USP – UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E COMPUTAÇÃO (ICMC)
DISCIPLINA: LABORATÓRIO DE INTRODUÇÃO A CIÊNCIA DA COMPUTAÇÃO II

RELATÓRIO 03 - EFICIÊNCIA DE ALGORITMOS DE ORDENAÇÃO

Carlos Filipe de Castro Lemos
nUSP: 12542630

Resumo

Este relatório faz uma sucinta análise assintótica dos mecanismos Bucket Sort, Counting Sort e Raddix Sort quando ordenam vetores com valores aleatórios ou quando estão preenchidos de forma crescente ou descrescente, variando-se, em todos os casos testes, a quantidade de elementos de entrada (100, 500, 1000, 5000, 7500, 10000, 50000 e 100000 elementos).
Palavras-chave: *Bucket Sort; Counting Sort; Raddix Sort; Notação Assintótica*.

1 Introdução

Após a elaboração dos *Relatórios 01 e 02 (Acerca da Eficiência de Algoritmos de Ordenação)*, os alunos deveriam elaborar um terceiro relatório, porém, dessa vez, com foco no *Bucket Sort*, *Counting Sort* e no *Raddix Sort*. Os critérios de abordagem seriam basicamente os mesmos dos trabalhos anteriores (constituição dos algoritmos, métodos de análise de eficiência, variáveis que interferiam em suas performances, notações assintóticas e outros). Assim, de forma resumida, podemos dizer que os alunos deveriam fazer a análise assintótica através da medição de marcos temporais (considerando a quantidade de entrada e a qualidade dos vetores). No entanto, além de fazer uma análise dos mecanismos em si mesmos, deveriam também fazer uma comparação com os outros cinco que foram objeto dos relatórios anteriores (Bubble Sort, Insertion Sort, Merge Sort, Quicksort e Heapsort).

Assim, os alunos poderiam confrontar as informações ministradas na literatura especializada, bem como comprovar o conteúdo lecionado pelos professores em sala de aula.

2 Metodologia e Desenvolvimento

2.1 Metodologia

A metodologia utilizada no presente relatório está diretamente relacionada com a medição de eficiência dos algoritmos de ordenação (Bucket Sort, Counting Sort e Raddix Sort). Nesse contexto, é preciso esclarecer que o termo "eficiência", embora apresente variados significados, está empregado no sentido de *rapidez* ou *velocidade*. Além disso, é conveniente explicitar que foram tomados alguns cuidados no cálculo do tempo, na obtenção dos resultados e na sua posterior comparação.

No *cálculo do tempo*, observou-se as seguintes medidas mitigadoras de erros:

1. **Casos de Teste:** os casos testes consideraram a quantidade de elementos (100, 500, 1000, 5000, 7500, 10.000, 50000 e 100.000) e a qualidade do vetor (pior e melhor caso). Para *casos aleatórios*, considerou-se vetores formados com números inteiros gerados aleatoriamente na faixa de 0 a 1000. Como *melhores casos*, considerou-se vetores formados com números devidamente ordenados em ordem crescente (supondo se tratarem de casos em que, teoricamente, deveriam ser os mais rápidos). Enquanto que *piores casos*, considerou-se vetores que foram preenchidos ordenadamente de forma decrescente (supondo mais complicados de ordenação).

2. **Geração Vetores:** valendo-se da linguagem C, os casos de testes foram gerados de forma aleatória por meio da biblioteca *<time.h>* (funções *rand()* e *srand()* - alimentadas por um *seed* baseado no horário em que foi feito o teste). Vale dizer que tal vetor foi copiado para outros de mesmo tamanho de modo que cada medição de tempo tivesse os mesmos casos testes (não importando o método de ordenação). Essa medida teve a finalidade de assegurar solidez aos dados para posterior comparação de resultados.

3. **Medição do Tempo:** a medição do tempo foi realizada por meio da biblioteca *<time.h>*, que disponibiliza a função *clock()*. Essa função permitiu realizar a demarcarção do tempo antes e depois da execução do método de ordenação e, mediante uma simples subtração, obter o tempo de execução do código.

Na *consolidação dos resultados*, adotou-se um método de prevenção/diluição de erros. Isto é, com a finalidade de evitar variações de resultados, em razão de elementos impossíveis de serem controlados, realizou-se 10 medições para cada caso teste, sendo certo que, posteriormente, calculou-se a média aritmética simples do conjunto.

Por fim, foram realizadas *comparações* entre os resultados de tempos médios calculados nos passos anteriores. Isto é, comparou-se os casos testes de mesma quantidade elementos em um mesmo método ordenação (ou seja, os casos aleatórios, melhores e piores casos), bem como entre os diferentes mecanismos de ordenação (confrontando-se os tempos médios

para casos aleatórios, melhores e piores). Além disso, para conferir a análise assintótica da literatura especializada, ou ministrada em sala de aula, tomou-se a liberdade de dividir o tempo médio pela quantidade de elementos do caso teste, proporcionando a verificação do tempo gasto para ordenar um único caso teste. Por fim, comparou-se os casos médios para ordenar uma unidade e isso permitiu avaliar a variação media de tempo (ou seja, verificou-se o impacto do aumento do caso médio no tempo de ordenação a fim de demonstrar se lhe houve aumento, diminuição ou constância).

2.2 Bucket Sort e suas Características

2.2.1 Conceito

Etimologicamente, *Bucket Sort* significa ordenação por baldes. Isso remete ao seu conceito que consiste em dividir o intervalo entre o maior e o menor valor do conjunto em subintervalores de igual tamanho (baldes). Logo em seguida, os elementos são ordenados dentro do subintervalo, seja pelo próprio Bucket Sort ou por outros mecanismos de ordenação.

2.2.2 Código-Fonte

O Bucket Sort pressupõe a criação de um arranjo auxiliar (responsável por organizar os subintervalos), bem como mecanismos que possibilitam o uso de uma lista ligada. No exemplo deste trabalho, inicialmente, o mecanismo utiliza uma busca para identificar o maior e o menor numero no conjunto a ser ordenado, bem como cria um vetor auxiliar com um intervalo entre o maior e o menor número (B). Em seguida, lê o vetor adicionndo os dados organizadamente, sendo certo que, por fim, concatena os intervalos formando um vetor ordenado. Portanto, é um código mais elaborado que os outros algoritmos:

```
typedef struct{
  int key;
} Record;

typedef struct node{
  Record elem;
  struct node* next;
} Node;

typedef struct bucket{
  Node* begin;
```

```
  Node* end;
} Bucket;

void bucket(Record* vetor, int tamanho){
  int max, min;
  max = min = vetor[0].key;
  int i = 0;
  for(i = 0; i<tamanho; i++){
    if (vetor[i].key > max) max = vetor[i].key;
    if (vetor[i].key < min) min = vetor[i].key;
  }

  Bucket* B = (Bucket*) calloc(max-min+1, sizeof(Bucket));

  for(i = 0; i<tamanho; i++){
    int posicaoKey = vetor[i].key - min;

    Node* novo = malloc(sizeof(Node));
    novo->elem = vetor[i];
    novo->next = NULL;

    if (B[posicaoKey].begin == NULL)
      B[posicaoKey].begin = novo;
    else
      (B[posicaoKey].end)->next = novo;
    B[posicaoKey].end = novo;
  }

  int j = 0;
  for(i = 0; i<=(max-min); i++){
    Node* posicao;
    posicao = B[i].begin;
    while(posicao != NULL){
      vetor[j] = posicao->elem;
      j++;

      Node *deletar = posicao;
      posicao = posicao->next;
      B[i].begin = posicao;
```

```
      free(deletar);
    }
  }
  free(B);
}
```

2.2.3 Função de Eficiência e Casos Específicos

De acordo com a literatura de referência (CORMEN, 2002, p.140-143), o Bucket Sort tem eficiência derivada da equação $\Theta(n) + n.\mathcal{O}(2 - \frac{1}{n}) = \Theta(n)$, onde *n* é o numero de elementos inseridos na estrutura de listas ligadas. Em última análise, demonstra o algoritmo complexidade de tempo linear $\mathcal{O}(n)$, sendo certo que isso somente pode ser alcançado mediante o uso de memória adicional. Por fim, é de se ressaltar que o Bucket Sort é indicado para conjuntos de dados que estão bem distribuídos no intervalo, assim como para números reais com vírgula flutuante (desaconselhado ao Counting Sort).

2.3 Counting Sort e suas Características

2.3.1 Conceito

O Counting Sort é um mecanismo de ordenação que foi desenvolvido para ordenar números discretos, sendo certo que, por meio de contagem, determina a localização exata para sua inserção. Por exemplo, se há 43 elementos, o Counting Sort calcula que o elemento X tem 25 elementos que lhe são menores e, nesse caso, insere-o na posição 26. Nessa lógica, a contagem é feita pelos seguintes passos:

a) O primeiro laço de repetição é usado para determinar qual é o tamanho do intervalo entre o maior e o menor número, bem como realiza uma copia dos elementos originais em um vetor auxiliar.

b) O segundo laço de repetição realiza a contagem de quantas vezes o elemento foi encontrado no conjunto. Isso ocorre por meio do incremento do valor do índice que lhe foi reservado no vetor de contagem. Nesse contexto, é interessante notar que, embora não exista índices negativos na linguagem C, é possível convencionar que o indice 0 seja referente o menor elemento do conjunto de entrada (mesmo que seja número negativo).

c) Com o fim do procedimento de determinação da frequência de cada elemento, calcula-se a posição de inserção no vetor original, o que é executado no terceiro laço.

d) Por fim, o último laço insere os elementos no vetor original a partir do vetores auxiliares.

2.3.2 Código-Fonte

O código fonte do algoritmo Counting Sort é relativamente simples, apresentando maior complexidade na porção matemática que determina a posição de inserção dos elementos ordenados. Além disso, funciona por meio de quatro laços de repetição não aninhados (gerando complexidade de tempo linear) e utiliza memória extra para alocação de dois vetores auxiliares (vetor de contagem de frequência e cópia do vetor original - o que provoca uso de memória extra adicional, aumentando a complexidade de espaço).

```
typedef struct{
  int key;
} Record;

void countingSort(Record* vetor, int tamanho){
  Record* vetorAux = (Record*) malloc(tamanho * sizeof(Record));
  int max, min;
  max = min = vetor[0].key;
  int i = 0;
  for(i = 1; i<tamanho; i++){
    if (vetor[i].key > max) max = vetor[i].key;
    if (vetor[i].key < min) min = vetor[i].key;
    vetorAux[i] = vetor[i];
  }

  int* vetorContagem = (int*) calloc(max-min+1, sizeof(int));

  for(i = 0; i<tamanho; i++){
    int posicaoKey = vetor[i].key - min;
    vetorContagem[posicaoKey]++;
  }

  int total = 0;
  for(i = 0; i<=(max-min); i++){
    int contagemAnterior = vetorContagem[i];
    vetorContagem[i] = total;
    total = total + contagemAnterior;
  }

  for(i = 0; i<tamanho; i++){
    int posicaoOrdenada = vetorContagem[vetorAux[i].key-min];
```

```
    vetor[posicaoOrdenada] = vetorAux[i];
    vetorContagem[vetorAux[i].key-min]++;
  }

  free(vetorContagem);
  free(vetorAux);
}
```

2.3.3 Função de Eficiência e Casos Específicos

De acordo com a literatura de referência (CORMEN, 2002, p.135-137), a complexidade de tempo para a realização completa dos laços de repetição será $\Theta(k)$ ou $\Theta(n)$, onde *k* é o número de elementos do espectro e *n* é o número de elementos a serem ordenados. Isso resulta em uma complexidade linear $\mathcal{O}(n + k)$. Como não utiliza comparações, consegue eficiência melhor que $\Omega(n.\log(n))$. Por fim, convém lembrar que, em razão de usar memória adicional, possui complexidade de espaço $\mathcal{O}(n + k)$.

Além disso, é interessante notar que este mecanismo de ordenação é centrado no intervalo entre o maior e o menor elemento dos dados de entrada. Logo, a proximidade entre tais valores e a quantidade de dados (número de elementos a serem ordenados) será fundamental para determinar sua eficiência. Ou seja, se tiver 1.000 elementos, porém com um espectro variando de 0 a 1000, podemos afirmar que o mecanismo será eficiente; mas, se o conjunto de entrada possuir apenas 4 elementos e um espectro variando de milhões de unidades, será ineficiente.

2.4 Raddix e suas Características

2.4.1 Conceito

O Raddix Sort surgiu de forma prática quando era utilizado para fazer a organização dos extintos cartões perfurados de processamento de dados. O mecanismo é inteligente, pois vale-se da lógica de ordenar sem comparar, mediante o uso de sucessivas ordenações. O conceito do método trabalha com a ordenação de algarismo por algarismo de um número. Por exemplo, no sistema decimal, o algoritmo organiza, primeiro, a casa das unidades e, em seguida, organiza a casa das dezenas, sendo certo que, tal raciocínio continua até que o maior elemento do sistema tenha sido ordenado também.

2.4.2 Código-Fonte

Em primeiro lugar, o algoritmo pesquisa o maior número do sistema e, a partir dele, controla o laço de repetição externo (o qual é responsável por percorrer digito por digito dos

números do sistema). Dentro desse laço de repetição, há a criação de intervalos regulares, como faz o Bucket Sort, sendo certo que os elementos são sequencialmente ordenados. Ao final, a ordenação está completa.

```
int findLargestNum(int * array, int size){
  int i;
  int largestNum = -1;

  for(i = 0; i < size; i++){
    if(array[i] > largestNum)
      largestNum = array[i];
  }
  return largestNum;
}

void radixSort(int * array, int size){
  int i;
  int semiSorted[size];
  int significantDigit = 1;
  int largestNum = findLargestNum(array, size);

  while (largestNum / significantDigit > 0){
    int bucket[10] = { 0 };
    for (i = 0; i < size; i++){bucket[(array[i] / significantDigit) % 10]++;}
    for (i = 1; i < 10; i++) {bucket[i] += bucket[i - 1];}
    for (i = size - 1; i >= 0; i--)
semiSorted[--bucket[(array[i] / significantDigit) % 10]] = array[i];
    for (i = 0; i < size; i++){array[i] = semiSorted[i];}
    significantDigit *= 10;
  }
}
```

2.4.3 Função de Eficiência e Casos Específicos

De acordo com a literatura de referência (CORMEN, 2002, p.137-139), o funcionamento do Raddix Sort varia de acordo com $\Theta(d(n + k))$, sendo *n* a quantidade de números e *d* o número de digitos que cada número possui. Ressalte-se que, quando a quantidade de digitos for constante, o tempo de execução do Raddix Sort será $\mathcal{O}(n)$, sendo, portanto, inferior a $\Omega(n.\log(n))$. Vale acrescentar que o Raddix Sort faz uso de memória extra para a criação de

vetores auxiliares, o que aumenta sua complexidade de espaço; mas, caso use outros mecanismos de ordenação intermédiários, como o Counting Sort ou o Bucket Sort, esse acréscimo será ainda maior e, portanto, desaconselhado a sistemas com memória limitada.

3 Resultados

3.1 Analise Assintótica do Bucket Sort

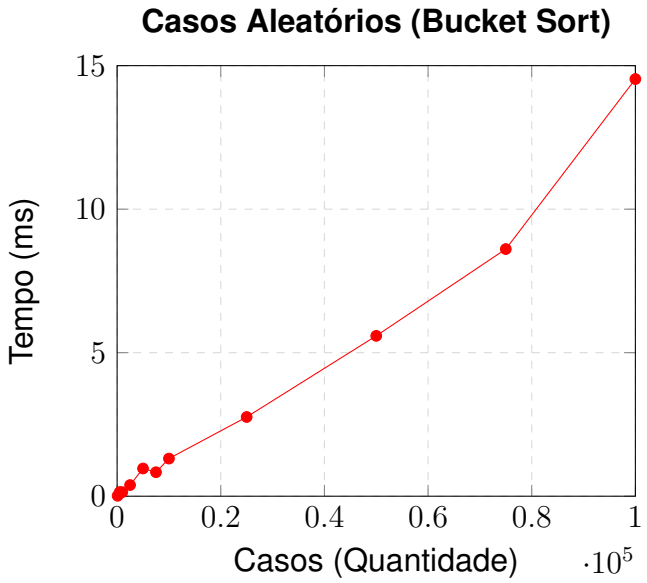
3.1.1 Casos Aleatórios (100, 500, 1000, 5000, 7500, 10000, 50000 e 100000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

CASOS ALEATÓRIOS BUCKET SORT (em ms)

Teste	A100	A500	A1.000	A5.000	A7.500	A10.000	A50.000	A100.000
1	0,01700	0,04500	0,08700	0,43000	0,85500	1,10100	5,97900	11,77100
2	0,01300	0,17500	0,07300	0,94000	0,83400	1,10500	6,43300	13,39200
3	0,02000	0,04100	0,15100	1,07100	0,60800	0,95700	6,60800	13,68100
4	0,01400	0,72300	0,15100	0,63800	1,02100	0,82900	5,84600	10,67300
5	0,01100	0,04900	0,30100	0,76400	0,77800	3,13400	5,51300	14,97800
6	0,01500	0,05100	0,07500	0,42100	0,90100	1,20300	6,51600	13,67000
7	0,01700	0,04500	0,29200	1,39600	0,73400	0,88600	4,19800	13,96100
8	0,01500	0,22000	0,07700	0,51900	0,99100	1,13800	5,33200	17,84300
9	0,01300	0,04800	0,14000	2,69800	0,86100	1,35800	4,21700	22,60400
10	0,01300	0,04700	0,07100	0,78000	0,78000	1,40000	5,22100	12,75400
Média	**0,01480**	**0,14440**	**0,14180**	**0,96570**	**0,83630**	**1,31110**	**5,58630**	**14,53270**
1 caso	**0,00015**	**0,00029**	**0,00014**	**0,00019**	**0,00011**	**0,00013**	**0,00011**	**0,00015**
Variação	**1,00000**	**1,95135**	**0,95811**	**1,30500**	**0,75342**	**0,88588**	**0,75491**	**0,98194**

A partir dos dados da tabela, verificamos que as ordenações do Bucket Sort foram estáveis, sendo certo que o tempo médio para a ordenação de um único elemento ficou próximo dos 0,0015 ms, apresentando pequenas variações erráticas em relação ao primeiro caso de ordenação. Esses dados confirmam as expectativas iniciais na medida em que mostram um crescimento próximo ao crescimento assintótico linear, conforme podemos vislumbrar no gráfico abaixo:



3.1.2 Melhores Casos (100, 500, 1000, 5000, 7500, 10000, 50000 e 100000 elementos)

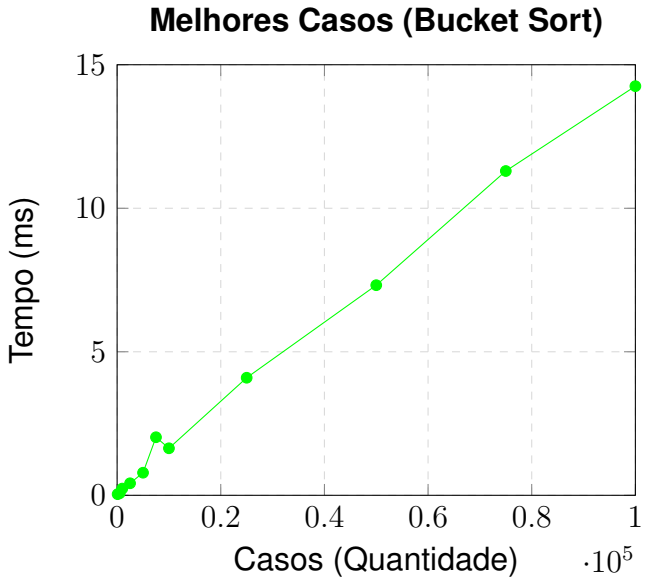
Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

MELHORES CASOS BUCKET SORT (em ms)

Teste	M100	M500	M1.000	M5.000	M7.500	M10.000	M50.000	M100.000
1	0,01100	0,04800	0,09300	0,60400	1,29000	2,17300	6,94600	12,71500
2	0,01400	0,11700	0,07800	0,66900	2,46500	1,39600	5,92100	12,26400
3	0,01200	0,17600	0,08700	0,86600	1,17100	1,52700	7,65200	13,15000
4	0,08200	0,04700	0,14800	0,71300	1,21000	1,55700	9,33700	13,70100
5	0,07900	0,11100	0,22000	0,56300	1,42900	1,26700	7,13900	12,69000
6	0,01400	0,04900	0,09000	0,95300	2,04300	1,66100	7,48700	15,36700
7	0,01200	0,04600	0,09200	0,95200	1,01900	1,43600	6,39000	17,24900
8	0,08500	0,04500	0,36900	1,00700	1,38500	1,75300	6,30300	13,22000
9	0,01000	0,04600	0,21900	0,82300	1,65200	1,89300	8,43900	16,20700
10	0,07600	0,04900	0,94000	0,71900	6,56600	1,71900	7,58500	15,96700
Média	**0,03950**	**0,07340**	**0,23360**	**0,78690**	**2,02300**	**1,63820**	**7,31990**	**14,25300**
1 caso	**0,00040**	**0,00015**	**0,00023**	**0,00016**	**0,00027**	**0,00016**	**0,00015**	**0,00014**
Variação	**1,00000**	**0,37165**	**0,59139**	**0,39843**	**0,68287**	**0,41473**	**0,37063**	**0,36084**

A partir dos dados da tabela, verificamos que as ordenações do Bucket Sort foram estáveis, sendo certo que o tempo médio para a ordenação de um único elemento ficou próximo dos 0,0001 ms, apresentando pequenas variações erráticas em relação ao primeiro caso de ordenação, bem como acentuada velocidade em relação aos casos aleatórios (uma vez que possui elementos uniformemente distribuídos no conjunto de entrada). Esses dados confirmam as expectativas iniciais na medida em que mostram um crescimento próximo ao crescimento

assintótico linear, conforme podemos vislumbrar no gráfico abaixo:



3.1.3 Pior Caso (100, 500, 1000, 5000, 7500, 10000, 50000 e 100000 elementos)

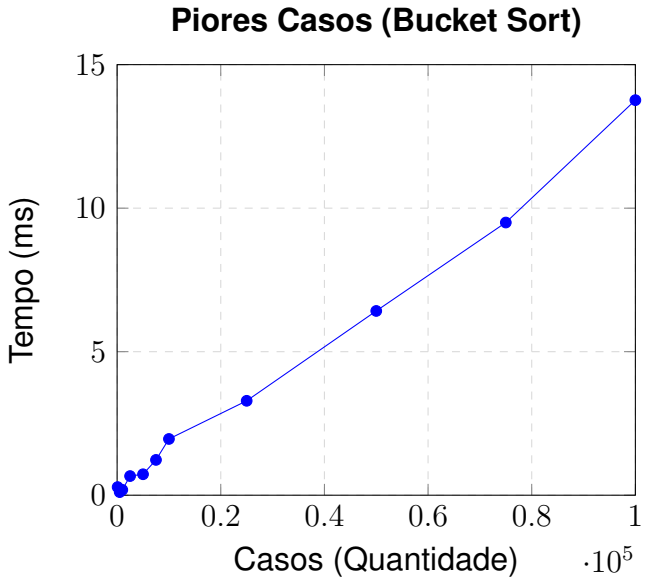
Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

PIORES CASOS BUCKET SORT (em ms)

Teste	P100	P500	P1.000	P5.000	P7.500	P10.000	P50.000	P100.000
1	0,01400	0,04800	0,09300	0,66800	0,96200	1,15400	5,90900	15,92200
2	0,01400	0,14700	0,07900	1,30000	1,29600	1,80500	7,34600	21,41500
3	0,01200	0,04600	0,09000	0,64100	1,20900	1,66300	5,29000	13,72200
4	0,01200	0,11100	0,15700	0,67000	1,99800	1,44500	9,03800	10,97500
5	0,01200	0,04600	0,31700	0,88200	1,89500	6,05700	6,43400	11,94200
6	0,01200	0,04900	0,25200	0,69300	0,93300	1,21200	6,82700	11,77900
7	0,01300	0,34200	0,52700	0,62300	1,00000	1,43100	5,60000	14,70500
8	0,01100	0,04400	0,15800	0,61500	0,99500	1,28900	5,47600	11,95500
9	0,01300	0,19800	0,09600	0,60600	0,98800	1,38300	6,84000	13,39600
10	0,17000	0,04800	0,09700	0,60300	1,02400	2,15700	5,43100	11,81700
Média	**0,02830**	**0,10790**	**0,18660**	**0,73010**	**1,23000**	**1,95960**	**6,41910**	**13,76280**
1 caso	**0,00028**	**0,00022**	**0,00019**	**0,00015**	**0,00016**	**0,00020**	**0,00013**	**0,00014**
Variação	**1,00000**	**0,76254**	**0,65936**	**0,51597**	**0,57951**	**0,69244**	**0,45365**	**0,48632**

A partir dos dados da tabela, verificamos que as ordenações do Bucket Sort foram estáveis, sendo certo que o tempo médio para a ordenação de um único elemento ficou próximo dos 0,0002 ms, apresentando pequenas variações erráticas em relação ao primeiro caso de ordenação, bem como acentuada velocidade em relação aos casos aleatórios (uma vez que pos-

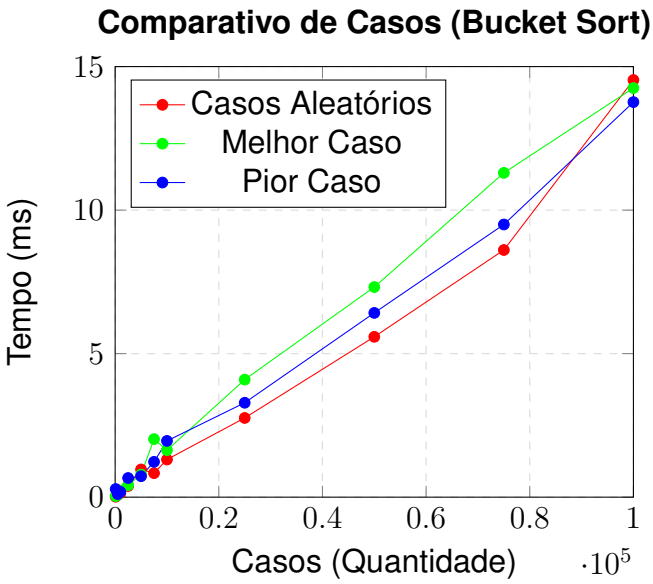
sui elementos uniformemente distribuídos no conjunto de entrada). Esses dados confirmam as expectativas iniciais na medida em que mostram um crescimento próximo ao crescimento assintótico linear, conforme podemos vislumbrar no gráfico abaixo:



3.1.4 Comparativo de Casos

A partir dos dados obtidos nos itens anteriores, podemos concluir haver uma compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório, uma vez que denotam eficiência linear de tempo ($\mathcal{O}(n)$).

No entanto, é necessário ressaltar que o melhor e o pior caso não foram constatados pela metodologia do presente trabalho, uma vez que se considerava como melhores e piores casos aqueles em que o vetor possui elementos ordenados, ou seja, elementos muito bem distribuidos, o que confere eficiência ao mecanismo do Bucket Sort.



3.2 Analise Assintótica do Counting Sort

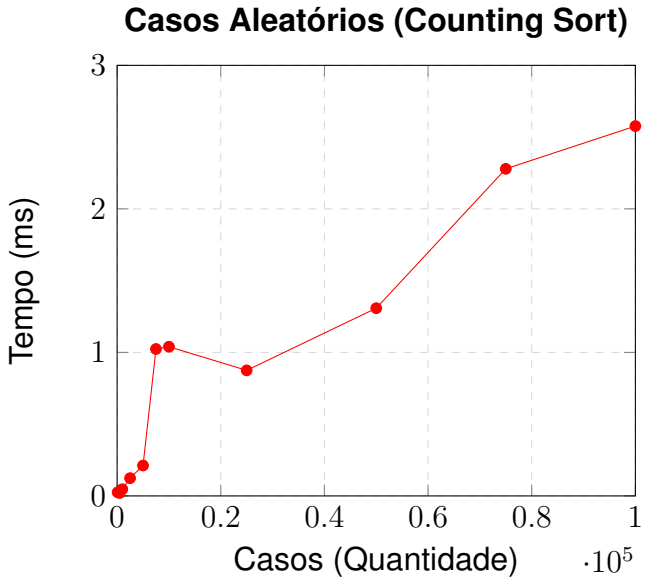
3.2.1 Casos Aleatórios (100, 500, 1000, 5000, 7500, 10000, 50000 e 100000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

CASOS ALEATÓRIOS (em ms)

Teste	A100	A500	A1.000	A5.000	A7.500	A10.000	A50.000	A100.000
1	0,00400	0,01300	0,02600	0,11400	0,17200	0,22100	1,48700	2,82500
2	0,00400	0,01200	0,02600	0,10700	0,44300	0,98800	1,24700	2,67000
3	0,00300	0,09400	0,02400	0,10700	0,62100	0,42600	1,24700	2,40900
4	0,00300	0,01300	0,02400	0,46500	0,88200	0,86800	1,34200	2,73100
5	0,00300	0,01200	0,02500	0,10500	0,33400	0,20400	1,11900	2,38700
6	0,00300	0,01200	0,02600	0,10500	0,49900	0,37800	1,27500	2,53000
7	0,00300	0,01300	0,02600	0,19900	0,51300	0,60000	1,58800	2,38300
8	0,00300	0,01200	0,02500	0,35500	0,30000	4,54800	1,15000	2,42300
9	0,00300	0,01200	0,09100	0,23700	6,31200	0,78000	1,49800	2,98000
10	0,00300	0,01200	0,17500	0,32300	0,16000	1,38300	1,12800	2,43000
Média	**0,02500**	**0,02050**	**0,04680**	**0,21170**	**1,02360**	**1,03960**	**1,30810**	**2,57680**
1 caso	**0,00003**	**0,00004**	**0,00005**	**0,00004**	**0,00014**	**0,00010**	**0,00003**	**0,00003**
Variação	**1,00000**	**1,28125**	**1,46250**	**1,32313**	**4,26500**	**3,24875**	**0,81756**	**0,80525**

A partir dos dados da tabela, verificamos que as ordenações do Counting Sort foram estáveis dentro dos casos testes de mesmas quantidades. No entanto, embora estejam longe de caracterizar crescimento quadrático ou logarítmico, esses dados não confirmam as expectativas iniciais, uma vez que há variação significativa de uma quantidade a outra, distante ao crescimento linear, conforme podemos vislumbrar no gráfico abaixo:



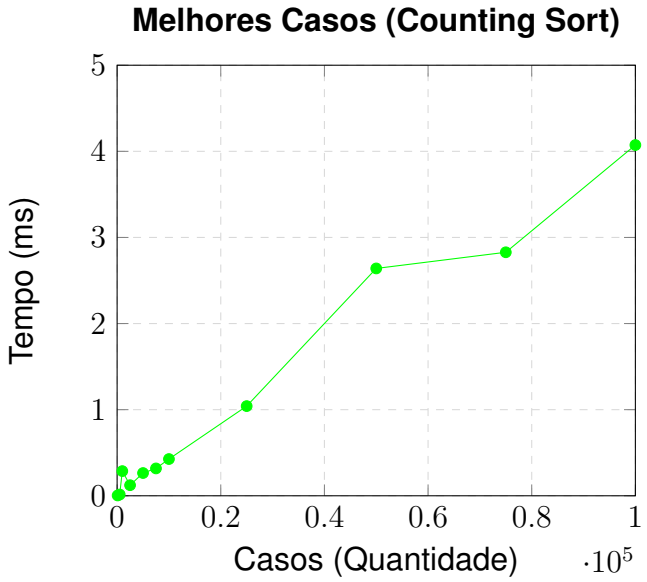
3.2.2 Melhores Casos (100, 500, 1000, 5000, 7500, 10000, 50000 e 100000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

MELHORES CASOS (em ms)

Teste	M100	M500	M1.000	M5.000	M7.500	M10.000	M50.000	M100.000
1	0,00500	0,01400	0,03300	0,29600	0,24700	0,34200	2,21800	5,18900
2	0,00300	0,01500	0,03000	0,15600	0,31300	0,29900	2,29300	5,34900
3	0,00400	0,01500	0,11900	0,15000	0,22500	0,67600	3,03400	3,82100
4	0,00400	0,01500	0,03200	0,29800	0,23300	0,63000	2,12100	3,08400
5	0,00300	0,01500	0,96000	0,60000	0,53100	0,62700	6,05100	3,57000
6	0,00300	0,01400	0,96000	0,14500	0,23400	0,31500	1,92900	5,03200
7	0,00400	0,01300	0,25700	0,15200	0,22800	0,39000	2,32700	3,74700
8	0,00400	0,01400	0,03000	0,51300	0,60200	0,33500	2,25800	3,21400
9	0,00400	0,01600	0,41400	0,16300	0,22600	0,33400	1,83200	3,62200
10	0,00400	0,01600	0,03100	0,16900	0,33800	0,31900	2,33800	4,10200
Média	**0,00380**	**0,01470**	**0,28660**	**0,26420**	**0,31770**	**0,42670**	**2,64010**	**4,07300**
1 caso	**0,00004**	**0,00003**	**0,00029**	**0,00005**	**0,00004**	**0,00004**	**0,00005**	**0,00004**
Variação	**1,00000**	**0,77368**	**7,54211**	**1,39053**	**1,11474**	**1,12289**	**1,38953**	**1,07184**

A partir dos dados da tabela, verificamos que as ordenações do Counting Sort foram estáveis, sendo certo que o tempo médio para a ordenação de um único elemento ficou próximo dos 0,00004 ms, apresentando pequenas variações erráticas em relação ao primeiro caso de ordenação, exceto nos casos de 50.000 elementos, que destoam dos demais casos. Esses dados confirmam as expectativas iniciais na medida em que mostram um crescimento próximo ao crescimento assintótico linear, conforme podemos vislumbrar no gráfico abaixo:



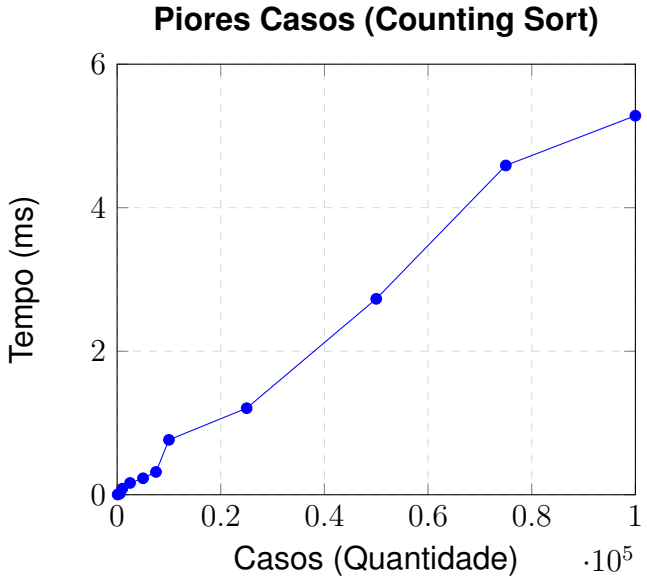
3.2.3 Pior Caso (100, 500, 1000, 5000, 7500, 10000, 50000 e 100000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

PIORES CASOS (em ms)

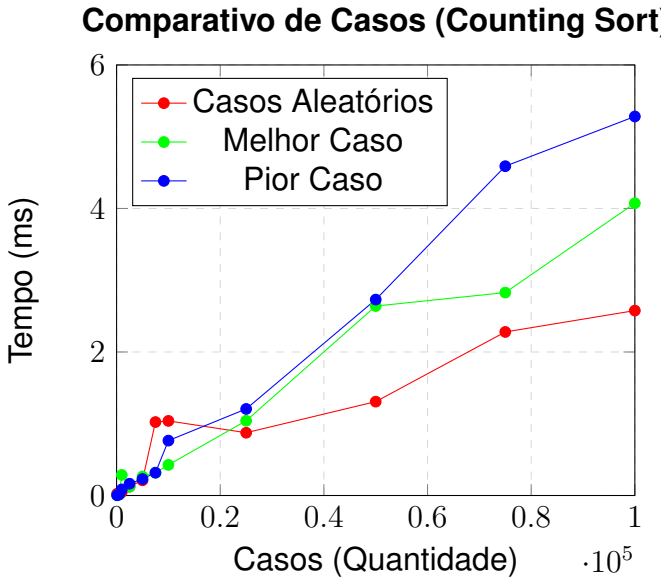
Teste	P100	P500	P1.000	P5.000	P7.500	P10.000	P50.000	P100.000
1	0,00400	0,01500	0,03300	0,16600	0,31200	0,32300	2,20700	3,79700
2	0,00400	0,01500	0,02700	0,15400	0,41900	0,43000	2,23400	3,92900
3	0,00300	0,01500	0,02900	0,13700	0,23700	0,94100	7,28900	8,91700
4	0,00300	0,01400	0,02900	0,29100	0,40200	0,77500	3,21100	3,89500
5	0,00300	0,01400	0,09300	0,54700	0,25200	0,64700	1,73900	9,65000
6	0,00300	0,01400	0,32300	0,15000	0,42800	0,94400	1,73900	3,99000
7	0,00300	0,01700	0,09500	0,15100	0,24200	0,89600	2,13400	4,64700
8	0,00300	0,01500	0,09500	0,40500	0,40000	1,65600	1,88400	6,57800
9	0,00300	0,01500	0,09400	0,15600	0,22800	0,37300	2,59300	3,79000
10	0,00300	0,01500	0,03000	0,14200	0,25800	0,66500	2,27200	3,62700
Média	**0,00320**	**0,01490**	**0,08480**	**0,22990**	**0,31780**	**0,76500**	**2,73020**	**5,28200**
1 caso	**0,00003**	**0,00003**	**0,00008**	**0,00005**	**0,00004**	**0,00008**	**0,00005**	**0,00005**
Variação	**1,00000**	**0,93125**	**2,65000**	**1,43688**	**1,32417**	**2,39063**	**1,70638**	**1,65063**

A partir dos dados da tabela, verificamos que as ordenações do Counting Sort foram estáveis, sendo certo que o tempo médio para a ordenação de um único elemento ficou próximo dos 0,00005 ms, apresentando pequenas variações erráticas em relação ao primeiro caso de ordenação. Esses dados confirmam as expectativas iniciais na medida em que mostram um crescimento próximo ao crescimento assintótico linear, conforme podemos vislumbrar no gráfico abaixo:



3.2.4 Comparativo de Casos

Comparando-se os casos do acima, por meio do gráfico abaixo, contatamos que o Counting Sort é um mecanismo de ordenação muito eficiente em todos os casos, mas, nos casos de testes do presente trabalho, demonstrou maior efetividade na ordenação de vetores preechidos com números aleatórios.



Além disso, os dados obtidos nos itens anteriores, exceto nos casos de vetores preenchidos com elementos aleatórios, revelam uma compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório, uma vez que denotam eficiência linear do tempo ($\mathcal{O}(n)$).

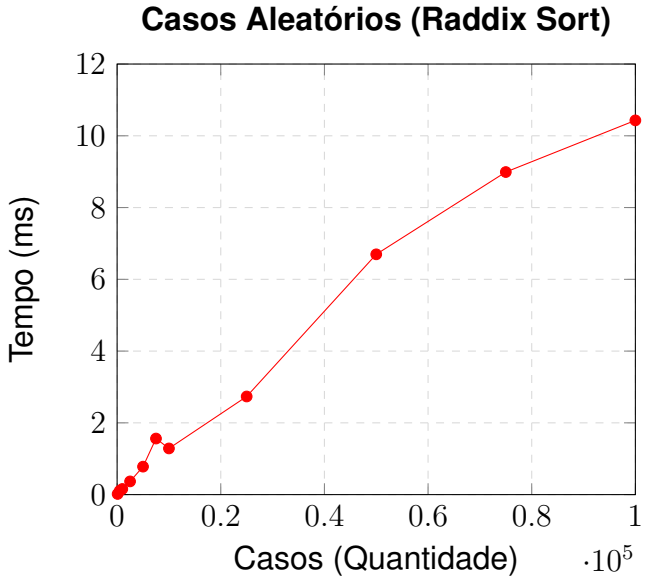
3.3 Analise Assintótica do Raddix Sort

3.3.1 Casos Aleatórios (100, 500, 1000, 5000, 7500, 10000, 50000 e 100000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

CASOS ALEATÓRIOS (em ms)								
Teste	**A100**	**A500**	**A1.000**	**A5.000**	**A7.500**	**A10.000**	**A50.000**	**A100.000**
1	0,01000	0,11300	0,08500	0,43000	1,32300	1,50100	5,40000	9,99400
2	0,00900	0,05100	0,15600	1,07500	0,96100	1,39100	4,73600	10,51700
3	0,00900	0,04500	0,29300	0,93000	3,57600	1,24600	6,65500	11,51800
4	0,00900	0,05500	0,08800	0,75000	1,96700	0,96100	5,00900	11,82200
5	0,09800	0,04600	0,22900	0,51000	1,05200	0,97000	4,56800	9,62500
6	0,01000	0,25000	0,08900	0,75100	1,34300	1,05900	8,04800	10,26700
7	0,00900	0,13400	0,30100	1,20600	0,77800	1,76700	8,24100	10,36100
8	0,01000	0,18100	0,15700	0,44000	0,96300	1,59500	11,54200	9,82500
9	0,00900	0,11400	0,08600	1,13100	0,87500	1,11300	7,96900	9,92300
10	0,00900	0,07000	0,08600	0,57000	2,78700	1,26500	4,79900	10,44400
Média	**0,01820**	**0,10590**	**0,15700**	**0,77930**	**1,56250**	**1,28680**	**6,69670**	**10,42960**
1 caso	**0,00018**	**0,00021**	**0,00016**	**0,00016**	**0,00021**	**0,00013**	**0,00013**	**0,00010**
Variação	**1,00000**	**1,16374**	**0,86264**	**0,85637**	**1,14469**	**0,70703**	**0,73590**	**0,57305**

A partir dos dados da tabela, verificamos que as ordenações do Raddix Sort foram estáveis, sendo certo que o tempo médio para a ordenação de um único elemento ficou próximo dos 0,00016 ms, apresentando pequenas variações erráticas em relação ao primeiro caso de ordenação. Esses dados confirmam as expectativas iniciais na medida em que mostram um crescimento próximo ao crescimento assintótico linear, conforme podemos vislumbrar no gráfico abaixo:



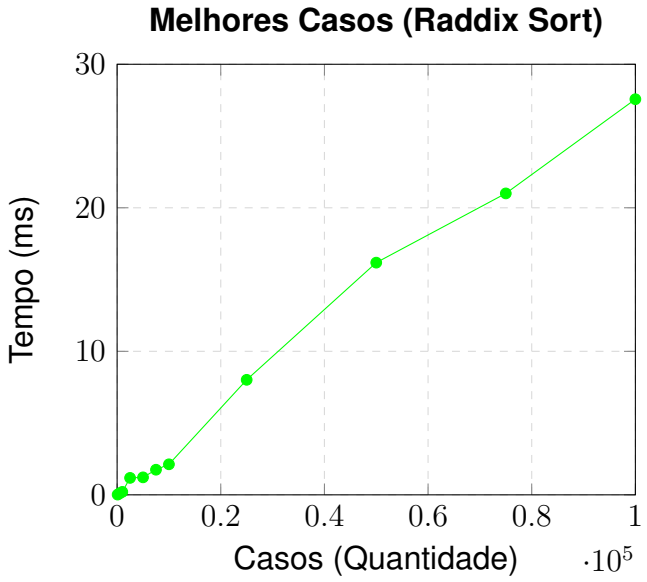
3.3.2 Melhores Casos (100, 500, 1000, 5000, 7500, 10000, 50000 e 100000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

MELHORES CASOS (em ms)

Teste	M100	M500	M1.000	M5.000	M7.500	M10.000	M50.000	M100.000
1	0,01100	0,06600	0,01310	0,97000	1,49700	2,03200	15,71800	30,75700
2	0,01100	0,20600	0,13100	0,96900	1,54500	1,95900	14,36000	27,91700
3	0,07700	0,24300	0,46600	1,26600	1,63800	2,06400	15,95800	27,42600
4	0,01000	0,08400	0,23300	1,44100	1,93000	1,84900	23,86900	27,96800
5	0,00900	0,07800	0,13900	1,22500	1,63600	2,02800	14,92000	27,57200
6	0,00900	0,08100	0,22900	1,44100	1,79800	2,37100	15,73000	26,69900
7	0,00900	0,07900	0,13400	1,14100	2,13400	2,15500	16,32500	27,74200
8	0,00900	0,08300	0,36500	1,15600	1,90300	2,73700	15,00800	27,35400
9	0,00900	0,08100	0,13000	1,45600	1,84700	2,02600	14,67100	25,50600
10	0,01000	0,08600	0,22500	1,11300	1,56300	2,05400	15,21700	26,70900
Média	**0,01640**	**0,10870**	**0,20651**	**1,21780**	**1,74910**	**2,12750**	**16,17760**	**27,56500**
1 caso	**0,00016**	**0,00022**	**0,00021**	**0,00024**	**0,00023**	**0,00021**	**0,00032**	**0,00028**
Variação	**1,00000**	**1,32561**	**1,25921**	**1,48512**	**1,42203**	**1,29726**	**1,97288**	**1,68079**

A partir dos dados da tabela, verificamos que as ordenações do Raddix Sort foram estáveis, sendo certo que o tempo médio para a ordenação de um único elemento ficou próximo dos 0,00024 ms, apresentando pequenas variações erráticas em relação ao primeiro caso de ordenação. Esses dados confirmam as expectativas iniciais na medida em que mostram um crescimento próximo ao crescimento assintótico linear, conforme podemos vislumbrar no gráfico abaixo:



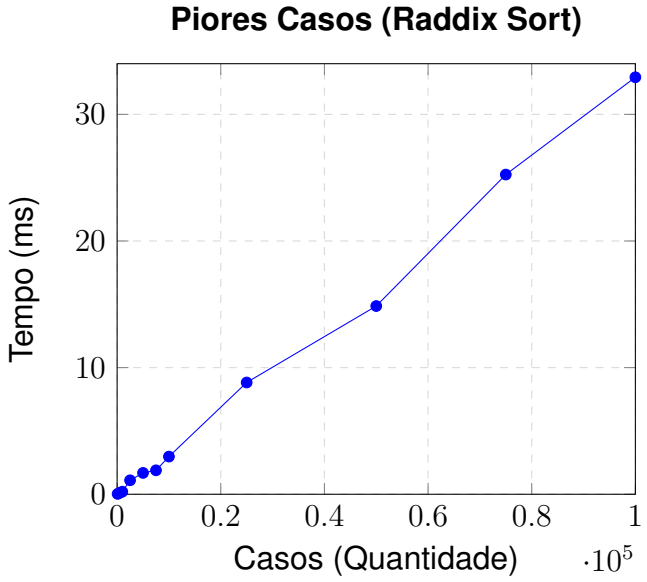
3.3.3 Pior Caso (100, 500, 1000, 5000, 7500, 10000, 50000 e 100000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

PIORES CASOS (em ms)

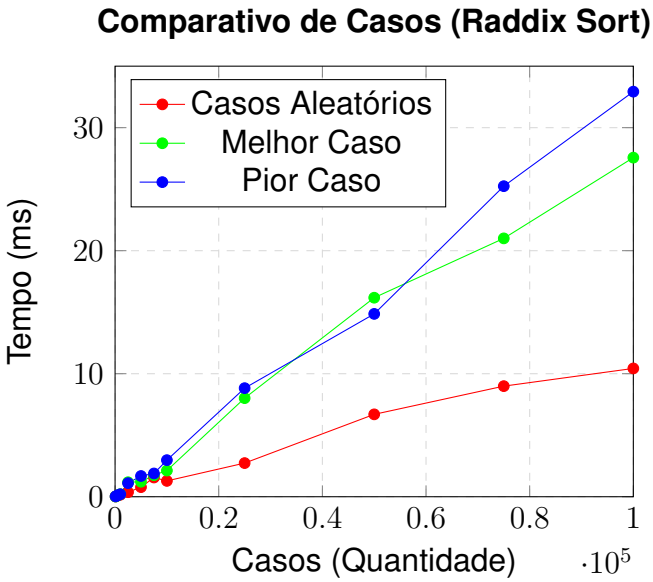
Teste	P100	P500	P1.000	P5.000	P7.500	P10.000	P50.000	P100.000
1	0,01500	0,06500	0,20000	1,00900	1,50200	2,45900	13,73900	35,85900
2	0,01500	0,14900	0,19000	0,99200	1,83900	3,07100	21,48500	33,59100
3	0,01500	0,06600	0,19300	2,10300	2,24200	2,70700	13,34100	33,17500
4	0,01300	0,13500	0,18900	2,36800	1,91800	2,60400	15,36900	32,79900
5	0,01400	0,27400	0,19300	2,66000	1,72500	2,51600	13,75000	32,55900
6	0,01500	0,01410	0,26900	1,54200	1,59400	2,79600	13,89400	31,61400
7	0,01600	0,22200	0,26700	0,88700	1,91100	2,91500	14,32000	31,73500
8	0,01700	0,06600	0,18700	1,34400	1,76000	2,39900	14,61500	32,06900
9	0,08300	0,06500	0,17500	1,31900	2,10500	4,48600	14,26600	32,67400
10	0,08200	0,06600	0,17300	2,63000	2,31000	3,78800	13,84800	33,21200
Média	**0,02850**	**0,11221**	**0,20360**	**1,68540**	**1,89060**	**2,97410**	**14,86270**	**32,92870**
1 caso	**0,00029**	**0,00022**	**0,00020**	**0,00034**	**0,00025**	**0,00030**	**0,00030**	**0,00033**
Variação	**1,00000**	**0,78744**	**0,71439**	**1,18274**	**0,88449**	**1,04354**	**1,04300**	**1,15539**

A partir dos dados da tabela, verificamos que as ordenações do Raddix Sort foram estáveis, sendo certo que o tempo médio para a ordenação de um único elemento ficou próximo dos 0,00029 ms, apresentando pequenas variações erráticas em relação ao primeiro caso de ordenação. Esses dados confirmam as expectativas iniciais na medida em que mostram um crescimento próximo ao crescimento assintótico linear, conforme podemos vislumbrar no gráfico abaixo:



3.3.4 Comparativo de Casos

Comparando-se os casos do acima, por meio do gráfico abaixo, contatamos que o Raddix Sort é um mecanismo de ordenação muito eficiente em todos os casos, mas demonstra melhor desempenho na ordenação de casos aleatórios.

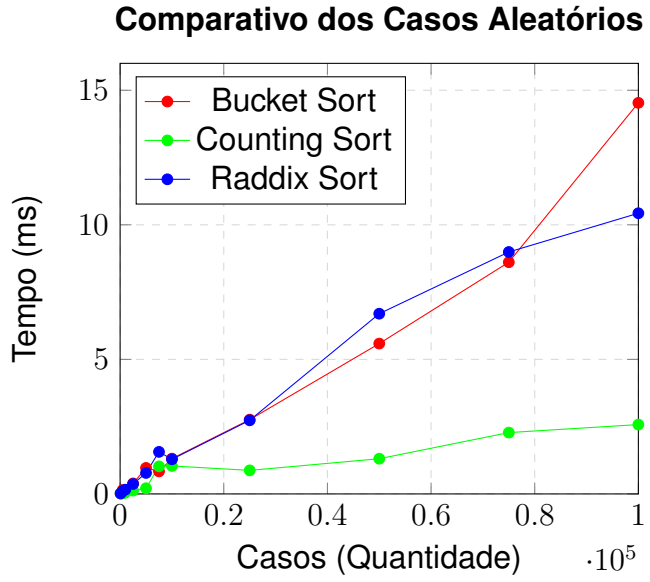


Além disso, os dados obtidos nos itens anteriores revelam uma compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório, uma vez que denotam eficiência linear de tempo ($\mathcal{O}(n)$).

3.4 Comparativo do Bucket Sort, Counting Sort e Raddix Sort

3.4.1 Comparativo dos Casos Aleatórios

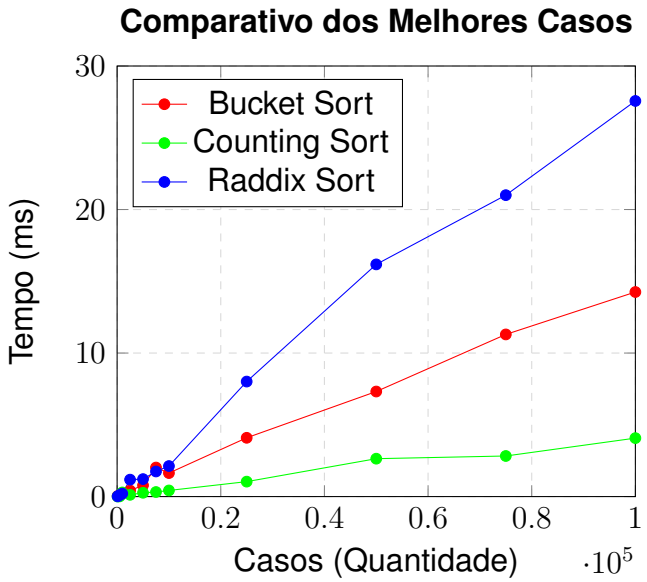
Os casos de testes com elementos aleatórios do Bucket Sort, Counting Sort e do Raddix Sort foram plotados conjuntamente no gráfico abaixo.



Tendo em vista essas informações, a comparação revela que não há diferença significativa entre a eficiência do Bucket Sort e do Raddix Sort para vetores pequenos e intermediários até 75.000 elementos, sendo certo que, em toda a extensão do gráfico, o Counting Sort foi mais eficiente. Porém, a partir dos 75.000 elementos, o Bucket Sort foi mais lento em relação ao Raddix Sort.

3.4.2 Comparativo dos Melhores Casos

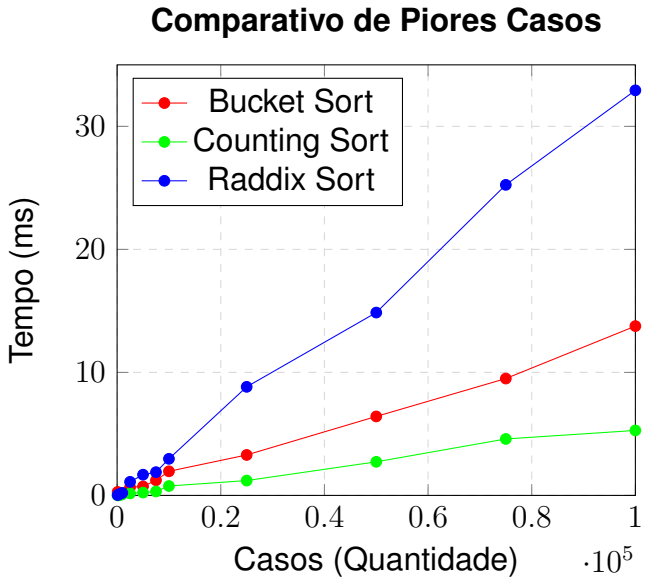
Os melhores casos testes do Bucket Sort, Counting Sort e do Raddix Sort foram plotados conjuntamente no gráfico abaixo.



Tendo em vista essas informações, a comparação revela três eficiências diferentes, sendo a mais veloz a do Counting Sort, a intermediária a do Bucket Sort e a mais lenta a do Raddix Sort. Nesse contexto, os dados do gráfico mostram compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório.

3.4.3 Comparativo dos Piores Casos

Os piores casos testes do Bucket Sort, Counting Sort e do Raddix Sort foram plotados conjuntamente no gráfico abaixo.

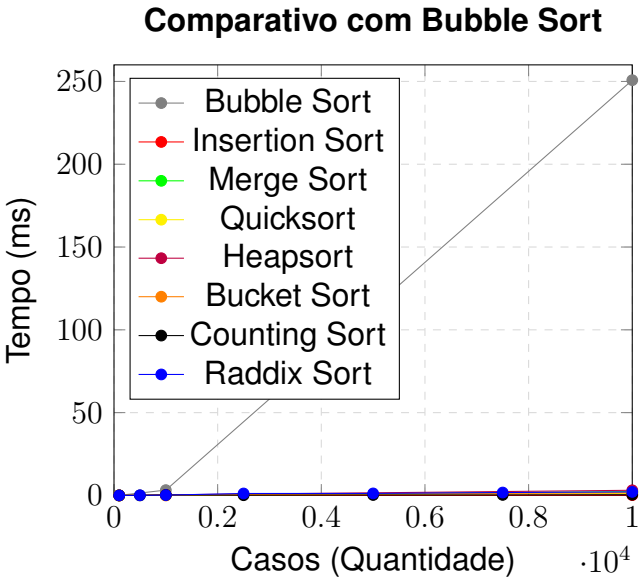


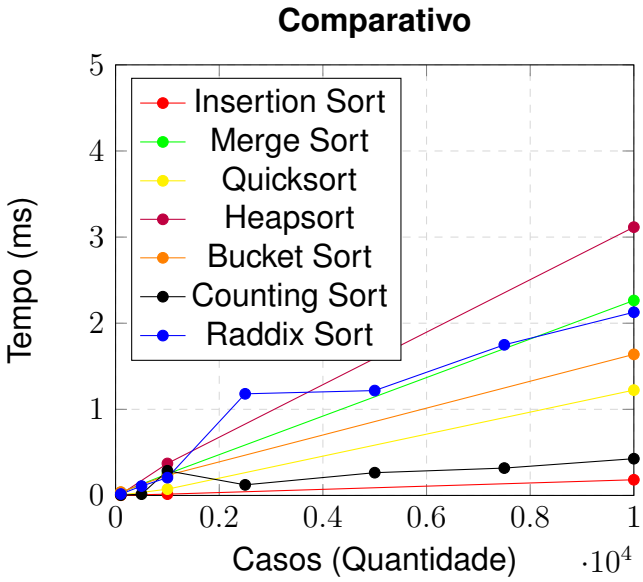
Tendo em vista essas informações, a comparação revela três eficiências diferentes, sendo a mais veloz a do Counting Sort, a intermediária a do Bucket Sort e a mais lenta a do Raddix Sort. Nesse contexto, os dados do gráfico mostram compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório.

3.5 Comparativo: Bubble Sort, Insertion Sort, Merge Sort, Quicksort, Heapsort, Bucket Sort, Counting Sort e Raddix Sort

3.5.1 Comparativo dos Casos Aleatórios

Os casos testes dos melhores casos do Bubble Sort, Insertion Sort, Merge Sort, Quicksort, Heapsort, Bucket Sort, Counting Sort e Raddix Sort foram plotados conjuntamente abaixo.



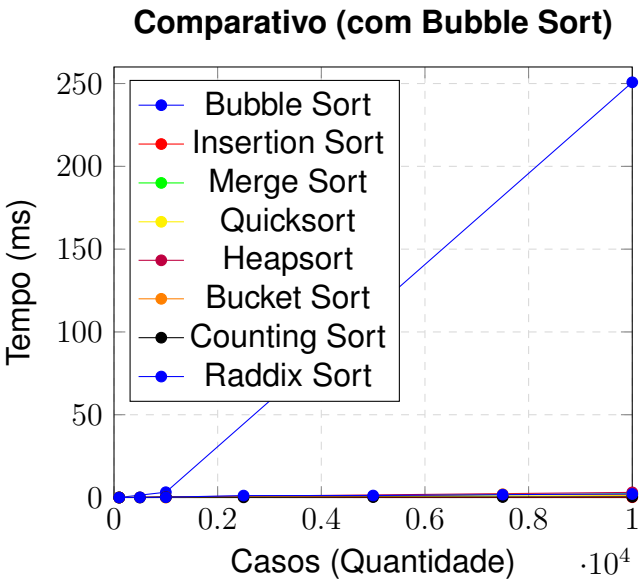


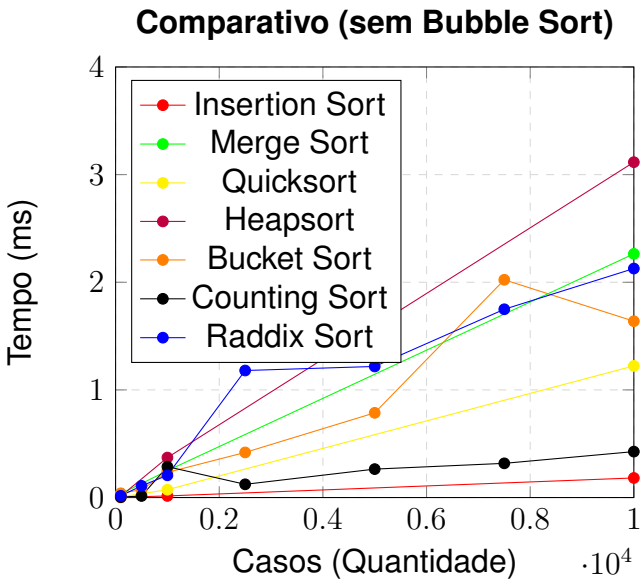
Nesse contexto, os dados do gráfico analisado com os dados dos relatórios anteriores

mostram compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório.

3.5.2 Comparativo dos Melhores Casos

Os casos testes dos melhores casos do Bubble Sort, Insertion Sort, Merge Sort, Quicksort, Heapsort, Bucket Sort, Counting Sort e Raddix Sort foram plotados conjuntamente abaixo.

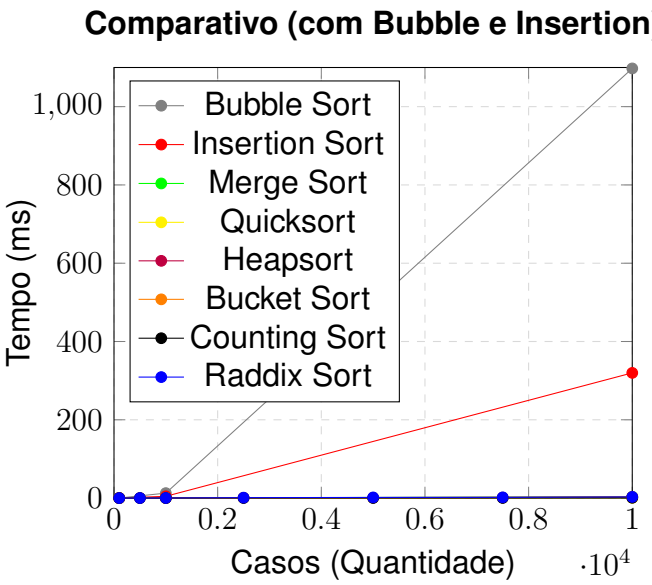


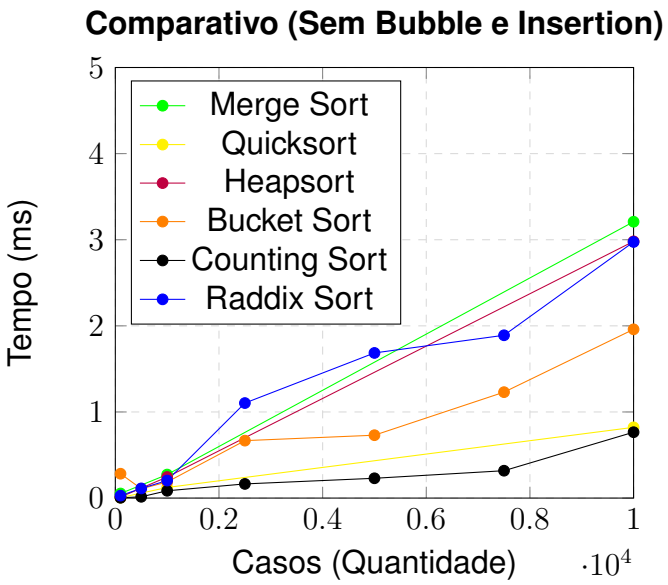


Nesse contexto, os dados do gráfico analisado com os dados dos relatórios anteriores mostram compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório.

3.5.3 Comparativo dos Piores Casos

Os casos testes dos melhores casos do Bubble Sort, Insertion Sort, Merge Sort, Quicksort, Heapsort, Bucket Sort, Counting Sort e Raddix Sort foram plotados conjuntamente abaixo.





Nesse contexto, os dados do gráfico analisado com os dados dos relatórios anteriores mostram compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório.

4 Conclusão

Embora tenha havido pequenas variações de performance em casos específicos, concluiu-se o relatório confirmando as expectativas reunidas inicialmente. Nesse sentido, comprovou-se, por meio de dados empíricos, as informações fornecidas pela literatura especializada e pelas aulas de laboratório. Isto é, mostrou-se, por meio da análise assintótica dos algoritmos de ordenação, que os mecanismos Bucket Sort, Counting Sort e Raddix Sort possuem ordem linear $\Theta(n)$).

5 Referências Bibliográficas

BUCKET SORT. In: GEEKS FOR GEEKS. Disponível em: <https://www.geeksforgeeks.org/bucket-sort-2/?ref=lbp>. Acesso em: 26/11/2021.

CORMEN, Thomas et alii. Algoritmos: Teoria e Prática. Rio de Janeiro: Editora Elsevier, 2002.

COUNTING SORT. In: GEEKS FOR GEEKS. Disponível em: <https://www.geeksforgeeks.org/counting-sort/>. Acesso em: 26/11/2021.

LEMOS, Carlos Filipe de Castro. 01 - Relatório - Eficiência de Algoritmos de Busca e Ordenação. USP, ICMC, Laboratório de Introdução a Ciência da Computação II, São Carlos: 2021.

_____. 02 - Relatório - Eficiência de Algoritmos de Busca e Ordenação. USP, ICMC, Laboratório de Introdução a Ciência da Computação II, São Carlos: 2021.

RADIX SORT. In: WIKIPÉDIA, a enciclopédia livre. Flórida: Wikimedia Foundation, 2020. Disponível em: <https://pt.wikipedia.org/w/index.php?title=Radix_sort&oldid=58054217>. Acesso em: 26/11/2021.