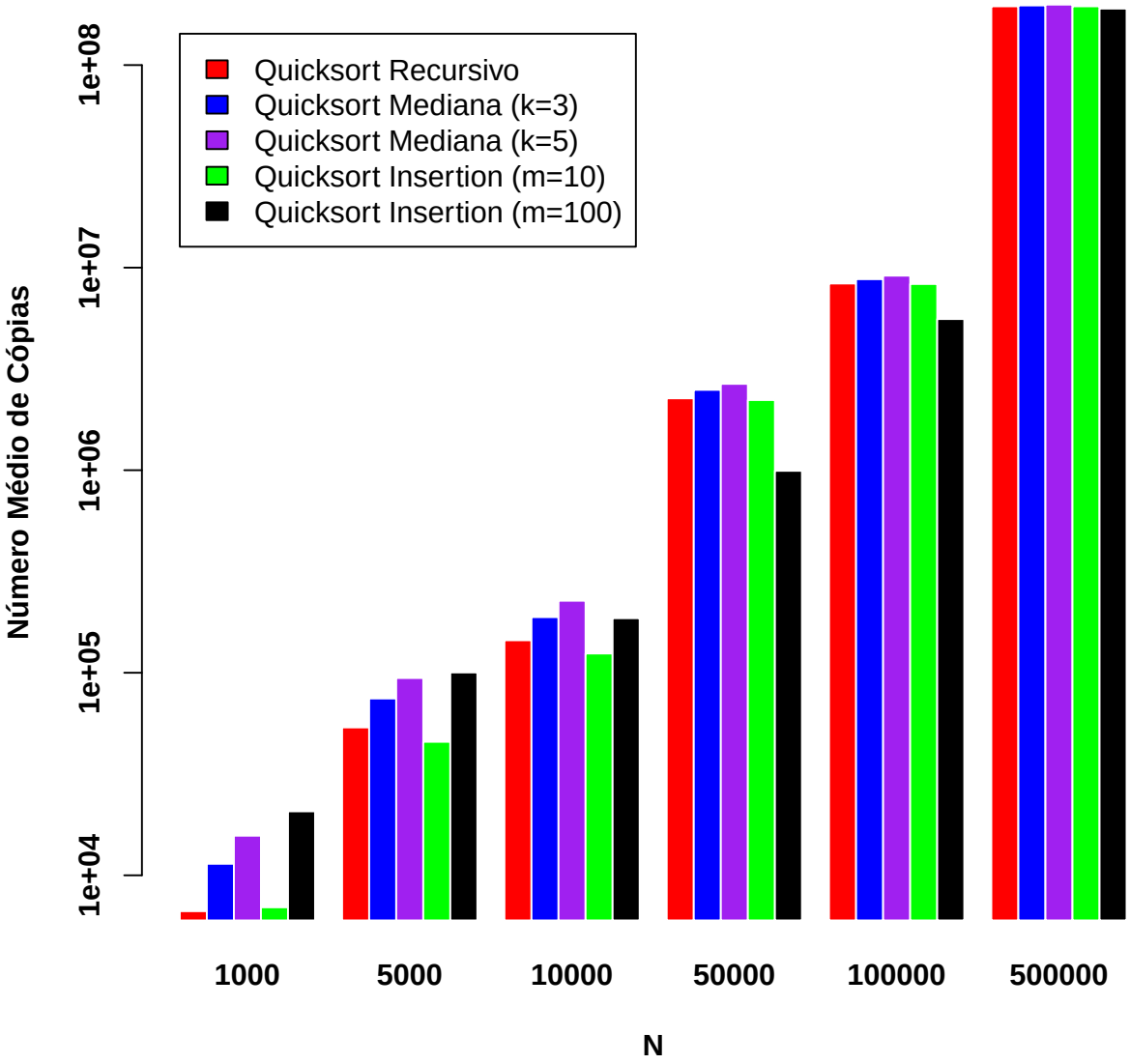
Uma tendência similar é observada no que diz respeito à média de comparações entre chaves. Para os menores valores de N testados, o Quicksort Insertion m = 100 executou menos comparações do que o mesmo algoritmo com m = 10, e o Quicksort Recursivo. Em particular, para N = 1000, o Quicksort Recursivo fez o menor número de comparações. Para valores maiores de N, o Quicksort Insertion m = 100 foi o algoritmo superior nesse quesito, seguido pela variação

m = 10. Os algoritmos que mais fizeram comparações foram as duas variações do Quicksort Mediana (k = 3 e k = 5), o que pode ajudar a explicar o maior tempo de CPU médio gasto por esses algoritmos. Para valores de N até 10000, a Quicksort Mediana k = 5 fez um número maior de comparações do que sua variação k = 3. Para valores maiores de N, não houve uma diferença grande.

Número médio de cópias de registros

N	Recursivo	Mediana (k = 3)	Mediana (k = 5)	Insertion (m = 10)	Insertion (m = 100)
1000	6652	4074	4151	6952	20691
5000	53711	41883	41161	45624	100355
10000	144707	125219	123449	124460	185925
50000	2264008	2189174	2165420	2220314	992989
100000	8352839	8167388	8126653	8306217	5576320
500000	195086102	193974989	193863305	195036682	190027220





Novamente, é possível observar uma tendência similar no número médio de cópias de registros realizadas por cada um dos algoritmos. Para valores de N a partir de 50000, o Quicksort Insertion m = 100 faz menos cópias do que seus concorrentes; para valores menores de N = 5000 ou 10000, o mesmo pode ser dito sobre esse algoritmo com parâmetro m = 10. Para N = 1000, o Quicksort Recursivo é o algoritmo que realiza o menor número de cópias. Em geral,

o parâmetro k = 3 resultou em um número menor de cópias para o Quicksort Mediana (k) do que k = 5, porém, para N = 500000, a diferença é mínima.

A partir desses resultados, verifica-se que o Quicksort Insertion (m = 100) foi o algoritmo com melhor desempenho médio para valores grandes de N (em especial, dando importância ao quesito tempo de CPU). Por isso, foi a variação do Quicksort escolhida para a realização dos testes do próximo cenário.

1.3 Cenário III: Quicksort X InsertionSort X Mergesort X Heapsort X CountingSort

Neste cenário, foram comparados, para a tarefa de ordenação de um conjunto de N inteiros, os algoritmos QuicksortInsertion(m = 100), InsertionSort, Mergesort, Heapsort e CountingSort. O QuicksortInsertion com m = 100 foi a melhor variação do Quicksort, como mostrado pelos resultados do Cenário II. Os algoritmos Mergesort e Heapsort foram implementados conforme o visto em sala de aula. O vetor de N inteiros a ser ordenado corresponde a valores de `deputy_id`, importados aleatoriamente da base de dados, como visto nas subseções anteriores.

Para fins de comparação, foi testado também o desempenho do algoritmo Counting Sort[1]. Esse algoritmo tem uma série de propriedades que o tornam interessante em relação aos outros algoritmos aqui testados. Primeiramente, não é um algoritmo baseado em comparações. Deste modo, o limite inferior de complexidade $\Omega(n \log n)$ não é aplicável. Ao invés de comparar chaves, o algoritmo usa as chaves como índices em um vetor de contadores. Cada elemento desse vetor, em uma dada posição i, corresponde a quantas chaves de valor i existem no vetor de entrada (a ser ordenado). O possui complexidade linear[2] $O(n + k)$, em que n é o tamanho do vetor, e k é a diferença entre a chave de maior valor e a de menor valor. Por isso, é um algoritmo eficiente quando essa diferença não é grande[3]. Segue abaixo o pseudocódigo, e o detalhamento das rotinas utilizadas nesta seção. Os algoritmos QuicksortInsertion(m = 10) e InsertionSort são os mesmos usados no Cenário II.

[1] `https://www.geeksforgeeks.org/counting-sort/`
[2] `https://www.codingeek.com/algorithms/counting-sort-explanation-pseudocode-and-implementation/`
[3] `https://www.codingeek.com/algorithms/counting-sort-explanation-pseudocode-and-implementation/`

```
1: procedure COUNTINGSORT(V[N])
2:     range ← max(V) − min(V) + 1
3:     countVec ← int[range]
4:     output ← int[N]
5:     for 0 ≤ i < N do        //Montando o vetor de contadores
6:         + + countVec[V[i] − min(V)]
7:     for 1 ≤ i < range do    //Adicionando os contadores dos elementos maiores ou iguais
8:         countVec[i]  + = countVec[i − 1]
9:     for N − 1 ≥ i ≥ 0 do        //Montando o vetor output, na ordem correta
10:        output[countVec[V[i] − min(V)] − 1] ← V[i];
11:        − − countVec[V[i] − min(V)];
12:    for 0 ≤ i < N do
13:        V[i] ← output[i]        //Transferindo output para o vetor de entrada
14:    return
```

```
void mergeSort(int *vec, int l, int r, unsigned int *comp, unsigned int *copias)
```

Argumentos:

`*vec` ponteiro para o primeiro elemento do vetor a ser ordenado.

`l` índice do elemento mais à esquerda a partir do qual a ordenação é realizada.

`r` índice do elemento mais à direita até o qual a ordenação é realizada.

`*comp` ponteiro para contador de comparações de chaves.

`*copias` ponteiro para contador de cópias de registros.

O algoritmo faz uso da rotina `mergeVec` para a intercalação dos subvetores, como segue.

```
void mergeVec(int *vec, int start, int mid, int finish, unsigned int *comp,
              unsigned int *copias)
```

Argumentos:

`*vec` ponteiro para o primeiro elemento do vetor a ser ordenado.

`start` índice do elemento mais à esquerda a partir do qual a intercalação é realizada.

`mid` índice do elemento do meio.

`finish` índice do elemento mais à direita, até o qual a intercalação é realizada.

`*comp` ponteiro para contador de comparações de chaves.

`*copias` ponteiro para contador de cópias de registros.

```
void heapSort(int vet[], int tamanho, unsigned int *comp, unsigned int *copias
```

Argumentos:

`vet[]` vetor a ser ordenado.

`tamanho` número de elementos do vetor a ser ordenado.

`*comp` ponteiro para contador de comparações de chaves.

`*copias` ponteiro para contador de cópias de registros.

A heap é armazenada como um vetor, como visto em sala de aula. Para construir a max heap, se faz uso da seguinte rotina:

```
void HeapMax(int vet[], int filho, unsigned int *comp, unsigned int *copias)
```

Argumentos:

`vet[]` vetor a ser ordenado.

`filho` índice do nó filho na heap.

`*comp` ponteiro para contador de comparações de chaves.

`*copias` ponteiro para contador de cópias de registros.

```
void countingSort(int *vec, int vecSize, unsigned int *comp,
                  unsigned int *copias)
```

Argumentos:

`*vec` ponteiro para o primeiro elemento do vetor a ser ordenado.

`vecSize` número de elementos do vetor a ser ordenado.

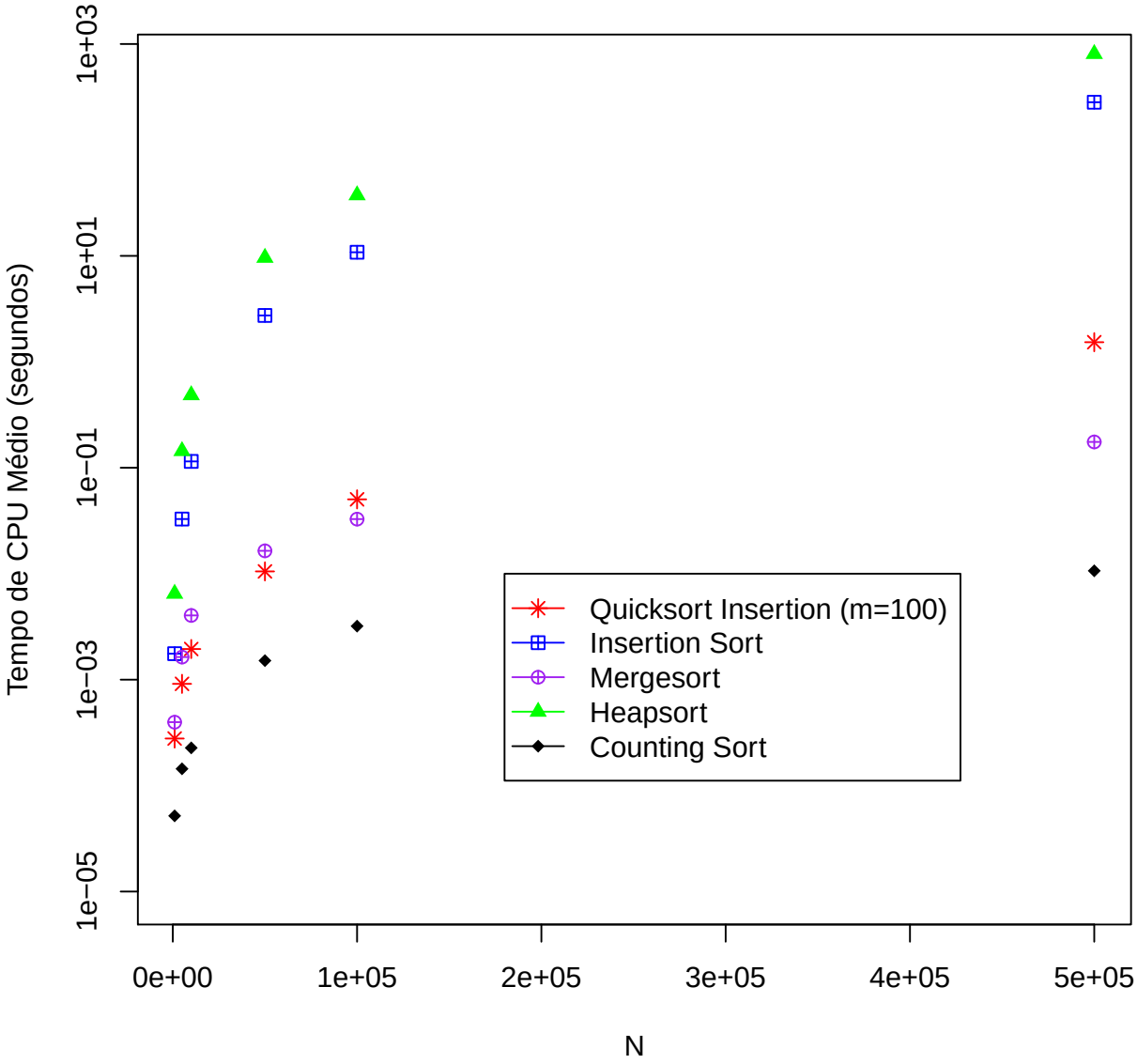
`*comp` ponteiro para contador de comparações de chaves.

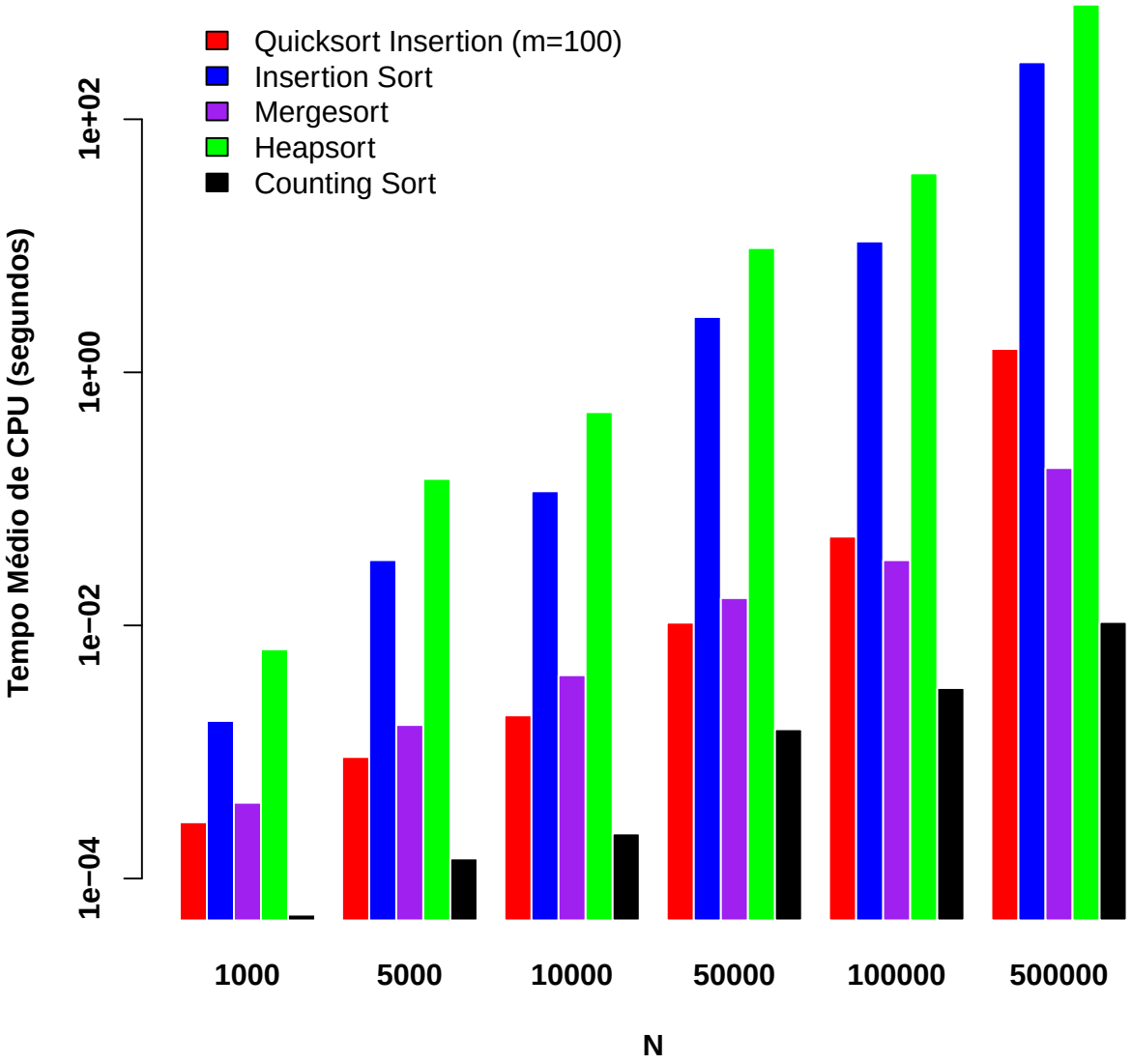
`*copias` ponteiro para contador de cópias de registros.

Os resultados são exibidos nas tabelas e gráficos abaixo.

Tempo de CPU médio (segundos)

N	QuicksortInsertion (m = 100)	InsertionSort	Mergesort	Heapsort	CountingSort
1000	0.0002782	0.001759	0.000397	0.0064756	5.18e-05
5000	0.0009128	0.0328126	0.0016342	0.143827	0.000144
10000	0.0019464	0.114847	0.0040398	0.484715	0.000227
50000	0.010515	2.7394	0.016448	9.60385	0.001512
100000	0.0503384	10.8174	0.0327732	37.3693	0.0032012
500000	1.53142	282.069	0.175349	805.575	0.0106524

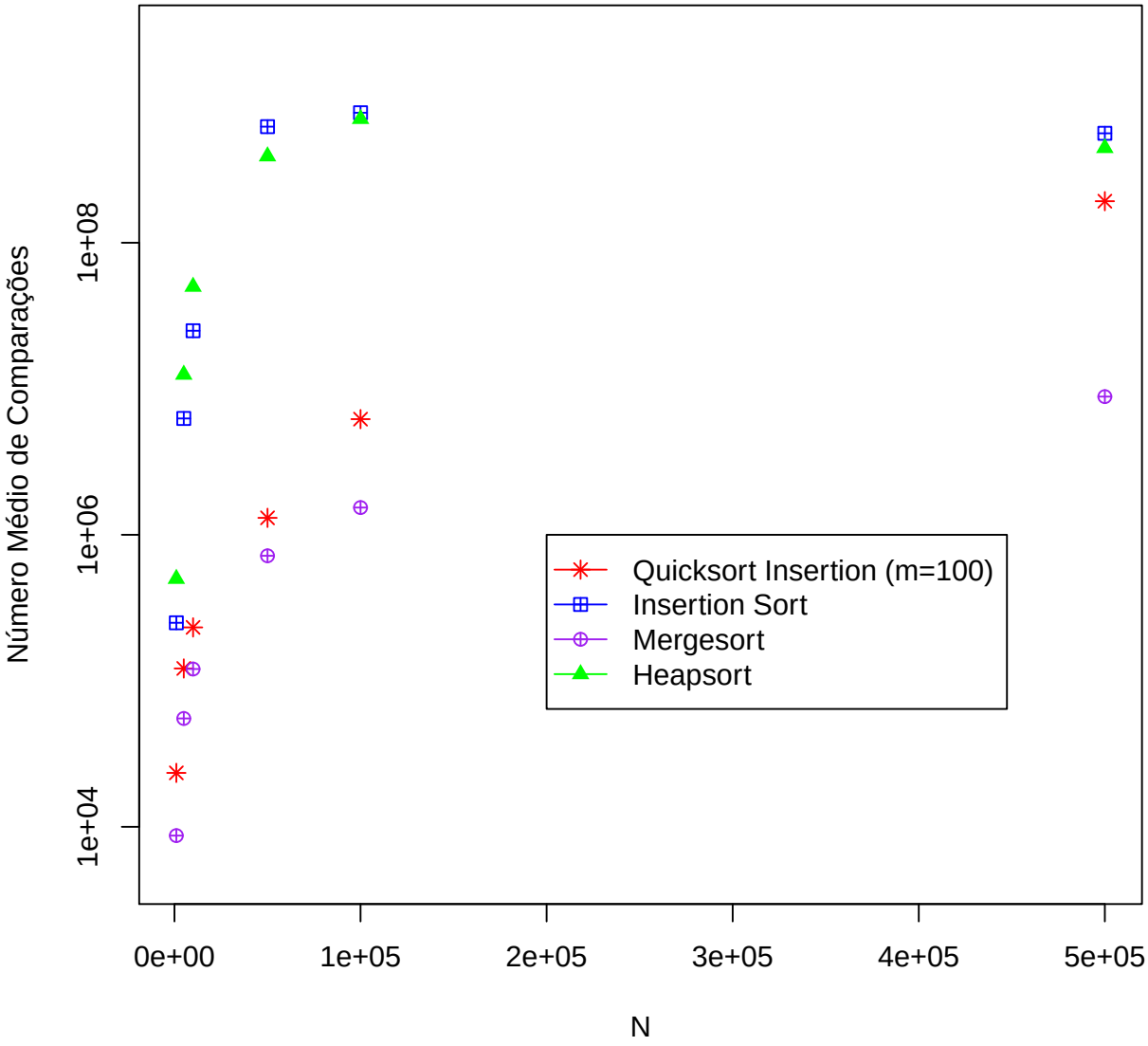

Tempo de CPU Médio (segundos)
1e+03
1e+01
1e−01
1e−03
1e−05
0e+00
1e+05
2e+05
3e+05
4e+05
5e+05
N
Quicksort Insertion (m=100)
Insertion Sort
Mergesort
Heapsort
Counting Sort

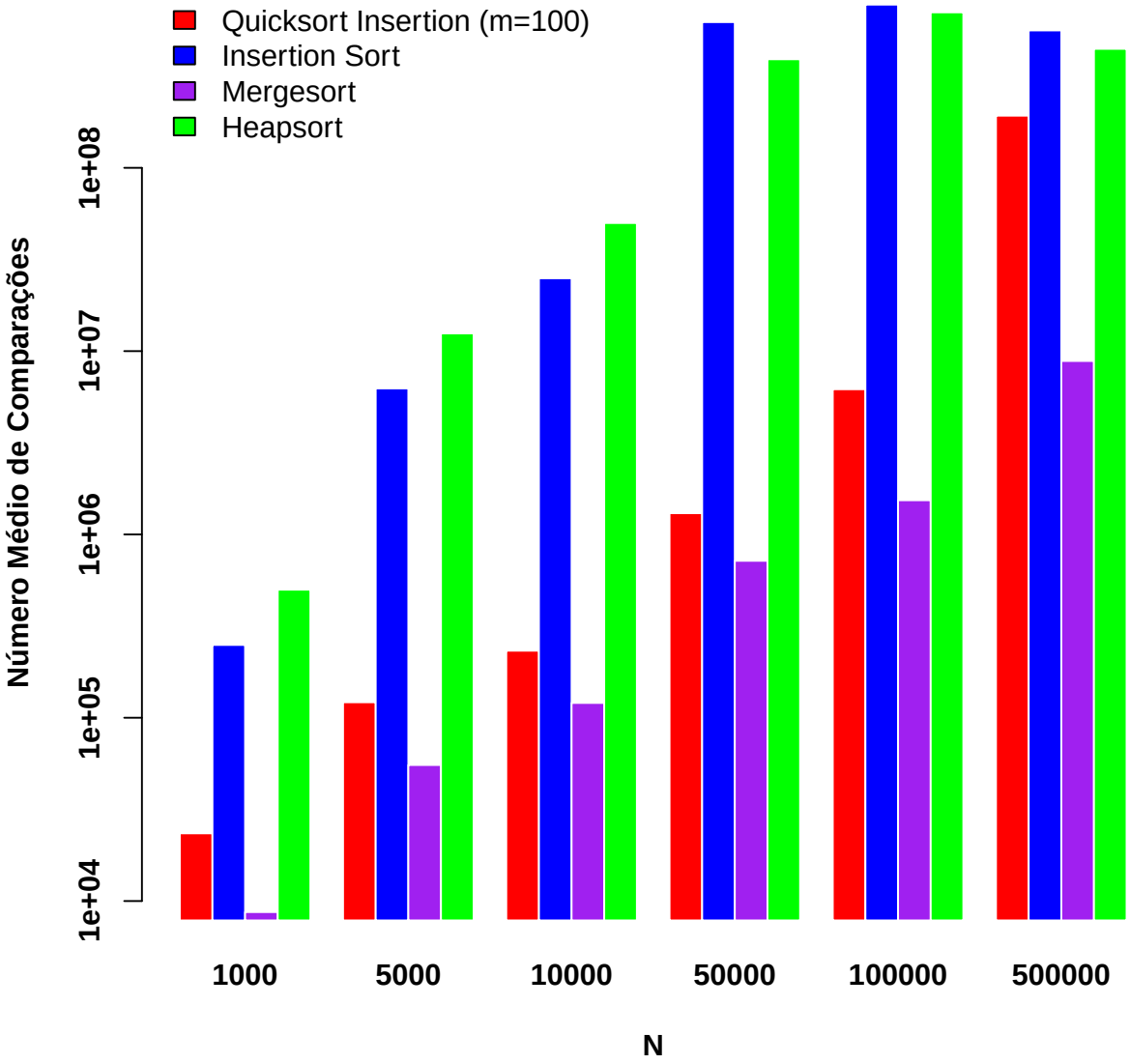


A superioridade do Counting Sort nos testes é deixada bem clara pelos resultados. Como a diferença entre o maior e o menor valor dos conjuntos ordenados não foi significativamente maior do que o número de elementos a serem ordenados (N), o Counting Sort foi o algoritmo mais rápido para todos os valores de N. Por outro lado, Heapsort foi o algoritmo mais lento, seguido do Insertion Sort, para entradas de todos os tamanhos. É interessante observar que, para valores grandes de N (100000 e 500000), o Quicksort Insertion (m = 100) foi mais rápido, em média, do que o Mergesort, enquanto o inverso foi observado para conjuntos menores.

Número médio de comparações de chaves

N	QuicksortInsertion (m = 100)	InsertionSort	Mergesort	Heapsort	CountingSort
1000	23405	249573	8707	499500	0
5000	121446	6260792	55224	12497500	0
10000	232145	24969571	120445	49995000	0
50000	1305734	623790327	718213	390981540	0
100000	6195081	776874100	1536325	704982704	0
500000	192565549	561752736	8837163	445698416	0



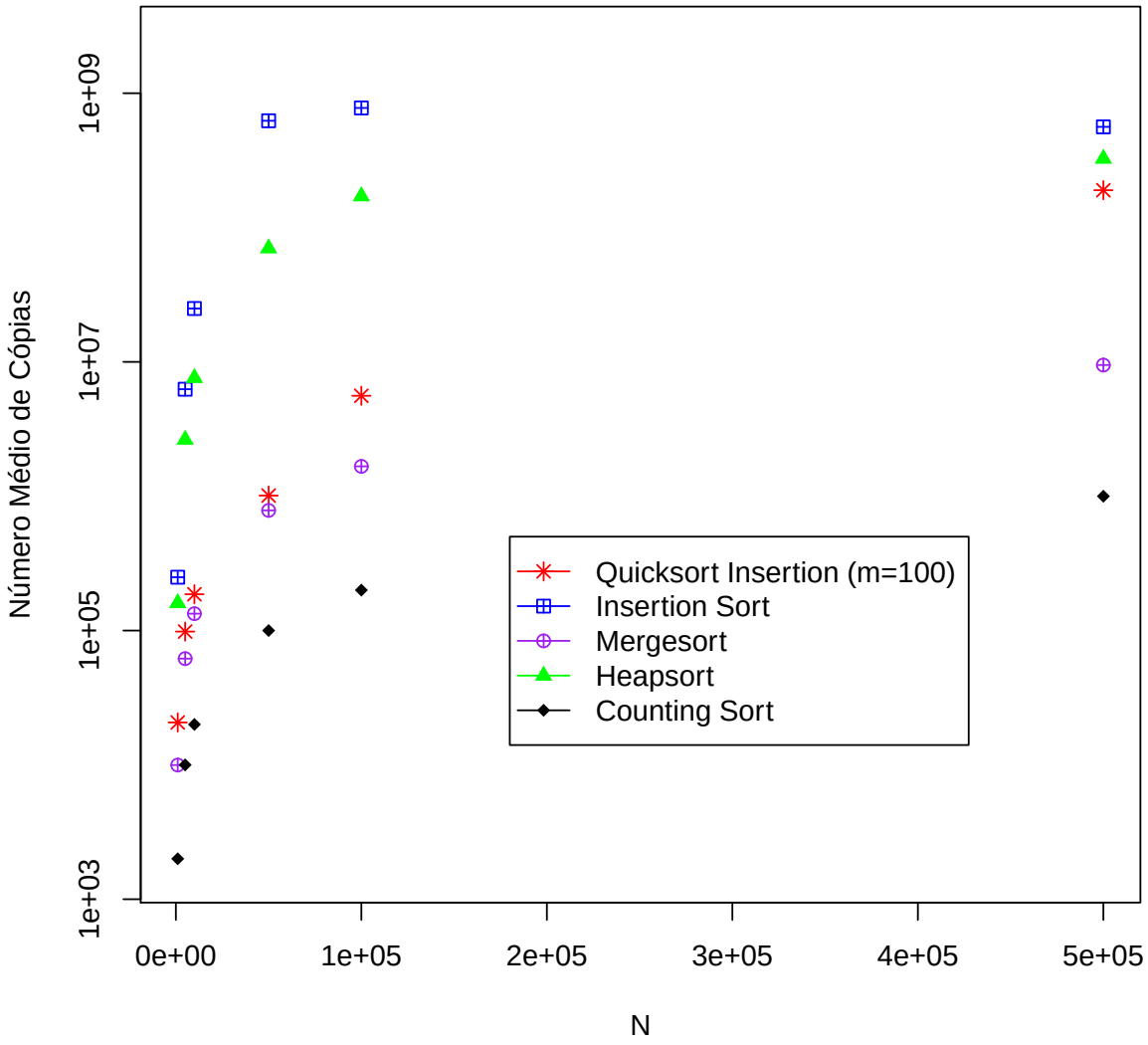


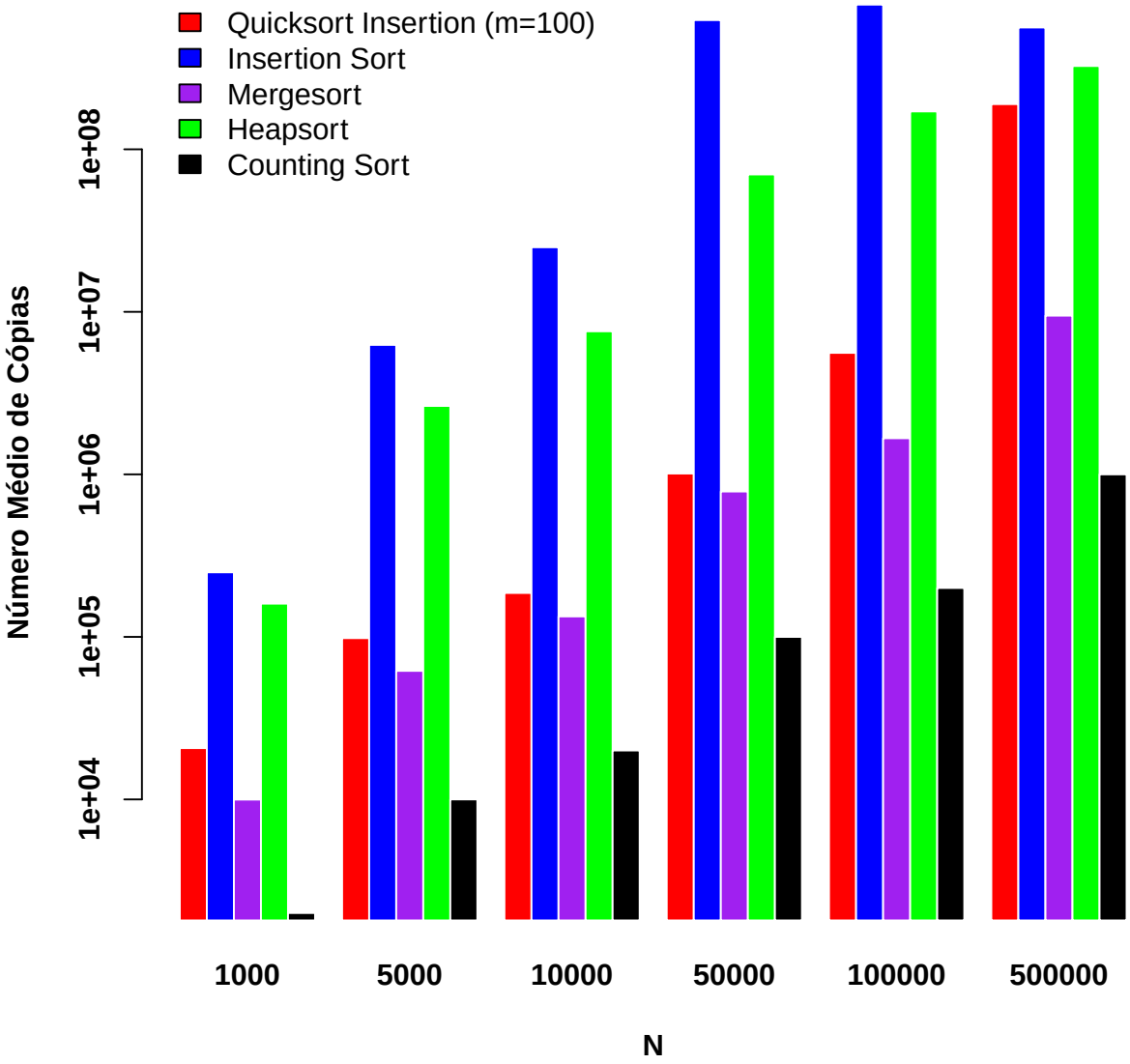
Para a avaliação do número de comparações feitas por cada algoritmo, o Counting Sort não foi considerado nos gráficos, uma vez que seu funcionamento não envolve comparações entre chaves. O Mergesort foi o algoritmo com o menor número de comparações para todos os valores de N, seguido do Quicksort Insertion (m=100). É interessante observar que, para valores grandes de N (50000, 100000, e 500000), o Insertion Sort fez mais comparações do que o Heapsort, enquanto o inverso foi observado para conjuntos de tamanhos menores. Ademais, outro ponto de interesse

é que o número de comparações desempenhadas por ambos os algoritmos, em conjuntos de tamanho N = 500000, foi menor do que a quantidade análoga, para N = 100000.

Número médio de cópias de registros

N	QuicksortInsertion (m = 100)	InsertionSort	Mergesort	Heapsort	CountingSort
1000	20691	249573	9976	160175	2000
5000	98277	6260792	61808	2636603	10000
10000	186304	24969571	133616	7591515	20000
50000	1011819	623790327	784464	70050742	100000
100000	5593326	776874100	1668928	171034064	200000
500000	189750004	561752736	9475712	325645141	1000000





Pelo gráfico acima, novamente se evidencia a maior eficiência do Counting Sort para a tarefa requerida. Nesse ponto é importante deixar claro que o número de cópias feitas pelo algoritmo é sempre o mesmo, para um mesmo valor de N. Em outras palavras, não existe "pior caso" nem "melhor caso" do ponto de vista da operação de cópia de registro.

O mesmo padrão pode ser observado em todos os valores de N: O Counting Sort realiza

menos cópias (em média), seguido do Mergesort, Quicksort Insertion (m = 100), Heapsort, e Insertion Sort.

As quantidades vistas a respeito do número de comparações e de cópias são compatíveis com os resultados sobre o tempo de CPU gasto por cada algoritmo. O Counting Sort, desta forma, foi selecionado para a tarefa de ordenação dos gastos dos deputados, a ser discutida na Seção 2.

1.4 Cenário IV: Tratamento de colisões: Endereçamento X Encadeamento

Neste cenário, foram implementados cinco algoritmos para tratamento de colisões considerando o número de comparações de chaves e o gasto de memória como métricas de desempenho. Os algoritmos implementados foram:

1. Endereçamento - Sondagem Linear;
2. Endereçamento - Sondagem Quadrática;
3. Endereçamento - Duplo Hash (ou Re-Hashing);
4. Encadeamento Separado;
5. Encadeamento Coalescido (sem porão);

Todos os algoritmos foram criados usando `TADS`, com funções básicas de inserção. A estrutura básica para todos casos ficou da seguinte forma:

- `*vetor (TIPO_DADOS)` – ponteiro para o vetor que seria criado no construtor. Para cada algoritmo, é usado o tipo de dados particular (ex: ListaEncadeada para Encadeamento Separado);
- `tamanho (int)` – tamanho do vetor que é passado como parâmetro no construtor;
- `func_Calcula_Indice (int)` – função que retorna o índice que o valor deverá ser inserido no vetor;
- `inserir (void)` – função que recebe como parâmetro o valor que deverá ser inserido;
- `calc_Comparacoes (int)` – variável para calcular a quantidade de comparações de chaves no algoritmo;

- `calc_GastoMemoria (int)` – variável para calcular o gasto de memória no algoritmo;

Alguns `TADS` podem apresentar certas diferenças. É o caso, por exemplo, do `encadeamento separado`, que utiliza a estrutura de uma `lista encadeada`. Ou seja, cada elemento do vetor desse algoritmo é constituído de uma lista encadeada. Caso ocorra colisão, o novo elemento será inserido no final da lista. Além desse caso, é possível notar outra diferença na implementação do `TAD` de `sondagem quadrática`, que utiliza mais de uma função para calcular o índice do vetor que deve ser utilizado. Caso ocorra colisão, essas outras funções serão chamadas.

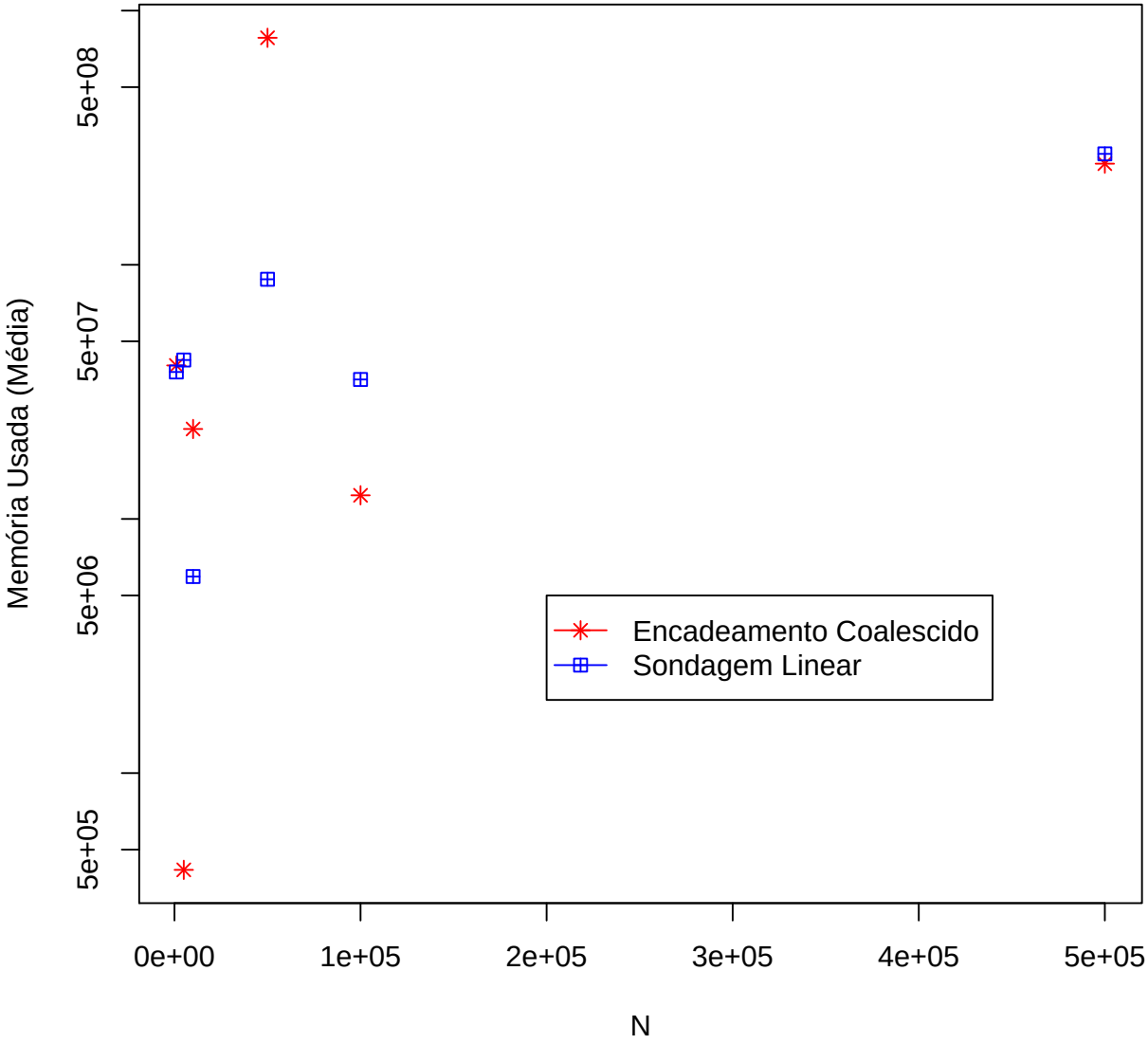
Com exceção desses casos particulares, as estruturas de todos os algoritmos de tratamento de colisões seguem basicamente o mesmo padrão. Ou seja, um vetor de inteiros e funções para definir o que deve ser feito caso ocorra colisão. Cada elemento dos vetores é constituído de um `deputy_ide` importado aleatoriamente da base de dados. Foram realizados testes com cinco valores de N diferentes: 5000, 10000, 50000, 100000 e 500000. Os algoritmos de `duplo hash`, `sondagem quadrática` e `encadeamento separado` apresentaram erros pontuais quando foram aplicados aos problemas do trabalho, porém, nos casos de teste, apresentaram os resultados esperados. Os códigos serão enviados juntamente com o relatório. Em relação a `sondagem quadrática`, foi relatado um erro de loop infinito e nos outros códigos, um aviso de `segmentation fault`. Assim, a análise se deu em cima dos algoritmos de `sondagem linear` e `encadeamento coalescido`. Os algoritmos apresentaram resultados parecidos em relação a média de gasto de memória, porém, em número de comparações de chaves, a `sondagem linear` apresentou resultados melhores em 4 casos, por isso, esse algoritmo foi escolhido para ser usado na segunda parte do trabalho.

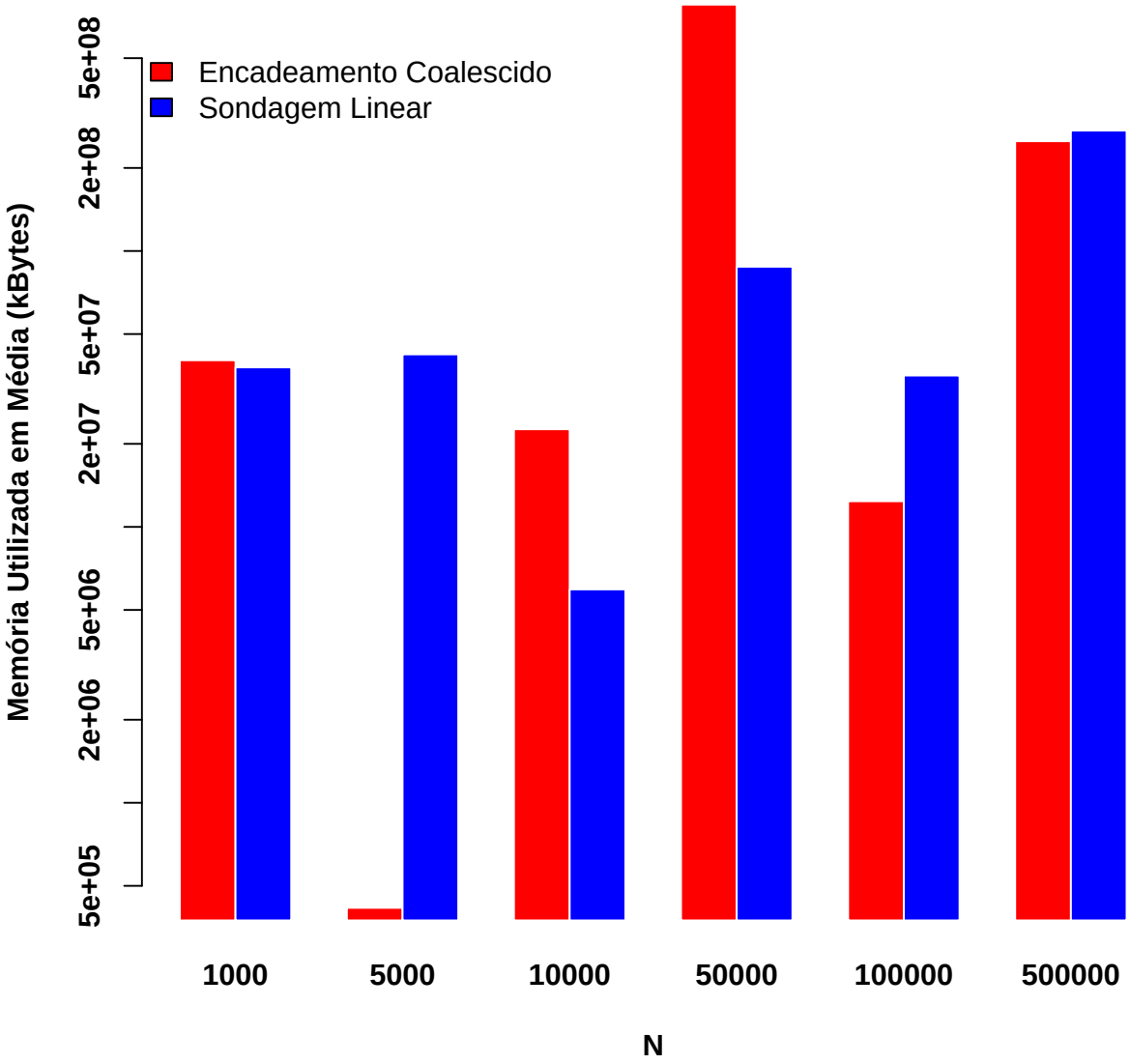
Memória Utilizada em Média (kBytes)

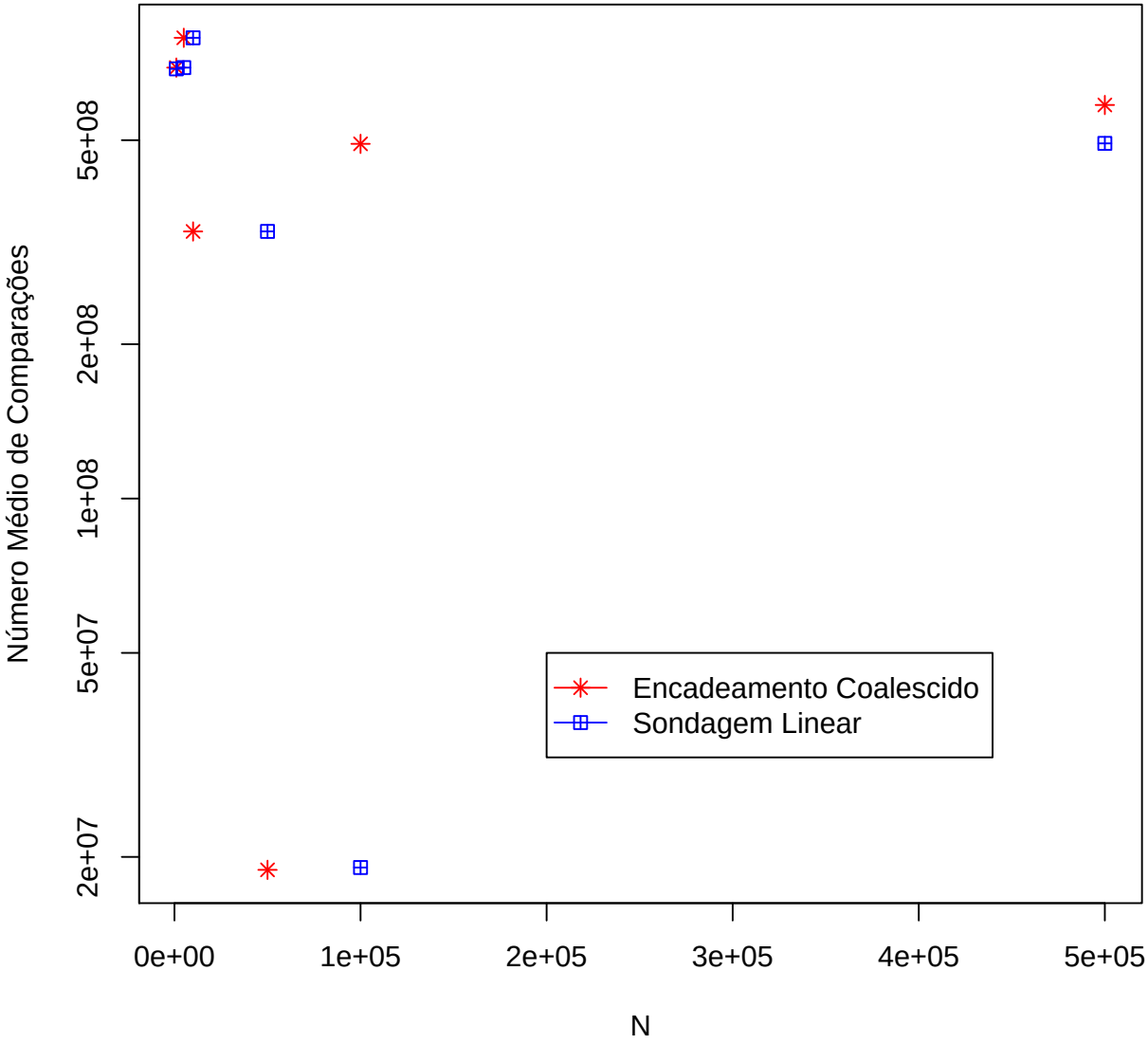
N	Sondagem Linear	Encadeamento Coalescido
1000	37860147	40173568
5000	42198630	416153
10000	5935923	22586982
50000	87696998	781705216
100000	35373056	12388761
500000	585664454	249933824

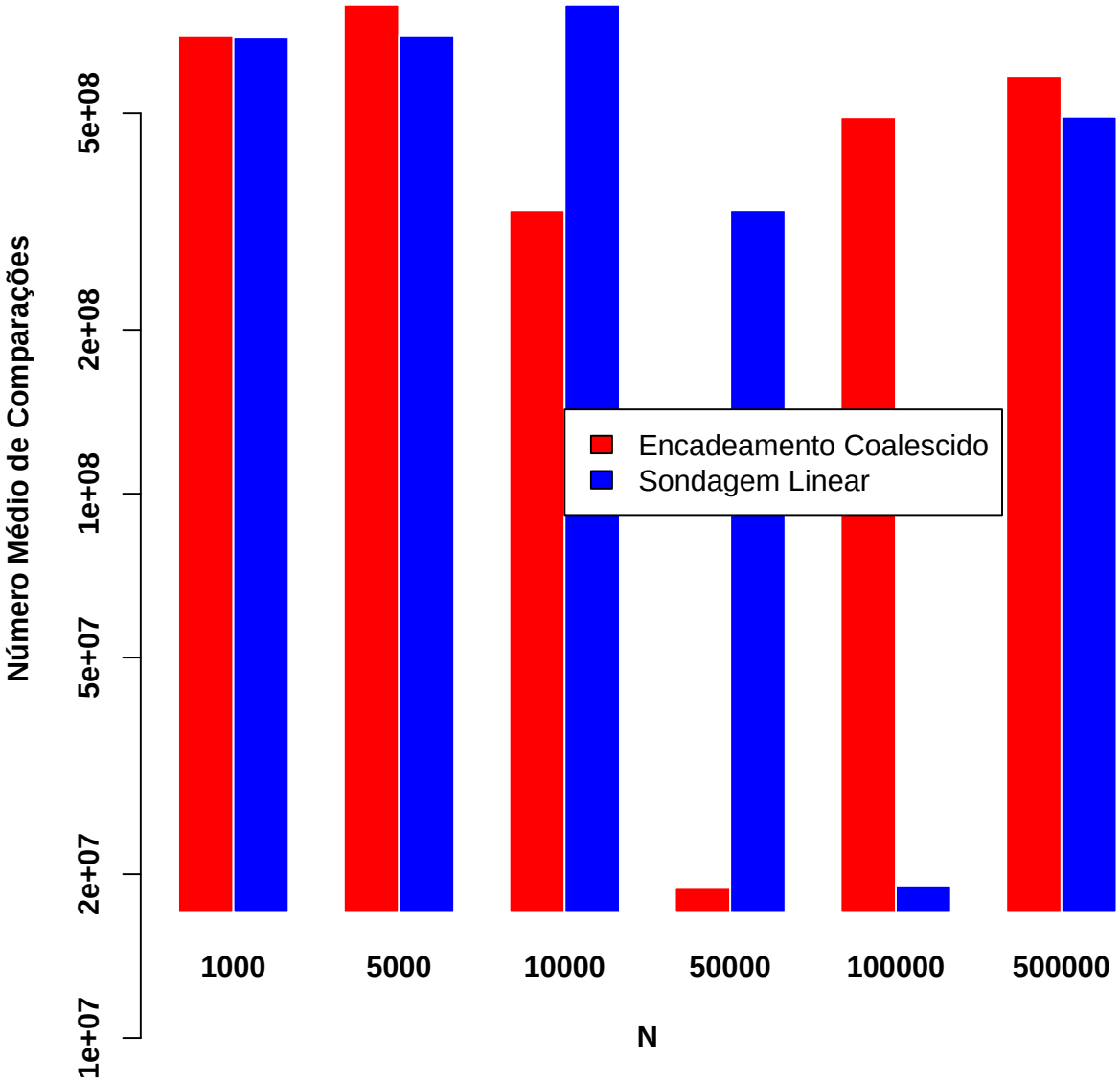
Número médio de comparações de chaves

N	Sondagem Linear	Encadeamento Coalescido
1000	688844656	692691056
5000	692701056	791881056
10000	791901056	331851596
50000	331951596	18870086
100000	19070086	491850963
500000	492850963	273176166


Memória Usada (Média)
5e+08
5e+07
5e+06
5e+05
0e+00
1e+05
2e+05
3e+05
4e+05
5e+05
N
Encadeamento Coalescido
Sondagem Linear


Encadeamento Coalescido
Sondagem Linear
Memória Utilizada em Média (kBytes)
5e+05
2e+06
5e+06
2e+07
5e+07
2e+08
5e+08
1000
5000
10000
50000
100000
500000
N


Número Médio de Comparações
2e+07
5e+07
1e+08
2e+08
5e+08
0e+00
1e+05
2e+05
3e+05
4e+05
5e+05
N
Encadeamento Coalescido
Sondagem Linear



Observando as tabelas e gráficos, é possível notar que existe uma certa variação ao longo das mudanças do tamanho do vetor. Por exemplo, para N = 1000, as médias de gasto de memória entre o método de `sondagem linear` e `encadeamento coalescido` foram muito parecidas, mas para N = 5000, passa a existir uma diferença muito grande. Isso pode ser resultado de uma característica aleatória da `sondagem linear`, pois se ocorre colisão, o elemento passa a ser inserido no próximo campo vazio, mas é algo incerto, pois é possível que existam vários

elementos preenchidos em sequência ou não. Já o `encadeamento coalescido` utiliza uma outra estrutura durante o processo de inserção, algo que pode influenciar em questões como gasto de memória. Outro destaque seria o valor das médias para N = 50000, em que se nota um aumento considerável no gasto de memória da `sondagem linear`, chegando a um número alto em relação aos outros resultados.

Em relação aos resultados do número médio de comparações, é preciso destacar as mesmas características relatadas acima. Ou seja, o algoritmo de `sondagem linear` pode apresentar um número de comparações muito alto devido a sua característica de não utilizar nenhuma função a mais no caso de colisões. Mas o gráfico apresenta um resultado de igualdade entre os dois algoritmos comparados, em que se nota valores semelhantes para N = 1000 e N = 5000, já para N = 10000, a `sondagem linear` utiliza mais comparações de chaves. Em N = 50000 e N = 100000, existe um comportamento curioso, com uma inversão de valores abrupta. Isso pode ser consequência de alguma peculiariedade dos números aleatórios gerados naquele momento. Em N = 500000, o último e maior vetor, volta-se a notar uma maior média para o encadeamento.

2 Implementação dos deputados que mais gastam

Nessa seção, foi implementado um algoritmo para o calculo dos gastos dos deputados e dos partidos a que eles são afiliados. Para esse cálculo o arquivo `deputies_dataset_tratado.csv` foi , assim como na parte 1, lido e armazenado na classe `Deputado`. Cada linha do arquivo gerou um objeto da classe `Deputado` e teve as suas informações salvas nos atributos da mesma, que são:

- `nome` – nome do deputado
- `partido` – partido a qual é afiliado
- `id`–id do deputado
- `valor` – valor do recibo
- `bugged_date`
- `receipt_date` – data do recibo
- `estado` – estado que o deputado representa
- `receipt_social_security_number` – numero do recibo

- `receipt_description` – descrição do recibo
- `establishment_name` – nome do estabelecimento

Também foi implementado os gets e sets dos atributos listados acima e assim como dito na Parte 1 `deputy_id` e `receipt_value` são armazenados como inteiros (`int`). O resto dos atributos são armazanados como strings (biblioteca `<string>`.

O calculo dos gastos foi realizado através da estrutura de dados `TabelaHash`, que foi implementada utilizando diversas variáveis e funções, como as funções hash, funções sort entre outras. Durante a realização do trabalho foi utilizadas duas Tabelas Hash, uma para os gastos dos deputados e outra para os gastos por partido. Em cada uma delas foi passado como parâmetro um objeto do tipo `Deputado` assim que esse objeto tinha os seus atributos inicializados pela leitura do arquivo. As Tabelas Hash foram formadas utilizando conjuntos de vetores que possuíam uma importância para o calculo de gastos.

1 A Tabela dos deputados usou os seguintes vetores:

   - `*id_deputado` – vetor do id dos deputados
   - `*ocupado_deputado` – vetor se a posição em questão esta ocupada
   - `*nome_deputado` – vetor com os nomes dos deputados
   - `*partido_deputado` – vetor dos partidos afiliados
   - `*valor_deputado` – vetor com os valores dos gastos
   - `*vet_deputado` – vetor com as posições ocupadas por cada deputado

2 Já a tabela dos partidos usou os vetores:

   - `*vet_partido` – vetor com as posições ocupadas por cada partido
   - `*ocupado_partido` – vetor se as posições em questão estão ocupadas
   - `*valor_partido` – vetor com os valores dos gasto
   - `*partido_partido` – vetor com o nome dos partidos

As Tabelas possuem uma variável comum, `int tam` que indica o tamanho das tabelas. As duas também apresentam contadores `cont_deputado` e `cont_partido` que são utilizados para controlar o número de elementos na tabela. Elas também apresentam funções que tem as mesmas funcionalidades e estruturas lógicas que apenas se diferenciam nos tratamentos das variáveis já que os números das mesmas e o nível de importância são diferentes. Exemplos disso

são funções `imprimeDeputados()` e `imprimePartidos()` que tem a mesma função: imprimir as tabelas mas as variáveis impressas são diferentes, por isso falarei dos pares de funções como sendo apenas uma. A primeira função utilizada nas Tabelas, ainda durante a leitura do arquivo é a operação de inserção:

```
void insereDeputados(Deputado d)
void inserePartidos (Deputado d)
```

Sempre que uma linha é lida por completa essas funções são chamadas para irem preenchendo as Tabelas com as informações dos deputados. Elas utilizam a única função que apresenta uma diferença, as funções HASH. Após a utilização da função HASH uma chave é gerada e ocorre a tentativa de inserção na posição da chave, se a posição estiver ocupada ela ira comparar um atributo específico de cada tipo de Tabela (`id_deputado` para a Tabela de Deputados e `partido_partido` para a Tabela de Partidos) com um atributo também específico do objeto `d` do tipo `Deputado` (`d.getId` e `d.getPartido` para as Tabelas de Deputados e de Partidos, respectivamente). Se os atributos comparados não forem igual ocorrerá à chamada da função de colisão. Se a posição encontrada pela função HASH estiver vazia ocorrera à inserção de forma natural, com todos os elementos do private da Tabela recebendo um valor do objeto `d` através das operações de `get`.

Dentro da função ocorre a utilização de mais dois pares de funções, a função de colisão e as funções HASH. As funções HASH utilizadas são:

```
int funcaoHash (int id)
int funcaoHashACII (string partido)
```

A `funcaoHash` recebe como parâmetro um inteiro que representam o valor do id do deputado, calcula o resto da divisão dele com o tamanho `tam` da tabela e retorna um inteiro que será usado como chave para a inserção na tabela. Já a `funcaoHashACII` recebe como parâmetro uma string representando o nome do partido inserido, por meio de operações em uma for ela transforma essa string em um valor inteiro correspondente da tabela ASCII, calcula o resto desse novo inteiro com o tamanho da tabela e retorna o resultado para ser utilizado na inserção.

Para resolver as colisões, foi utilizado o método de endereçamento por sondagem linear, conforme os resultados do Cenário IV da Parte I.

Depois da leitura de todo o arquivo ocorrerá a ordenação das Tabelas, porém antes as Tabelas serão impressas para o usuário poder analisá-las.

```
void imprimirDeputados()
 void imprimirPartidos()
```

Essas funções apenas imprimem todos os vetores utilizados em colunas para o usuário conseguir visualizar a Tabela formada. Essas funções não imprimem linhas vazias da tabela devido a um `if` que não permite a impressão se o vetor ocupado indicar que a posição esta vazia.

Depois, ocorrem as funções de organização:

```
void ordenacaoPartido()
void ordenacaoDeputado()
```

Para a implementação dessas funções foi escolhida, do cenário 3, a função `CountingSort` e será usada para organizar os gastos totais de deputados e partidos. A única diferença nas aplicações delas com a implementada no cenário 3 é que, nesse caso, as funções não recebem nenhum parâmetro e ocorre uma operação para calcular a ordenação de todos os vetores que formam a lista, a partir da ordenação dos vetores valores correspondentes. As ultimas funções utilizadas pela tabela são funções:

```
void N_maioresDeputados(int n)
void N_maioresPartidos(int n)
void N_menoresDeputados(int n)
void N_menoresPartidos(int n)
```

As duas primeiras utilizam um valor inteiro `N` escolhido pelo usuário para impressão dos `N`

últimos deputados e partidos que mais gastaram, em ordem decrescente de gastos. Já as duas últimas funções recebem o mesmo `N` que as anteriores, mas imprimem os `N` deputados e partidos que menos gastaram, utilizando um contador para pular os espaços vazios das Tabelas.

3 Rotinas Auxiliares

Nesta seção são documentadas as funções auxiliares utilizadas nas duas partes do trabalho.

`int randomInt(int from, int to)`

Argumentos:

`from` inteiro.

`to` inteiro.

Descrição:
Wrapper para `rand()`. Retorna um inteiro (pseudo)aleatoriamente selecionado, entre `from` e `to`.

`void swapPtr(int *a, int *b)`

Argumentos:

`*a` ponteiro para inteiro.

`*b` ponteiro para inteiro.

Descrição:
Troca os valores em duas posições de um vetor de inteiros, isto é, os valores apontados por `a` e `b`.

`void printVec(int *vec, int vecSize)`

Argumentos:

`*vec` ponteiro para o primeiro elemento de um vetor de inteiros.

`vecSize` número de elementos de um vetor.

Descrição:
Imprime todos os elementos de um vetor `vec`.

```
int minVec(int *vec, int vecSize)
```

Argumentos:

`*vec` ponteiro para o primeiro elemento de um vetor de inteiros.

`vecSize` número de elementos de um vetor.

Descrição:
Retorna o menor elemento em um vetor de inteiros `vec`.

```
int minVecPos(int *vec, int vecSize)
```

Argumentos:

`*vec` ponteiro para o primeiro elemento de um vetor de inteiros.

`vecSize` número de elementos de um vetor.

Descrição:
Retorna a posição do menor elemento em um vetor de inteiros `vec`.

```
double minVec(double *vec, int vecSize)
```

Argumentos:

`*vec` ponteiro para o primeiro elemento de um vetor de inteiros.

`vecSize` número de elementos de um vetor.

Descrição:
Retorna o menor elemento em um vetor de números reais (ponto flutuante com precisão dupla) `vec`.

```
int maxVec(int *vec, int vecSize)
```

Argumentos:

`*vec` ponteiro para o primeiro elemento de um vetor de inteiros.

`vecSize` número de elementos de um vetor.

Descrição:
Retorna o maior elemento em um vetor de inteiros `vec`.

```
void swapPtrDep(GastoDeputado *a, GastoDeputado *b)
```

Argumentos:

`*a` ponteiro para GastoDeputado.

`*b` ponteiro para GastoDeputado.

Descrição:
Troca os registros em duas posições de um vetor de `GastoDeputado`, isto é, os TADs apontados por `a` e `b`.

```
bool ordenado(int *vec, int vecSize)
```

Argumentos:

`*vec` ponteiro para o primeiro elemento de um vetor de inteiros.

`vecSize` número de elementos de um vetor.

Descrição:
Verifica se um vetor de inteiros `vec` está ordenado.

```
bool ordenado(GastoDeputado *vec, int vecSize);
```

Argumentos:

`*vec` ponteiro para o primeiro elemento de um vetor de GastoDeputado.

`vecSize` número de elementos de um vetor.

Descrição:
Verifica se um vetor de GastoDeputado `vec` está ordenado de acordo com a chave `deputy_id`.

4 Descrição das atividades realizadas por cada membro do grupo

As tarefas desempenhadas por cada membro do grupo para a realização deste trabalho são enumeradas a seguir.

José Santos Sá Carvalho: Código para a leitura dos dados sobre os deputados; leitura e processamento do arquivo `entrada.txt`, e escrita em arquivos `.txt` de saída. Código, testes, e relatório dos Cenários I, II, e III da Parte I do trabalho (algoritmos de ordenação, com exceção do HeapSort). Implementação e relatório das funções auxiliares enumeradas na Seção 3.

Brian Luís Coimbra Maia: Código e relatório do Cenário IV da Parte I do trabalho (algoritmos de pesquisa para resolução de colisões). Implementação do HeapSort para o Cenário III da parte I.

Matheus de Oliveira Carvalho: Implementação do programa para calcular os deputados que mais gastam (Parte 2 do trabalho), incluindo a implementação da Hash Table para o armazenamento dos gastos. Código e relatório correspondentes a essa parte.