Universidade de São Paulo

Instituto de Ciências Matemáticas e Computação

Relatório 3 de Laboratório de Introdução à Ciências da Computação 2

Vítor Amorim Fróis

27 de dezembro de 2021

Resumo

Durante o último módulo da matéria de Laboratório de ICC2, foram estudados métodos de ordenação que não utilizam comparação: Counting, Bucket e Radix, obtendo assim resultados melhores em diversas situações. Apesar disso, existem limitações para esses algoritmos, que nem sempre funcionam tão bem.
Esse relatório busca analisar isso de modo a detalhar quando cada método pode ou não ser útil, além de comparar o que foi estudado ao longo do semestre.

---

1 Introdução

Ao longo do último semestre, foram estudados diversos métodos de ordenação, alguns com complexidade melhor ou pior de acordo com as regras da notação assintótica.
É fato que dependendo da situação, cada algoritmo pode performar melhor ou pior. Até agora, entretanto, nenhum dos métodos estudados ultrapassa a barreira de $\Theta(n) = n \log n$. Entretanto, ao olhar o problema por outra forma é possível melhorar os métodos. Esse relatório possui como objetivo analisar tais métodos, trazendo suas análises assintóticas, os melhores e piores casos, além de gráficos comparativos com o que foi estudado nos últimos relatórios.

2 Metodologia e desenvolvimento

2.1 A barreira logarítmica

Ao longo dos últimos módulos foram estudados métodos que se utilizavam de comparações para ordenar. Tais algoritmos possuem um limite inferior que pode ser demonstrado a partir de uma árvore de decisões binária. [2]
Existem $n!$ resultados possíveis para um algoritmo de ordenação, isto é, a permutação de todos os elementos do vetor. Uma árvore de decisão do modelo que usa somente comparações deve possuir $n!$ folhas, e portanto uma altura de no mínimo $\Omega(log(n!))$. É possível aplicar a aproximação de Stirling para encontrar que $n! = \sqrt{2\pi n}(\frac{n}{e})^e$ ou considerar que $n! > (n/2)^{n/2}$. Assim:

$$\log(n!) = \log(n/2)^{n/2}$$

$$\log(n!) = \frac{n}{2}\log(n/2)$$

$$\log(n!) = n\log(n)$$

Isso nos leva a uma complexidade mínima de $\Theta(n) = n\log(n)$ [1]. Entretanto, ao quebrar a barreira das comparações, é possível melhorar os algoritmos.

2.2 Counting Sort

Como o próprio nome acusa, o Counting Sort é um algoritmo que se vale da contagem a fim de ordenar sequências de números. Para isso, é necessário um vetor auxiliar de tamanho $k = \max - \min$, que incrementa o valor de das chaves de acordo com os índices do original. Logo, é necessário $O(n)$ para inserir no vetor auxiliar e $O(k)$ ordenar no final.

$$\text{Complexidade de tempo: } O(n+k)$$

$$\text{Complexidade de memória: } O(k)$$

Espera-se que o Counting Sort performe bem com números inteiros e chaves em pequenos intervalos, dado que para um k muito grande, a complexidade do algoritmo cresce muito. Diferente dos métodos estudados nos módulos 1 e 2, a taxa de desordem do vetor não importa.

Para o código a seguir e dos próximos métodos, considere que a representa operações aritméticas, b atribuições, e c comparações.

```c
void countingsort(int* v, int n) {
  // 0 - encontrar chaves min e max
  int final[n];
  int max, min, i;
  max = min = v[0];
  for (int i=1; i < n; i++) {
    if (v[i] > max) max = v[i]; // c+b
    if (v[i] < min) min = v[i]; // c+b
  }
  //N(2b+2c)

  // 1 - cria vetor de contagem
  int k = max-min+1;
  int *contagem = (int*) calloc(k, sizeof(int));
  // Kb

  // 2 - conta a frequencia de cada chave
  for (i = 0; i < n; i++)
    contagem[v[i]-min]++;  // a + a
  // 2aN

  contagem[0] = 0;
  for(i = 1; i < k; i++)
    contagem[i] += contagem[i-1]; //a + b
  //K(a+b)

  // 3 - recria o vetor usando as frequencias para cada chave no vetor de contagem
  for(i = 0; i < n; i++)
    final[contagem[v[i]-min]++] = v[i]; //a+a+b
  //(2a+b)

  for(i = 0; i < n; i++)
        v[i] = final[i]; //b
  //Nb

  // total = 2cN+2bN+Kb+2aN+aK+bK+2aN+bN+bN
  //       = 4Na + 4Nb + 2Nc + Ka + 2Kb
  //       = 10N + 3K
  free(contagem);
}
```

2.3 Bucket Sort

Ainda que a implementação do Counting Sort simples seja instável e seja possível resolver isso através do uso de registros, outro método existe para solucionar tal problema. Através do uso de "baldes", o Bucket Sort realiza ordenação por contagem abaixo do limite anteriormente conhecido.

De forma similar ao Counting, é necessário alocar um vetor de listas ligadas de tamanho $k = \max - \min$, que representarão os baldes. Cada balde pode armazenar um valor ou intervalo. [1] Com complexidade $O(n)$ os elementos são inseridos no vetor. Então, basta esvaziar os baldes, ordenados, a partir da cabeça, inserindo no vetor original $(k + n)$.

$$\text{Totalizando: } n + k + n + k \rightarrow 2(n + k)$$

$$O(n + k) \text{ em tempo}$$

Em termos de memória gasta, porém, devido ao uso de ponteiros nas listas ligadas, existe uma constante ϵ [3], tal que a complexidade de espaço é

$$O(n + k + \epsilon)$$

Isso significa que o algoritmo possui complexidade muito semelhante ao Counting Sort, porém usando mais memória auxiliar com o objetivo de manter a estabilidade.

```c
void bucketsort(int *v, int n) {
  // 1- encontra min e max
  int max, min, i;
  max = min = v[0];
  for (int i=1; i < n; i++) {
    if (v[i] > max) max = v[i]; // c+b
    if (v[i] < min) min = v[i]; // c+b
  }
  // (2c+2b)n

  // 2- criar um vetor auxiliar contendo listas (buckets)
  //    cada bucket possui um ponteiro para o inicio e outro
  //    para o fim da lista
  int k = max-min+1;
  Bucket *B = (Bucket *) calloc(k, sizeof(Bucket));
  for(int i = 0; i < k; i++)
    B[i].begin = NULL;
  // Kb

  // 3 - preenche os buckets com as chaves do vetor
  //     de entrada
  for (i = 0; i < n; i++) { //(c+a)
    int key = v[i]; //b
    Node *newnode = malloc(sizeof(Node));
    newnode->elem = v[i]; //b
    newnode->next = NULL; //b
    //3a

    if (B[key].begin == NULL) B[key].begin = newnode; //b
    else (B[key].end)->next = newnode; //b

    B[key].end = newnode; //b
    //3b+c+2b+a+c
  }
  //n(a+5b+2c)

  // 4 - percorre cada bucket, removendo os elementos do inicio da fila e inserindo na posicao correta
  int j; // percorre buckets
  i = 0; // percorre vetor de entrada //b
  for (j = 0; j < k; j++) { //a+c
    Node *pos;
```

[1]Ao receber um intervalo, é possível, de acordo com a aplicação, aplicar outro método de ordenação que funcione bem com pequenos vetores ou intervalos para ordenar cada Bucket.

```
    pos = B[j].begin; //b
    while (pos != NULL) { //c
      v[i] = pos->elem; //b
      i++; //a+b
      Node *del = pos; //b
      pos = pos->next; //b
      B[j].begin = pos; //b
      free(del);
    }
  }
  //(n+K)(a+5b+2c)

  //total = 2Nc + 2Nb + Kb + Na + 5Nb + 2Nc + Na + 5Nb + 2Nc + Ka + 5Kb + 2Kc
  //     = 6Nc + 12Nb + 6Kb + 2Na + Ka + 2Kc
  //     = n(2a+12b+6c) + K(a+6b+2c)
  //     = n + K
  free(B);
```

2.4 Radix Sort

Essencialmente, os algoritmos vistos até agora são muito bons, mas ainda é possível fazer melhor. Para um k muito grande, Counting e Bucket não são suficientes. Um método de ordenação mais enxuto é o Radix Sort. A ideia do algoritmo é dividir cada número em potências da base utilizada e ordenar do menos significativo para o mais significativo com ajuda de algum dos dois métodos anteriores para alcançar complexidade de tempo linear.

Utilizando Counting Sort: Para um número qualquer na base b, o número de dígitos máximo d pode ser representado por $\log_b c + 1$ e a complexidade será $d \times$ Counting Sort. Como todos os algarismos estarão entre 0 e b, a complexidade do counting será $O(n + b)$.

$$O((n + b) \times d) \text{ ou}$$

$$O((n + b) \times (\log_b c + 1))$$

Aqui, é importante encontrar uma boa base para diminuir a complexidade de tempo do algoritmo. Ao resolver, encontra-se $b = n$.

$$O((n + n) \times (\log_n c + 1)) \rightarrow O(n \log_n c)$$

Se $c \leqslant n^p$, o algoritmo se torna $O(np)$. [1] Para o Radix funcionar, é necessário fazer algumas modificações no algoritmo de subrotina.

```
void counting(int* original, int n, int exp, int base) {
    //Nao e preciso calcular min e max, pois min sera 0 e max o exp
  // 1 - cria vetor de contagem
  int contagem[base];
    for(int i = 0; i < base; i++) contagem[i] = 0; //
    int final[n];
  // Kb

  // 2 - conta a frequencia de cada chave
  int i;
  for (i = 0; i < n; i++)
    contagem[(original[i]/exp)%base]++;    // 4a
  // 4aN
    print(contagem, base);

    for(i = 1; i < base; i++)
        contagem[i] += contagem[i-1]; //K(a+b)

  // 3 - recria o vetor usando as frequencias para cada chave no vetor de contagem
    for(i = n-1; i >= 0; i--)
        final[--contagem[(original[i]/exp)%base]] = original[i]; //b + a+a+a
    //N(3a+b)

  for(i = 0; i < n; i++)
        original[i] = final[i];  //Na
```

```

    //Total = Kb + 4Na + Ka + Kb + N3a + Nb
    //      = N(7a+b) + K(a+2b)
    //      = N + K
}
```

Então, simples chamadas do método ordenam um vetor.

```
void radixsort(int v[], int n){
    int max = v[0];
  for (int i=1; i < n; i++)
    if (v[i] > max) max = v[i]; // c+b
    //n(c+b)
    int base = n;

    for (int exp = 1; max / exp > 0; exp *= base) //log_n(c)
        counting(v, n, exp, base); //counting
    //log_n(c) * (N+K)
}
```

Veja que a complexidade ainda é logarítmica. Espera-se que com a escolha correta de base, o algoritmo se torne linear.

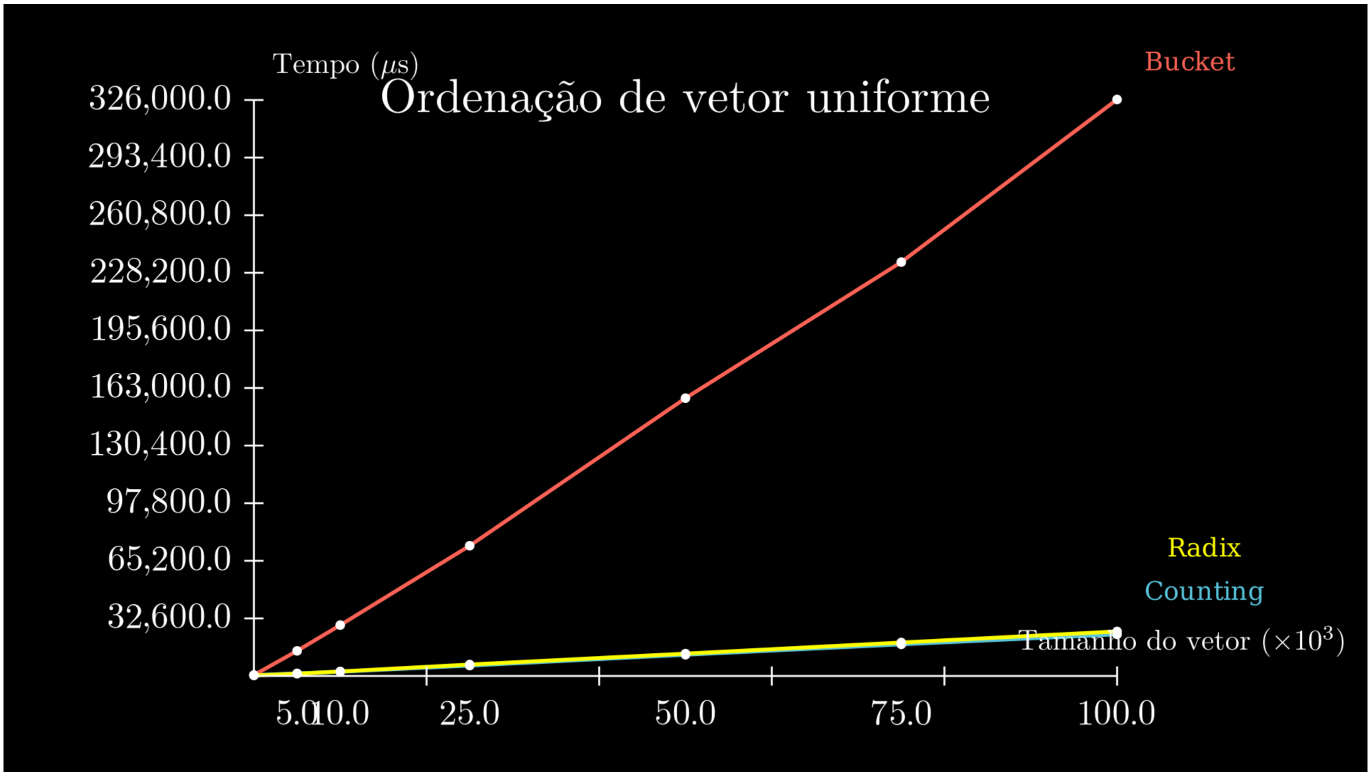
3 Resultados

Para a seção de resultados, é importante repetir as mesmas experiências que foram realizadas com outros métodos afim de fazer boas comparações. Portanto, houveram testes de ordenação com vetores de 5000 a 100000 elementos. Todos os vetores utilizados foram aleatórios, pois a falta de comparação nos algoritmos faz com que ordenação prévia não seja um fator importante na análise.

Entretanto, como destacado na parte de Metodologia, a complexidade de tempo depende do tamanho máximo do vetor k. Assim, para comparação desses 3 métodos, haverão 3 gráficos inicialmente, gerados a partir de vetores uniforme, denso e esparso.

3.1 Vetor Uniforme

No vetor uniforme, os elementos são distribuídos de maneira uniforme pelo vetor. Ou seja, $k = n$, e consequentemente, $O(n + k) \to O(2n)$. É esperado que todos vetores sejam lineares, incluindo o Radix Sort que conta com a escolha adequada de n, descrita anteriormente, para alcançar complexidade $O(n)$.



De fato, os vetores cumprem a linearidade, contudo, o Bucket Sort performa muito pior que os outros. Isso acontece devido as constantes excluídas na análise assintótica, fator que interferiu em análises passadas também. Além disso, é possível perceber que Counting e Radix ficam bem próximos. E isso se deve ao fato de que para vetores uniformes, o Radix é apenas um Counting Sort.

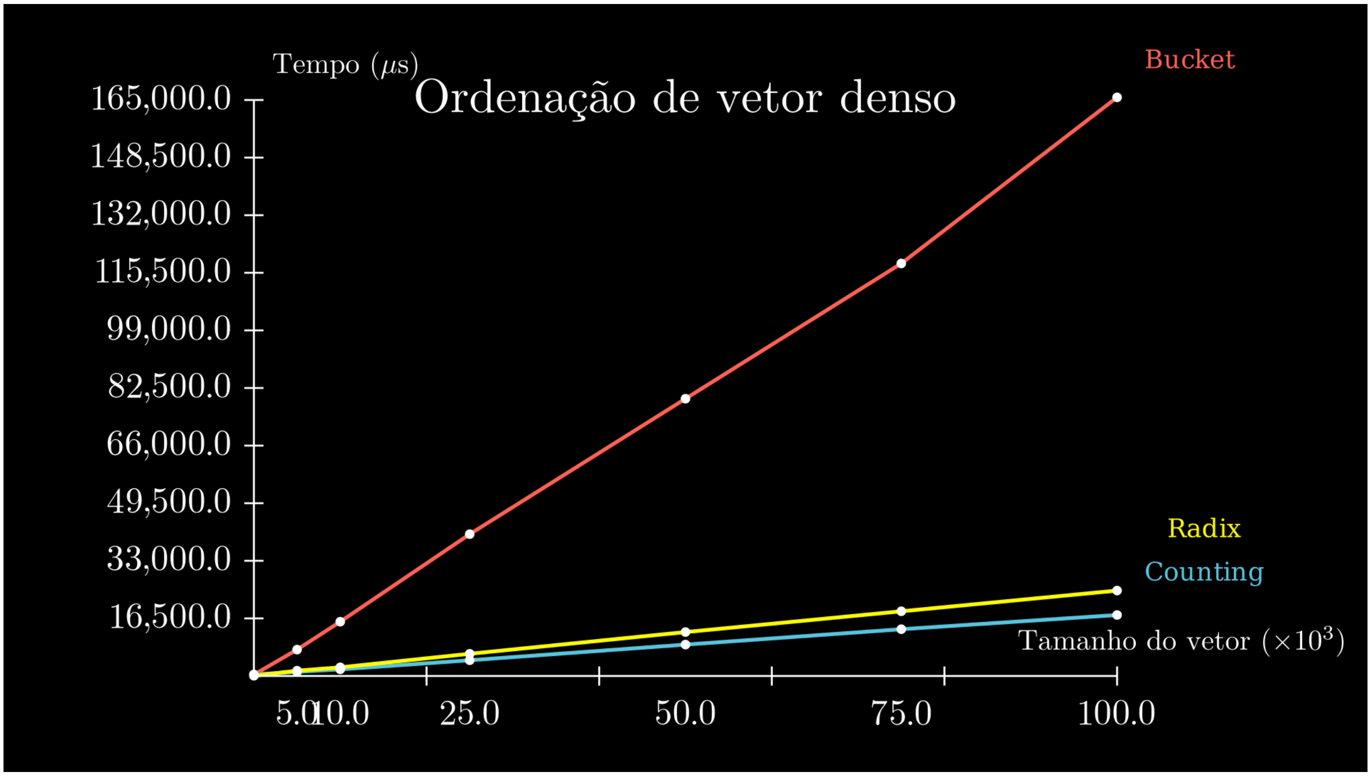
3.2 Vetor Denso

Já no vetor denso, os elementos possuem tamanho máximo de $\frac{1}{100}$ do número de elementos. Isso significa que há diversas repetições, cenário em que os algoritmos analisados melhor devem atuar, pois considerando que a complexidade média é $O(n + k)$ e quando $k = \frac{n}{100}$,

$$O(n + \frac{n}{100}) < O(n + n).\ \text{Veja 3.1}$$

$$\text{A fração é desprezível: } O(n) < O(2n).$$

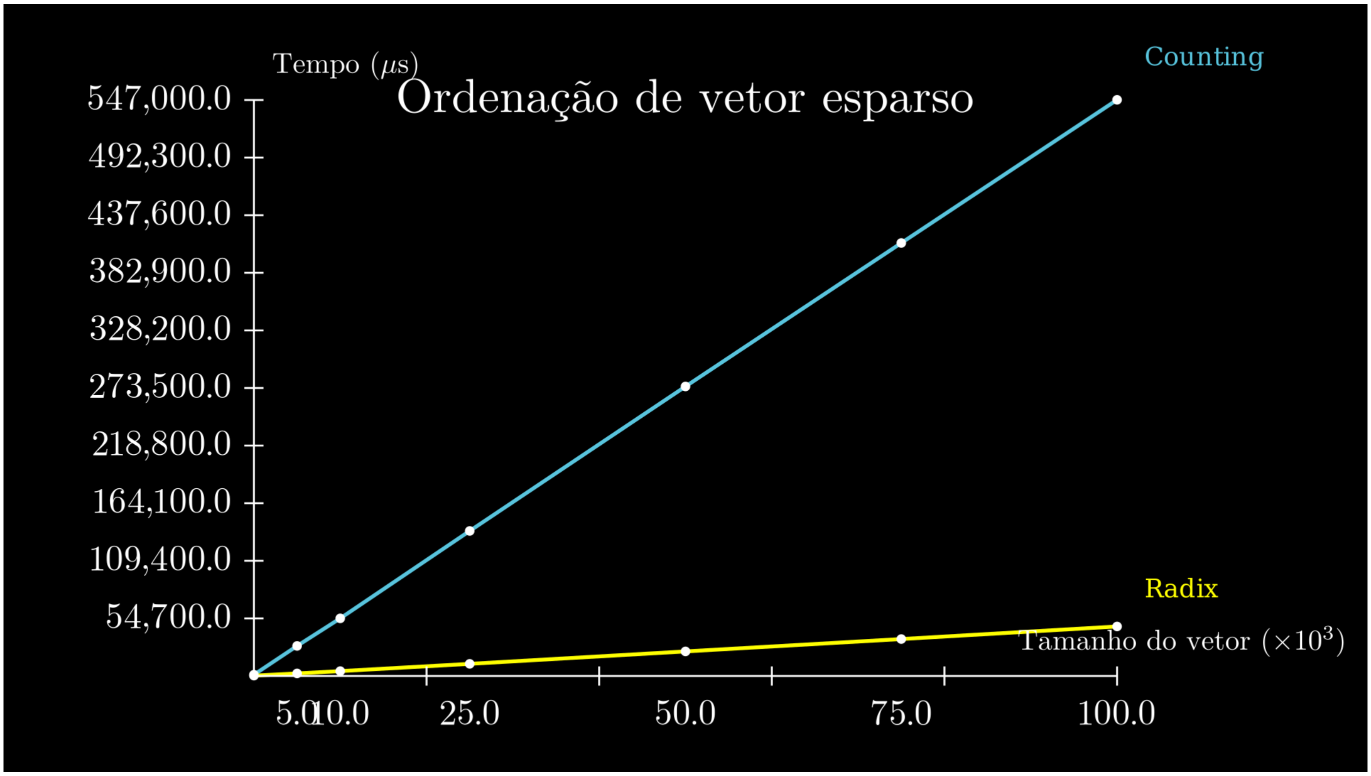
Assim, os métodos devem apresentar desempenho 2 vezes mais rápido que no vetor uniforme.



Ao comparar com o gráfico anterior, é possível observar que isso realmente acontece. O tempo médio passa de aproximadamente 32000 μs s para 16000 μs.

3.3 Vetor Esparso

Ao contrário do denso, o vetor esparso possui os elementos espalhados e pouca repetição. O valor máximo é até 100$\times$ o tamanho do vetor. Logo, a complexidade é $O(n + k) \rightarrow O(n + 100n)$, isto é, 55.5$\times$ pior que no vetor uniforme para Counting e Bucket. Entretanto, com a escolha correta de base para o Radix, é possível manter baixo tempo de execução.

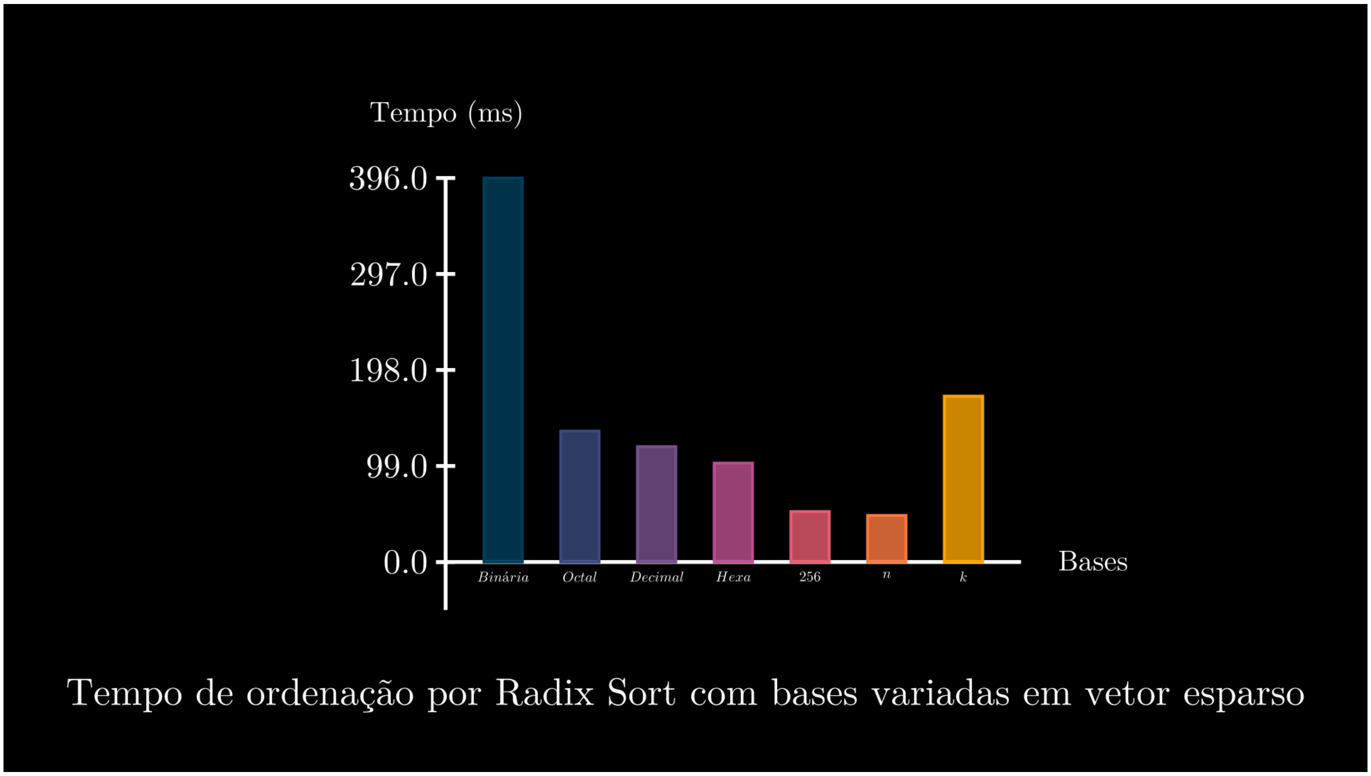


O computador não conseguiu suportar o alto custo de espaço do Bucket Sort, com milhares de listas encadeadas, excedendo a memória reservada para o funcionamento do algoritmo e assim sendo retirado do gráfico.

Nos algoritmos restantes, entretanto, Counting funciona muito mal, devido ao alto valor de k, enquanto com um bom b, o Radix continua com pequena constante.

3.4 Base do Radix

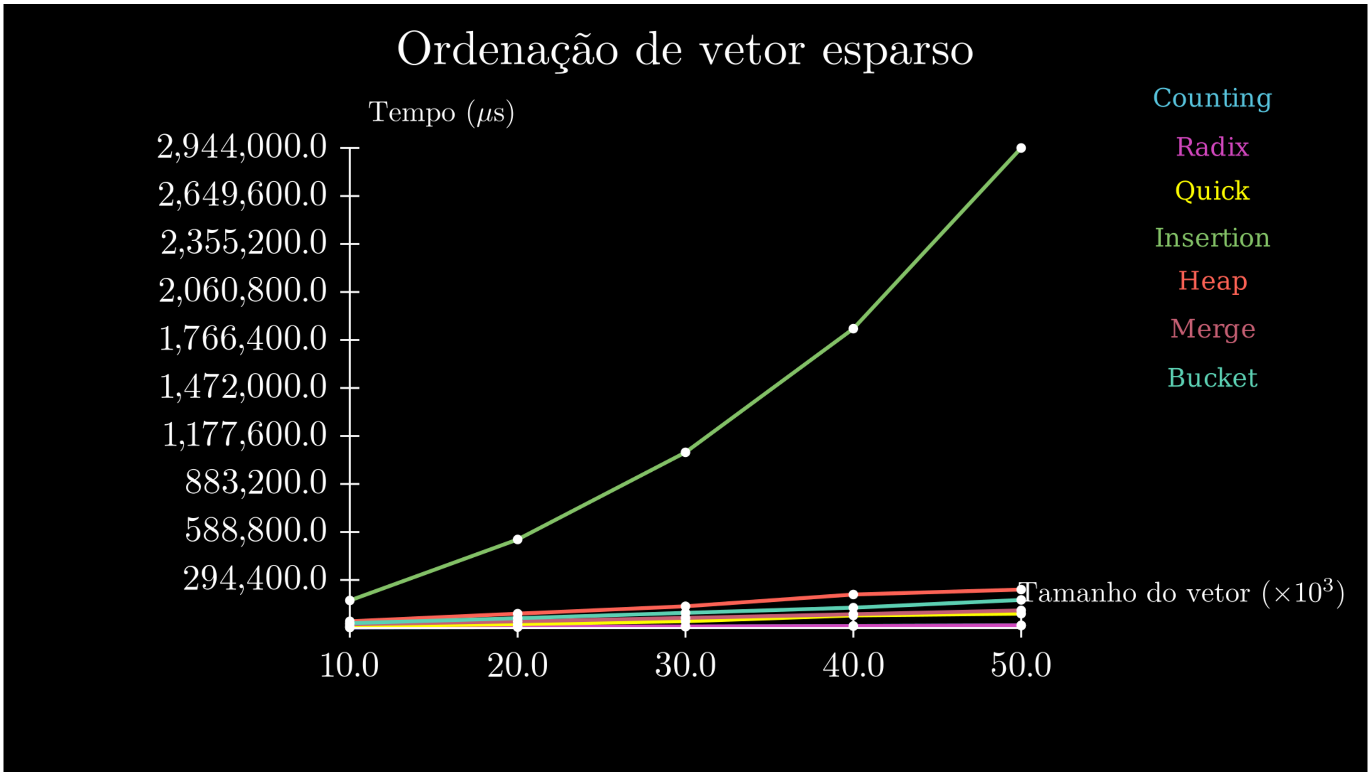
O diferencial dos dois outros métodos para o Radix Sort é a escolha de base, que permite ao algoritmo quebrar um número em partes para ordenar com tempo linear e baixo tempo de execução em qualquer situação. O gráfico a seguir mostra porque a melhor escolha de base é $b = n$.



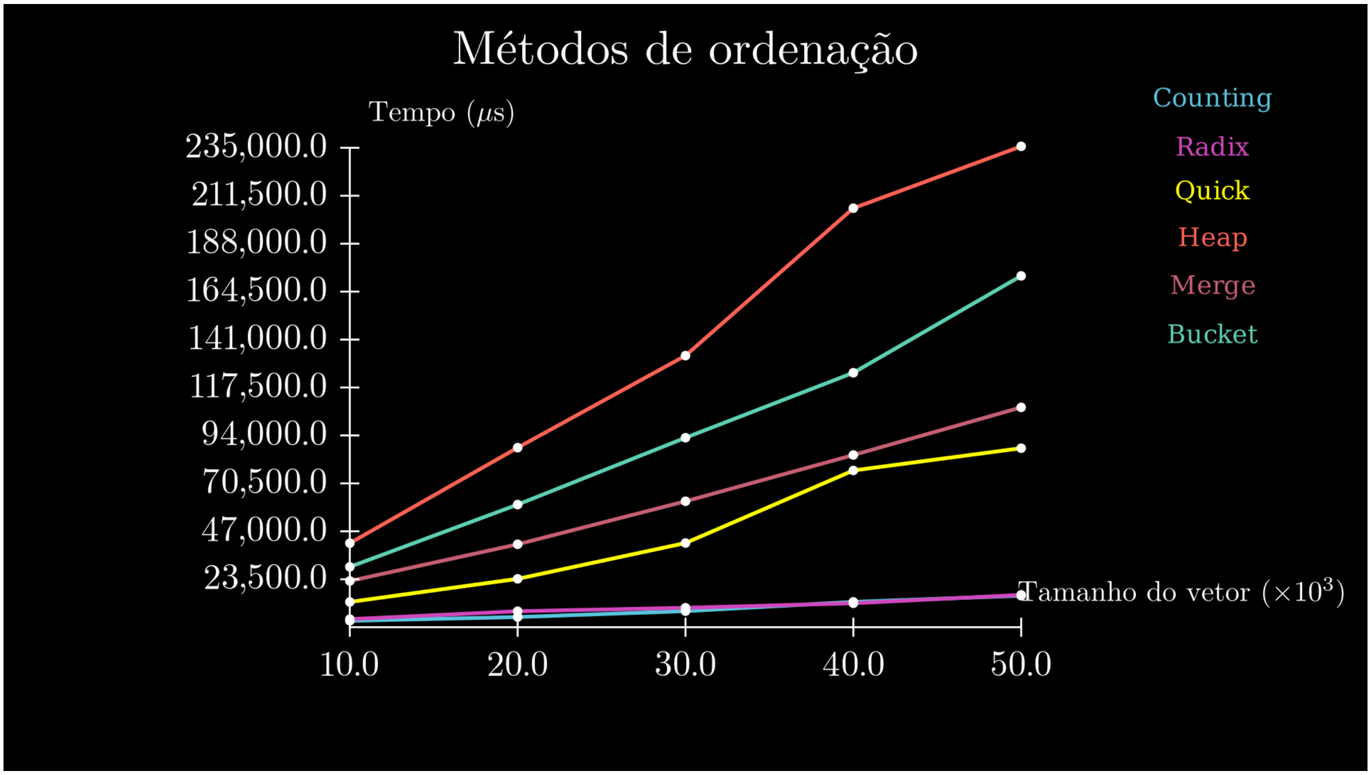
Nesse gráfico foi utilizado um vetor de tamanho 100000 e máximo de 10000000. De todos as bases, binária, decimal, octal, k, a que melhor performa é n. Assim, confirmam-se os cálculos feitos na parte de metodologia sobre o Radix Sort. 2.4 apesar de ainda linear, é bem pior que o Radix.

3.5 Comparação com outros métodos

Também é preciso comparar os métodos estudados nesse relatório com os passados. Ainda que os lineares tendam a ser melhores no geral, cada um tem suas nuances e especializações, de forma que é importante analisar a situação antes de escolher o método com o qual se deseja trabalhar. Para a construção desse gráfico foram utilizados vetores de tamanho 10000 a 50000, randomicamente gerados.



O Insertion, de complexidade $O(n^2)$, atrapalha a análise dos outros métodos. Ao retirá-lo, obtemos



Assim, é possível observar os métodos de contagem se sobressaindo, enquanto aqueles que utilizam comparação se mostram mais demorados.

4 Conclusão

Durante a elaboração desse relatório, além de aprender mais sobre a ferramenta LaTeX, foi possível constatar a eficiência dos métodos estudados na teoria, bem como comparar com o que havia sido estudado ao longo do ano. Com o Bucket Sort sofrendo por *Stack Overflow*, foi possível entender quão grave é a super utilização de ponteiros. Também pude constatar que a utilização da base $n = k$ é a mais eficiente dentre todas, no método mais eficiente. Por fim, da comparação com outros métodos que estudamos ao longo do ano, foi possível comparar a linearidade e linear-logaritmicidade entre eles.

Referências

[1] Erik Demaine. *7. Counting Sort, Radix Sort, Lower Bounds for Sorting.* MIT OpenCourseWare, 2013. URL: `https://www.youtube.com/watch?v=Nz1KZXbghj8`.

[2] Jason Ku Erik Demaine e Justin Solomon. *Introduction to Algorithms. Recitation 5.* URL: `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-spring-2020/lecture-notes/MIT6_006S20_r05.pdf`.

[3] Moacir Antonelli Ponti. *ICC2 (3) 3 - Bucketsort: algoritmo e análise.* Universidade de São Paulo, 2021. URL: `https://www.youtube.com/watch?v=8ZyFKHJJJIA`.