Universidade de São Paulo

Instituto de Ciências Matemáticas e Computação

Análise de algoritmos de ordenação: Counting Sort, Bucket Sort e Radix Sort

Ana Lívia Ruegger Saldanha

5 de janeiro de 2022

Resumo

Este relatório apresenta uma análise teórica e experimental acerca da eficiência de três algoritmos de ordenação: counting sort, bucket sort e radix sort. Com base na bibliografia, cada algoritmo é brevemente debatido a partir de seu funcionamento e implementação em linguagem C, identificando-se suas respectivas complexidades assintóticas (notações Big-O e Big-Omega). Essa análise é complementada com os dados obtidos experimentalmente através de medições temporais feitas também em linguagem C. Na parte experimental, esses três algoritmos também são comparados com outros três: quick sort, heap sort e merge sort.

---

1 Introdução

O problema da ordenação apresentado por Cormen no primeiro capítulo de *Algoritmos: teoria e prática* [1] é frequentemente estudado como introdução aos conceitos de algoritmos e eficiência. Pode ser definido formalmente da seguinte maneira:

- **Entrada:** Uma sequência de n números $< a_1, a_2, ..., a_n >$.
- **Saída:** Uma permutação (reordenação) $< a'_1, a'_2, ..., a'_n >$ da sequência de entrada, tal que $a'_1 \leq a'_2 \leq ... \leq a'_n$.

Para solucioná-lo, podemos utilizar diversos algoritmos, mas, mesmo que todos sejam corretos e resolvam o problema dado, esses algoritmos podem ser muito diferentes em termos de eficiência. Neste relatório, temos por objetivo debater essa questão através da análise e comparação de seis algoritmos de ordenação: counting sort (ordenação por contagem), bucket sort (ordenação por balde), radix sort (ordenação digital), merge sort (ordenação por intercalação), quick sort e heap sort, com enfoque nos três primeiros, sobre os quais apresentaremos uma análise teórica mais aprofundada, visto que os outros já foram estudados em relatórios anteriores[2][3].

Primeiramente, buscamos contextualizar esses três algoritmos (counting, bucket e radix sort), apresentando seu funcionamento básico e implementações em linguagem C, assim como as funções obtidas através da contagem de operações de cada algoritmo e respectivas análises utilizando o conjunto de notações assintóticas Big-O ($\mathcal{O}$) e Big-Omega (Ω).

Em seguida, apresentamos os resultados de um experimentos realizados com os seis algoritmos através de medições temporais. Os experimentos têm por objetivo comprovar as discussões levantadas anteriormente para que possamos, por fim, realizar uma análise abrangente desses algoritmos em diferentes cenários.

2 Metodologia e desenvolvimento

Nos relatórios anteriores, foram estudados algoritmos de ordenação baseados em comparação, isto é, que realizam a ordenação de um conjunto comparando seus valores entre si. Para estes algoritmos, o desempenho alcançado não consegue vencer o limite inferior $n \cdot log(n)$ no pior caso.[1] Neste relatório, nos focaremos em algoritmos de ordenação linear — *counting*, *bucket* e *radix sort* —, que são capazes de vencer essa barreira $n \cdot log(n)$, porém necessitam de memória auxiliar que depende não apenas do tamanho n do conjunto a ser ordenado, mas também da diferença entre os valores da maior e da menor chave presentes nesse conjunto (chamaremos essa amplitude de k).

Para estabelecer um padrão, estudaremos os três algoritmos em suas versões que trabalham com registros[2], adotando o tipo definido abaixo (em linguagem C):

```c
typedef struct registro
{
    int chave;
    // Outros atributos
    // (...)
}   registro_t;
```

Adiante, discutiremos mais a fundo o Counting Sort, o Bucket Sort e o Radix Sort, trazendo também um resumo do que já foi apresentado sobre o Quick Sort, o Heap Sort e o Merge Sort [2][3].

2.1 Counting Sort

O *counting sort* é um algoritmo que ordena um conjunto a partir de chaves inteiras. Sua heurística é baseada na contagem dos elementos que apresentam dada chave, utilizando esse mesmo valor chave para indexar um vetor de tamanho k que armazena a contagem em si (por isso, é preciso trabalhar com inteiros); essa contagem, por sua vez, é utilizada para determinar o índice final de cada elemento no conjunto ordenado.

Pelo processo descrito acima, podemos notar que o *counting sort* necessita, pelo menos, de um vetor auxiliar de tamanho k; sua implementação tradicional, portanto, pertence a $\mathcal{O}(k)$ quanto ao uso de memória. Porém, como poderemos observar adiante, na implementação que utilizamos — que trabalha com registros — precisamos de uma cópia do vetor original; neste caso, o *counting sort* pertence a $\mathcal{O}(n + k)$

A seguir, apresentamos a implementação do *counting sort* com registros, em linguagem C.

```c
void counting_sort(registro_t *vetor, int tamanho)
{
    registro_t *copia = (registro_t *)malloc(tamanho * sizeof(registro_t));

    int max, min;
    max = min = vetor[0].chave;

```

[1] Esta afirmação é provada por Cormen et al. na seção 8.1, *Limites inferiores para ordenação*, do livro *Algoritmos: teoria e prática*[1], levando em consideração que as ordenações por comparação funcionam como **árvores de decisão**.

[2] As implementações utilizadas no experimento e reproduzidas neste relatório estão de acordo com o material apresentado em aula pelo professor Fernando Pereira dos Santos[4]. Já as análises apresentadas foram baseadas tanto nas aulas quanto na bibliografia, especialmente no livro *Algoritmos: teoria e prática* (Cormen et al.)[1].

```
8       for (int i = 0; i < tamanho; i++)
9       {
10          if (vetor[i].chave > max)
11              max = vetor[i].chave;
12          if (vetor[i].chave < min)
13              min = vetor[i].chave;
14          copia[i] = vetor[i];
15      }
16
17      int k = max - min + 1;
18      int *contagem = (int *)calloc(k, sizeof(int));
19
20      for (int i = 0; i < tamanho; i++)
21      {
22          int posicao_chave = vetor[i].chave - min;
23          contagem[posicao_chave]++;
24      }
25
26      int total = 0;
27      for (int i = 0; i < k; i++)
28      {
29          int contagem_anterior = contagem[i];
30          contagem[i] = total;
31          total = total + contagem_anterior;
32      }
33
34      for (int i = 0; i < tamanho; i++)
35      {
36          int posicao_ordenada = contagem[copia[i].chave - min];
37          vetor[posicao_ordenada] = copia[i];
38          contagem[copia[i].chave - min]++;
39      }
40
41      free(contagem);
42      free(copia);
43  }
```

Pela lógica mostrada no código acima, podemos observar que, quanto à complexidade de tempo, o *counting sort* também é diretamente influenciado pelos valores de k e n. Dessa forma, para realizar a análise assintótica, precisamos observar quantas vezes o algoritmo percorre cada vetor de tamanho n ou k; esse processo pode ser facilmente observado nos **laços** presentes no código:

- O primeiro laço, iniciado na linha 8, percorre o vetor original de tamanho n para realizar uma cópia dos dados

e também buscar as chaves máxima e mínima; tem, portanto, complexidade de tempo $\mathcal{O}(n)$.

- Um segundo laço está presente na inicialização do vetor de contagem de tamanho k, sendo implícito na função `calloc` (linha 18); pertence a $\mathcal{O}(k)$.
- O terceiro laço (linha 20) percorre o conjunto inicial de tamanho n realizando a contagem das chaves; pertence a $\mathcal{O}(n)$.
- O quarto laço (linha 27) realiza a contagem acumulada das chaves, ou seja, determina as posições finais no conjunto ordenado; pertence a $\mathcal{O}(k)$.
- Finalmente, o último laço (linha 34) realiza o posicionamento final dos elementos no conjunto ordenado; pertence a $\mathcal{O}(n)$.

O *counting sort*, dessa forma, percorre três vezes um vetor de tamanho n e duas vezes um vetor de tamanho k, resultando na complexidade de tempo $\mathcal{O}(n+k)$. Podemos observar, também, que seu funcionamento dependerá sempre dos valores de n e k, de forma que ele jamais será capaz de ultrapassar o limite inferior de $n + k$; pertence, portanto, a $\Omega(n + k)$.

Outra característica do *counting sort* que devemos destacar é que este se trata de um algoritmo **estável**, ou seja, que preserva, no conjunto final, a ordem inicial dos elementos que possuem a mesma chave. Como veremos adiante, este é um fator essencial quando utilizamos a ordenação por contagem como sub-rotina do *radix sort*.

2.2 Bucket Sort

O *bucketsort*, ou ordenação por balde, funciona a partir da distribuição dos elementos de um conjunto em listas encadeadas chamadas de baldes. Estas listas, então, podem ser ordenadas por outro algoritmo de ordenação, ou recursivamente pelo próprio *bucketsort*; por fim, basta que, percorrendo balde por balde, os elementos sejam reposicionados no conjunto final ordenado. A implementação aqui estudada, no entanto, possui uma particularidade: cada lista, ou balde, está associada a uma chave específica[3] (portanto, não há uma sub-rotina de ordenação das listas).

Além da *struct* registro reproduziada anteriormente, no *bucketsort* utilizamos as seguintes declarações de nó e lista encadeada (balde):

```
1  typedef struct no
2  {
3      registro_t elem;
4      struct no *prox;
5  }   no_t;
6
7  typedef struct balde
8  {
9      no_t *inicio;
10     no_t *fim;
11 }   balde_t;
```

[3] Este tipo específico de *bucketsort* também é conhecido como *pigeonhole sort*.

Declaradas as *structs* acima, a implementação do *bucketsort* em linguagem C é dada da seguinte forma:

```c
void bucket_sort(registro_t *vetor, int tamanho)
{
    int max, min;
    max = min = vetor[0].chave;
    for (int i = 1; i < tamanho; i++)
    {
        if (vetor[i].chave > max)
            max = vetor[i].chave;
        if (vetor[i].chave < min)
            min = vetor[i].chave;
    }

    int k = max - min + 1;
    balde_t *baldes = (balde_t *)calloc(k, sizeof(balde_t));

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

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

        if (baldes[posicao_chave].inicio == NULL)
            baldes[posicao_chave].inicio = novo;
        else
            (baldes[posicao_chave].fim)->prox = novo;
        baldes[posicao_chave].fim = novo;
    }

    int j = 0;
    for (int i = 0; i < k; i++)
    {
        no_t *posicao;
        posicao = baldes[i].inicio;
        while (posicao != NULL)
        {
            vetor[j] = posicao->elem;
            j++;

            no_t *deletar = posicao;
```

```
42              posicao = posicao->prox;
43              baldes[i].inicio = posicao;
44              free(deletar);
45          }
46      }
47
48      free(baldes);
49  }
```

Nessa implementação em específico, a complexidade de tempo do *bucketsort* é similar à do *counting sort* e pode ser compreendida observando os laços presentes no código. O algoritmo percorre duas vezes um vetor de tamanho n: para verificar os valores máximo e mínimo dentre as chaves (linha 5) e para preencher os baldes (linha 16); por fim, para reposicionar os elementos no vetor original, retirando-os dos baldes, o algoritmo também percorre n nós no total (a partir da linha 32). Já o vetor de baldes, de tamanho k, é percorrido duas vezes: quando é inicializado pela função `calloc` (linha 14) e no processo de reposicionamento dos elementos no conjunto ordenado (a partir da linha 32).

A partir dessa análise, podemos afirmar que, assim como o *counting sort*, o *bucketsort* pertence a $\mathcal{O}(n+k)$, e sempre será dependente desses dois valores; dessa forma, esse método também não poderá ultrapassar o limite inferior de $n+k$ — ou seja, o *bucketsort* pertence à $\Omega(n+k)$.

No entanto, não podemos esperar que, na prática, o tempo de ordenação do *bucketsort* seja o mesmo que o do *counting sort*, visto que, ao trabalhar com listas ligadas, esse algoritmo realiza uma quantidade maior de operações (por exemplo, a alocação e a desalocação da memória ocupada pelos nós das listas). Além disso, para uma aplicação vantajosa do *bucketsort*, espera-se que o conjunto de entrada seja uniformemente distribuído, de forma que não tenhamos muitos elementos em um mesmo balde;[4] quando isso ocorre, o desempenho desse algoritmo é prejudicado.

O *bucketsort* é um algoritmo **estável** e também pode ser utilizado como sub-rotina na ordenação digital (*radix sort*).

2.3 Radix Sort

Como já foi adiantado, o *radix sort* utiliza um dos algoritmos acima como sub-rotina. Também se trata de um método de ordenação linear que necessita de memória auxiliar, mas sua implementação é capaz minimizar a complexidade de tempo e de memória em cenários com k elevado.

A heurística do *radix sort* é baseada na ordenação pelos **dígitos** das chaves, do menos significativo para o mais significativo; ou seja, um processo de ordenação estável — seja ele o *counting sort* ou o *bucketsort* — será executado uma quantidade d de vezes (uma vez por dígito), resultando, ao fim do processo, na ordenação completa do conjunto. O que chamamos de dígito será definido de acordo com uma base b, um valor previamente fixado que, a cada execução da sub-rotina, ocupará o lugar que no *bucket* e no *counting sort* era ocupado por k, definindo o tamanho do vetor de contagem.

Aqui, abordaremos uma implementação do *radix sort* que utiliza o *counting sort* como sub-rotina e adota a base $b = 256$ (1 byte); essa implementação, em linguagem C, é reproduzida abaixo:

```
1  void radix_sort(registro_t *vetor, int tamanho)
2  {
```

[4] Segundo Cormen et al., na seção 8.4 de *Algoritmos: teoria e prática*[1].

```
3       int contagem[256] = {0};
4       int acumulada[256];
5
6       registro_t *copia = (registro_t *)malloc(tamanho * sizeof(registro_t));
7
8       int i, shift;
9       for (shift = 0; shift <= 24; shift += 8)
10      {
11          for (i = 0; i < tamanho; i++)
12          {
13              short k = (vetor[i].chave >> shift) & 255;
14              contagem[k]++;
15              copia[i] = vetor[i];
16          }
17
18          acumulada[0] = 0;
19          for (i = 1; i < 256; i++)
20          {
21              acumulada[i] = acumulada[i - 1] + contagem[i - 1];
22              contagem[i - 1] = 0;
23          }
24
25          for (i = 0; i < tamanho; i++)
26          {
27              short k = (copia[i].chave >> shift) & 255;
28              vetor[acumulada[k]] = copia[i];
29              acumulada[k]++;
30          }
31      }
32
33      free(copia);
34  }
```

Como pode ser observado no código acima, o laço iniciado na linha 9, utilizando de operações de *bit shift*, é responsável por executar o *counting sort* uma vez para cada dígito. Como sabemos que a ordenação por contagem é $\mathcal{O}(n+k)$, poderíamos afirmar que o *radix sort* pertence a $\mathcal{O}(d \cdot (n+k))$; mas, nesse caso, sendo k o valor previamente fixado na base b, concluímos que o *radix sort* pertence, na verdade, a $\mathcal{O}(d \cdot (n+b))$. Por outro lado, o algoritmo também não será capaz de vencer o limite inferior de $(d \cdot (n+b))$, sendo, assim, $\Omega((d \cdot (n+b))$.

Para a implementação estudada, sabendo que $b = 256$ e $d = 4$ (pois todas as chaves são inteiros de 4 bytes), temos que este algoritmo é $\mathcal{O}(4 \cdot (n+256))$. Notemos ainda que, como a base fixa o tamanho do vetor de contagem, o uso de memória no *radix sort* também é controlado (e pode ser manipulado) pelo valor de b. Quanto ao uso de memória, portanto, o *radix sort* pertence a $\mathcal{O}(n+b)$.

2.4 Comparação com Quick Sort, Heap Sort e Merge Sort

Para fins de comparação, a Tabela 1 apresenta as informações essenciais acerca da complexidade do *quick*, do *heap* e do *mergesort*, como demonstrado nos relatórios anteriores[2][3].

Algoritmo	$\mathcal{O}$	Ω	Memória auxiliar	Melhor caso	Pior caso
Quick Sort	$\mathcal{O}(n^2)$	$\Omega(n\ logn)$	$\mathcal{O}(1)$	Pivô no centro	Pivô na extremidade (vetor ordenado)
Heap sort	$\mathcal{O}(n\ logn)$	$\Omega(n\ logn)$	$\mathcal{O}(1)$	-	-
Merge Sort	$\mathcal{O}(n\ logn)$	$\Omega(n\ logn)$	$\mathcal{O}(n)$	-	-

Tabela 1: Resumo acerca das complexidades de tempo e espaço dos algoritmos de ordenação Quick Sort, Heap Sort e Merge Sort.

3 Resultados

A seguir, apresentamos os resultados de três conjuntos de testes, realizados a fim de estudar, na prática, a eficiência dos algoritmos apresentados na seção anterior deste relatório.

3.1 Vetores gerados aleatoriamente

O primeiro experimento realizado teve como objetivo analisar o desempenho dos algoritmos *counting*, *bucket* e *radix sort* com vetores de entrada cada vez maiores. Destacamos algumas escolhas que fizemos na implementação do experimento:

- Executamos três testes com vetores de registros cujos valores das chaves foram gerados aleatoriamente, cada um adotando uma amplitude k diferente: valores entre 0 e n, entre 0 e $5n$, e entre 0 e $10n$, sendo n o tamanho da entrada.
- Para cada tamanho de vetor, realizamos 10 iterações/medições; os valores apresentados nos gráficos a seguir referem-se à média do tempo de execução nessas 10 iterações.
- Cada teste dos algoritmos foi realizado em uma única execução do programa, com exatamente os mesmos vetores, gerando um arquivo .csv para cada algoritmo. Os dados contidos nesses arquivos foram utilizados para plotar o gráficos reproduzidos a seguir.

Como era esperado, por conta do uso de listas ligadas e consequente maior número de operações, o *bucketsort* mostrou-se, nos três testes, o mais lento dentre os algoritmos de ordenação lineares.

Também como previsto, a diferença de eficiência entre o *counting sort* e o *radix sort* é alterada conforme aumentamos a amplitude de valores k. No teste com valores entre 0 e n, apresentado no gráfico da Figura 1, o *counting sort* mostrou-se mais rápido que o *radix*; com valores entre 0 e $5n$, os dois algoritmos apresentaram desempenho muito similar (Figura 2); finalmente, com valores entre 0 e $10n$, o *radix sort* mostrou-se o mais eficiente dos três algoritmos (Figura 3).

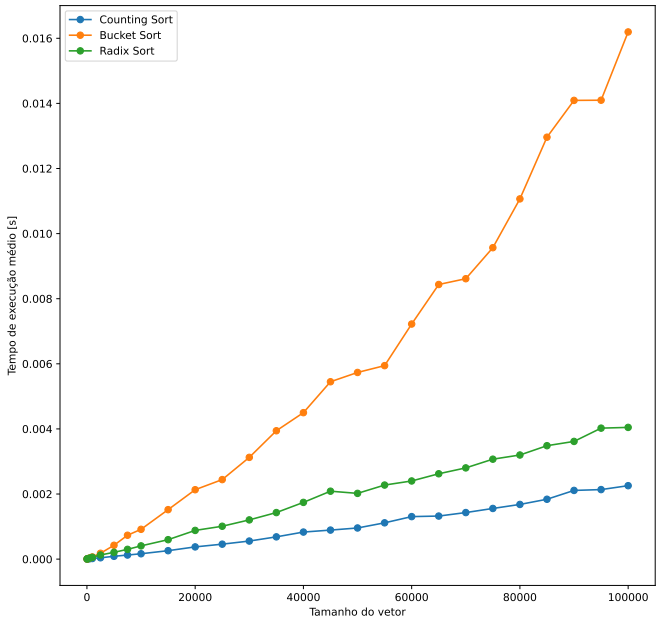

Figura 1: Resultado das medições temporais para cada algoritmo, ordenando com chaves de valores entre 0 e n; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

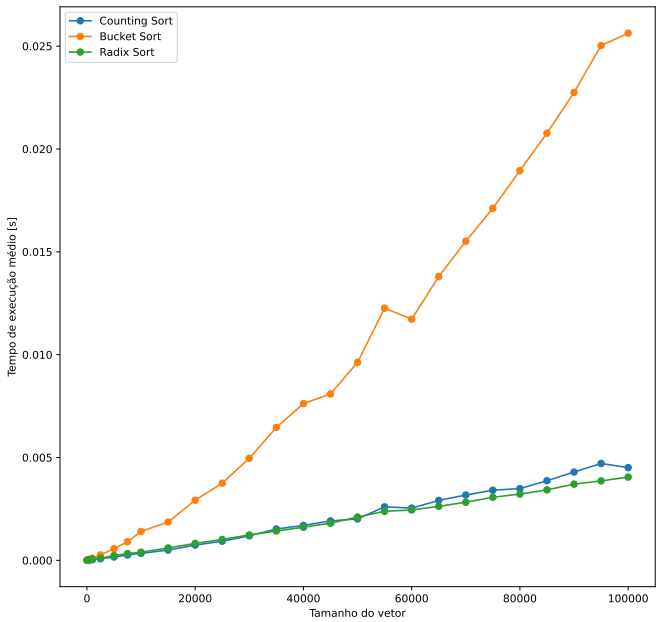

Figura 2: Resultado das medições temporais para cada algoritmo, ordenando com chaves de valores entre 0 e $5n$; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

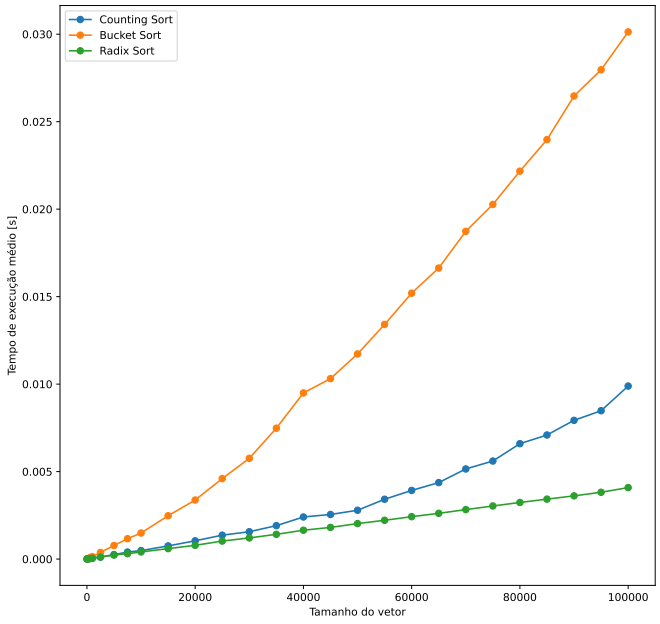

Figura 3: Resultado das medições temporais para cada algoritmo, ordenando com chaves de valores entre 0 e $10n$; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

3.2 Melhor e pior caso

Mantendo o padrão dos experimentos apresentados nos relatórios anteriores[2][3], realizamos testes com vetores de mesmo tamanho ($n = 1000$), porém de diferentes tipos: gerados aleatoriamente, ordenados e inversamente ordenados.

Tipo de vetor (tamanho $n = 100$)	Tempo de execução médio [s]		
	Counting Sort	**Bucket Sort**	**Radix Sort**
Gerado aleatoriamente	0.000019	0.000065	0.000057
Ordenado	0.000020	0.000054	0.000055
Inversamente ordenado	0.000019	0.000054	0.000053

Tabela 2: Resultado das medições temporais para ordenação de diferentes tipos de vetor, todos com o mesmo tamanho; a média de tempo de execução foi tirada a partir de 10 iterações para cada tipo de vetor.

Os resultados mostrados na Tabela 2 confirmam que, de fato, como estes não se tratam de algoritmos baseados em comparação, a ordenação prévia dos elementos do conjunto não altera sua eficiência. No entanto, podemos notar que o *bucketsort* apresentou um desempenho melhor para os vetores ordenados; essa diferença era esperada e não se deve à ordenação em si, mas ao fato de que, nesses vetores, as chaves foram distribuídas uniformemente.

Como visto na seção 2 deste relatório, os algoritmos de ordenação lineares tem seu desempenho diretamente afetado pela amplitude de valores k apresentada nas chaves. Dessa forma, para analisar o que chamamos de **pior caso** e **melhor caso**, devemos observar o comportamento de cada um desses algoritmos diante da variação de k. Utilizaremos, para tal, os dados do experimento já descrito no item 3.1 — mas, desta vez, apresentados através de gráficos que comparam o desempenho de um mesmo algoritmo nos três cenários diferentes (Figuras 4, 5 e 6).

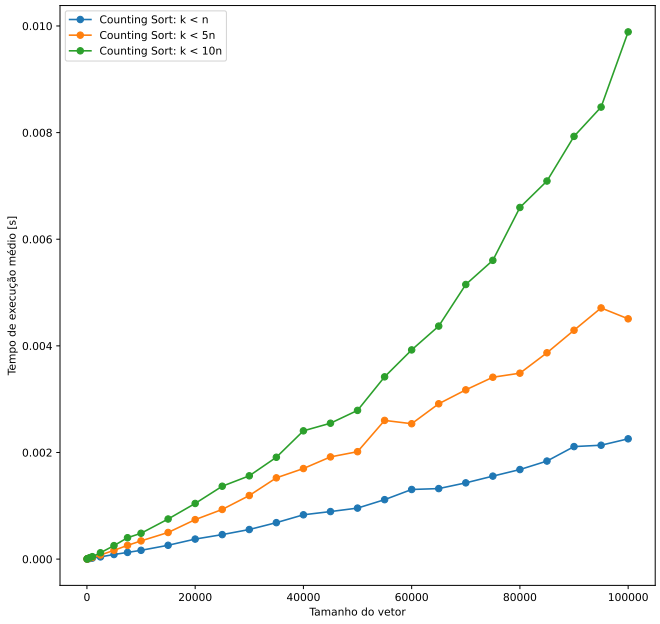

Figura 4: Resultado das medições temporais para o Counting Sort, ordenando valores com amplitude $k < n$, $k < 5n$ e $k < 10n$, respectivamente; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

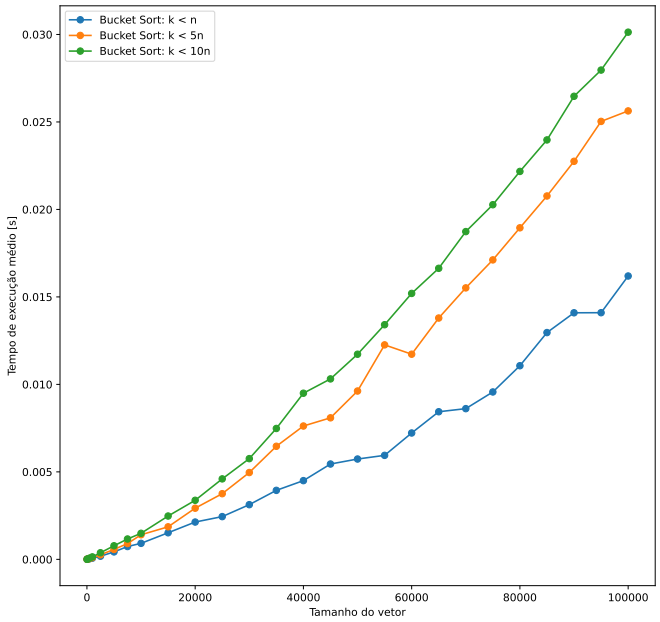

Figura 5: Resultado das medições temporais para o Bucket Sort, ordenando valores com amplitude $k < n$, $k < 5n$ e $k < 10n$, respectivamente; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

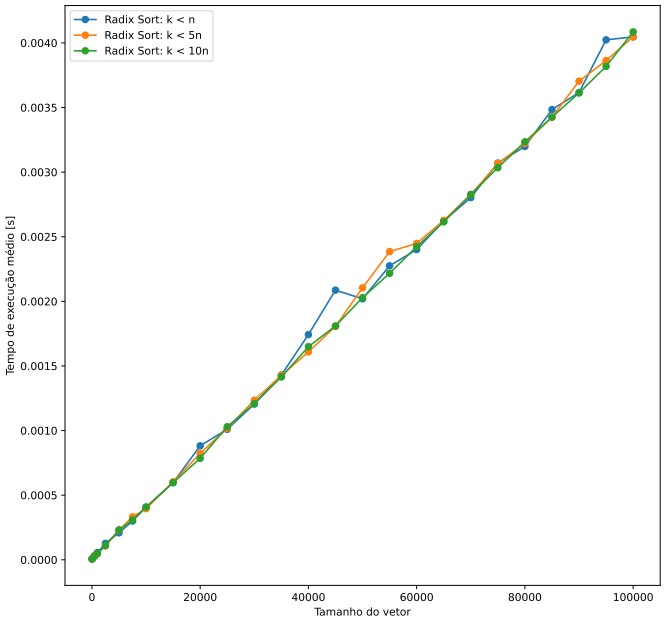

Figura 6: Resultado das medições temporais para o Radix Sort, ordenando valores com amplitude $k < n$, $k < 5n$ e $k < 10n$, respectivamente; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

No gráfico da Figura 4, é possível observar que o desempenho do *counting sort* de fato piora gradualmente com o aumento da amplitude das chaves; o mesmo ocorre com o *bucketsort* (Figura 5). Já a eficiência do *radix sort* (Figura 6) não foi afetada pelo aumento de k, visto que sua complexidade de tempo é $\mathcal{O}(d \cdot (n + b))$, ou seja, é afetada não por k, mas pela base e pelo número de dígitos das chaves, que permaneceram os mesmos.

3.3 Comparação com Quick Sort, Heap Sort e Merge Sort

Por fim, o último experimento teve como objetivo comparar os métodos de ordenação lineares com o *quicksort*, o *heapsort* e o *mergesort*, algoritmos de complexidade $\mathcal{O}(n \cdot log(n))$ estudados anteriormente[2][3]. Foram realizados dois testes, com conjuntos de entrada cujas chaves foram geradas aleatoriamente respeitando diferentes amplitudes de valores ($k \leq 10.000$ e $k \leq 100.000$). Os outros critérios para cálculo do tempo médio de execução foram os mesmos adotados no experimento descrito no item 3.1 deste relatório.

Na Figura 7, podemos observar que, para $k \leq 10.000$, os três métodos lineares mostraram-se mais rápidos que os de complexidade log-linear, como esperado. Quando elevamos a amplitude de valores entre as chaves para $k \leq 100.000$ (Figura 8), os algoritmos pertencentes a $\mathcal{O}(n \cdot log(n))$ mantiveram seu desempenho, enquanto o *counting* e *bucketsort* perderam vantagem; esta perda de vantagem foi pouco expressiva no caso do *counting sort*, mas o *bucketsort* mostrou-se, neste caso, também mais lento que o *quicksort*.

Na prática, o *quicksort*, mesmo apresentando complexidade log-linear, ainda pode ser considerado muito rápido na ordenação de conjuntos de tamanhos $n \leq 100.000$, não ultrapassando o tempo médio de execução de 0,015 segundo.

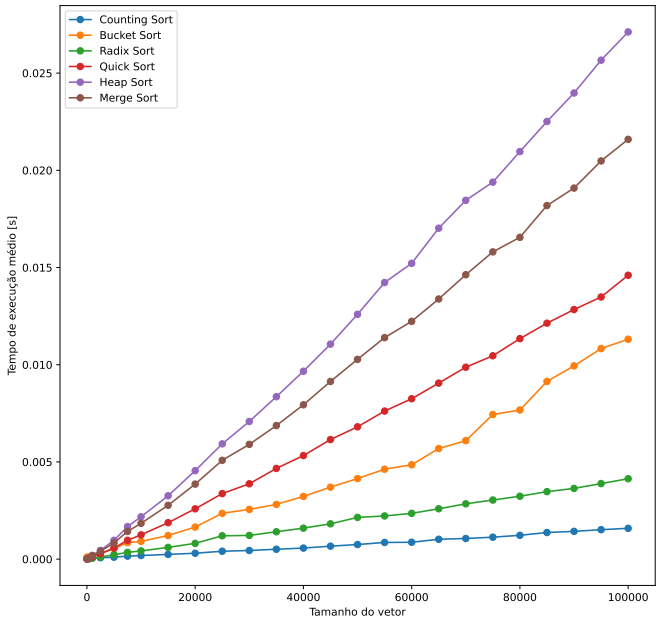

Figura 7: Resultado das medições temporais para cada algoritmo, ordenando com chaves de valores entre 0 e 10.000; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

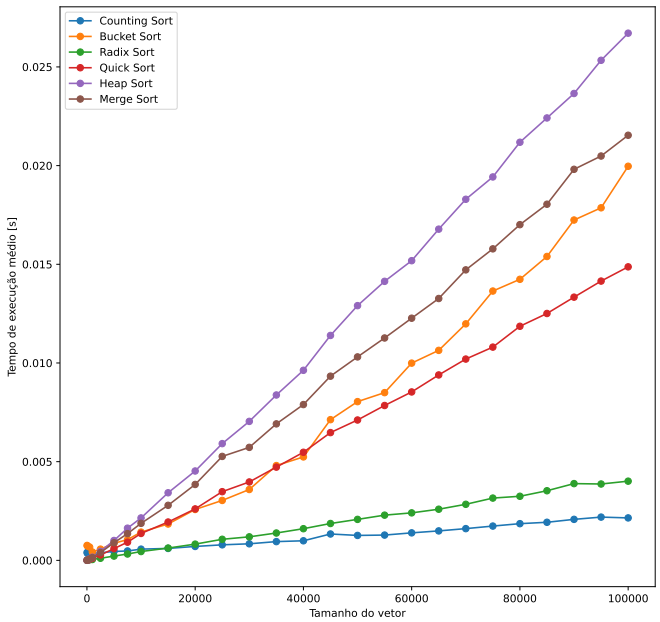

Figura 8: Resultado das medições temporais para cada algoritmo, ordenando com chaves de valores entre 0 e 100.000; a média de tempo de execução foi tirada a partir de 10 iterações para cada tamanho de vetor.

Por fim, é preciso reforçar que, apesar de estes apresentarem menores tempos médios de execução, os algoritmos de ordenação lineares têm limitações, necessitando da alocação de memória auxiliar e não podendo trabalhar com chaves não inteiras. Para cenários em que a memória do sistema é escassa, ou quando os valores das chaves são reais, ainda é recomendado o uso de um algoritmo baseado em comparação, de complexidade log-linear.

Ao escolher o melhor algoritmo para um dado problema, portanto, deve-se sempre levar em conta diversos fatores: o tamanho do conjunto a ser ordenado, a amplitude de valores das chaves nesse conjunto, a natureza de seus elementos e a memória disponível no sistema.

4 Conclusão

Os métodos de ordenação lineares, para superarem a eficiência de $\mathcal{O}(n \cdot log(n))$, tem um custo adicional de alocação de memória auxiliar. Em suma, os algoritmos estudados apresentaram as seguintes características:

- Para vetores de tamanho n com amplitude de valores $k < 5n$, aproximadamente[5], o *counting sort* é o mais rápido dentre todos os algoritmos estudados. Quanto ao uso de memória, pertence a $\mathcal{O}(n + k)$.
- O *radix sort* mostrou-se mais eficiente que o anterior quando a amplitude de valores k ultrapassa $5n$, aproximadamente, sendo a melhor opção para cenários com chaves esparsas. Pode utilizar menos memória auxiliar que o *counting* e o *bucketsort*, pois, em sua sub-rotina, k será sempre igual ao valor da base.
- Para a implementação utilizada, o *bucketsort* é o mais lento dos três; no entanto, é um algoritmo versátil que pode ser otimizado e adaptado. Quanto ao uso de memória, também pertence a $\mathcal{O}(n + k)$.

Sobre os algoritmos baseados em comparação — limitados inferiormente por $n \cdot log(n)$ — concluímos que ainda são a escolha recomendada para os cenários em que a memória do sistema é escassa, ou quando as chaves para ordenação não são discretas; dentre eles, o *quicksort* possui o menor tempo médio de execução.

Referências

[1] Thomas H. Cormen et al. *Algoritmos: teoria e prática*. Rio de janeiro: Elsevier, 2012.

[2] Ana Lívia Ruegger Saldanha. Análise de algoritmos de ordenação: Bubble Sort, Insertion Sort e Merge Sort. Novembro de 2021.

[3] Ana Lívia Ruegger Saldanha. Análise de algoritmos de ordenação: Quick Sort e Heap Sort. Novembro de 2021.

[4] Fernando Pereira dos Santos. Material da disciplina Laboratório de Introdução à Ciência da Computação II (SCC0220). ICMC-USP, 2º semestre de 2021.

[5] Fernando Pereira dos Santos. Material da disciplina Introdução à Ciência da Computação II (SCC0201). ICMC-USP, 2º semestre de 2021.

[6] Moacir Antonelli Ponti. Material da disciplina Introdução à Ciência da Computação II (SCC0201). ICMC-USP, 2º semestre de 2021.

[5] Este foi o valor obtido experimentalmente.