Universidade de São Paulo

Instituto de Ciências Matemáticas e Computação

Relatório 3 de Laboratório de Introdução à Ciências da Computação 2

Vítor Amorim Fróis

14 de janeiro de 2022

Resumo

No módulo 2 da disciplina de LICC2 buscamos analisar métodos de ordenação mais complexos que os tradicionais Bubble, Insertion e Merge, que podem ter complexidade de até $O(n^2)$. Aqui, Heap e Quick buscam o limite inferior de $O(n \log n)$. Na parte de metodologia são feitos os cálculos para provar esse valor enquanto na parte de resultados é possível comparar o que foi feito aqui com outros métodos e confirmar que a complexidade perfoma como esperado.

---

1 Introdução

Durante o módulo 1, métodos de ordenação simples, como o Bubble, Insertion e Merge. Os dois primeiros tinham complexidade $O(n^2)$, enquanto o Merge conseguia alcançar complexidade linear-logarítmica. Agora, através de novas ideias, é preciso aprimorar os métodos para buscar complexidade $O(n \log n)$ corriqueiramente. Esses são Heap e Quick.

2 Metodologia e desenvolvimento

2.1 Heap Sort

O Heap Sort é um método de ordenação que usa a estrutura chamada de Heap. Essa é uma estrutura semelhante a uma árvore binária, mas também pode ser representada através de um vetor.

No algoritmo, os elementos são posicionados em max heap, operação que possui complexidade $O(\log n)$ [2].

```
void heapify(int *array, int size, int i){
    int largest = i;  //b
    int l = 2 * i + 1; //2a + b
    int r = 2 * i + 2; //2a + b

    if (l < size && array[l] > array[largest]) //2c
        largest = l; //b

    if (r < size && array[r] > array[largest]) //2c
        largest = r; //b

    if (largest != i) { //c
        swap(&array[i], &array[largest]); //3b
        heapify(array, size, largest); //H(n/2)
    }
}
```

Então, troca os elementos $[1]; [n]$ de posição e descarta o último elemento ao diminuir o tamanho n do vetor [1].

```
    for (int i = size - 1; i > 0; i--) { //N
        swap(&array[0], &array[i]); //3b
        heapify(array, i, 0); //heapify
    }
```

Novas chamadas recursivas são executadas até que o vetor esteja completamente ordenado. Como é necessário passar pelo vetor uma vez para executar esse método, a complexidade é $O(n) \times O(\log n) = O(n \log n)$. Já que independente dos casos o mesmo algoritmo sempre é executado, o pior caso é igual ao maior caso. A complexidade de espaço é $O(1)$, usada apenas para os *swaps*.

2.2 Quick Sort

Já o Quick Sort é um algoritmo de ordenação muito eficiente que utiliza pivôs para ordenação. A partir desse elemento, chamadas recursivas onde cada elemento deve ser menor que o pivô são feitas e a ordenação se completa. Para o algoritmo manter sua alta eficiência, é necessário que a escolha de pivô seja bem feita. Assim, uma técnica utilizada é fazer a mediana entre o primeiro, último e elemento médio do vetor. Isso busca garantir que o elemento escolhido esteja o mais próximo da média de valores sem muito poder computacional.

```
1
2  int averagePivo(int *array, int began, int end){
3      int averageArray[3];
4      averageArray[0] = array[began]; //b
5      averageArray[1] = array[(began + end) / 2]; //2a + b
6      averageArray[2] = array[end-1]; //a + b
7      insertionSort(averageArray, 0, 3); //3^2= 9, constante
8      //Total: 3a + 3b + 9
9      return averageArray[1];
10 }
```

No pior caso, as chamadas recursivas são do tipo $n - 1$, e assim o algoritmo deve percorrer todo o vetor, elemento a elemento, para ordenar. Contudo, no melhor caso possível, há chamadas recursivas de tamanho $\frac{n}{2} - 1$ (considere n o tamanho da chamada recursiva anterior), de forma que o melhor caso executa $\log_2 n$ chamadas [3].

```
1
2  void quicksort(int* array, int began, int end){
3    int i, j, pivo, aux, n;
4      n = end - began; //a + b
5
6      pivo = averagePivo(array, began, end); //3a + 3b + 9
7
8    i = began;
9    j = end-1; //a + 2b
10
11   while(i <= j){ //N
12     while(array[i] < pivo && i < end) //a + b + 2c
13       i++;
14
15     while(array[j] > pivo && j > began) //a + b + 2c
16       j--;
17
18     if(i <= j){ //c
19       aux = array[i]; //b
20       array[i] = array[j]; //b
21       array[j] = aux; //b
22       i++; //2a + 2b
23       j--;
24     }
25   }
26     //N(4a + 7b + 5c)
27
28     if(j > began) //c
29         quicksort(array, began, j+1); //Q
30
31     if(i < end) //c
32     quicksort(array, i, end); //Q
33
34     //Total = 5a + 6b + 2c + 9 + N(4a + 7b + 5c) + Q
35
36 }
```

Pela análise do algoritmo, cada chamada possui complexidade $O(n)$. Assim, no melhor caso, o algoritmo faz $\log n$

chamadas de $O(n)$, resultando em complexidade de tempo $O(n \log n)$. Um ponto importante de se destacar é a não utilização de vetores auxiliares, que traz complexidade de espaço 0 para o algoritmo.

2.3 Comparação com métodos do módulo 1

Apesar de compartilharem complexidade com Merge, os métodos estudados no módulo presente são melhores que Insertion e Bubble. Assim, no caso médio, espera-se que log-lineares se sobressaiam, enquanto aqueles de ordem quadrática performem pior. Ainda assim, cada método, mesmo que com piores constantes, pode ser útil em ocasiões especiais.

3 Resultados

Para comparar Heap e Quick, serão plotados inicialmente gráficos até 10k e 100k. Então, um gráfico especial para o Quick será produzido a fim de aprofundar no pior caso desse algoritmo, que segundo a parte de Metodologia, tem complexidade $O(n^2)$.

3.1 Caso Médio

É esperado que ao *plottar* os gráficos das funções, os comportamentos sejam parecidos, já que a complexidade é igual, ao menos no caso médio.

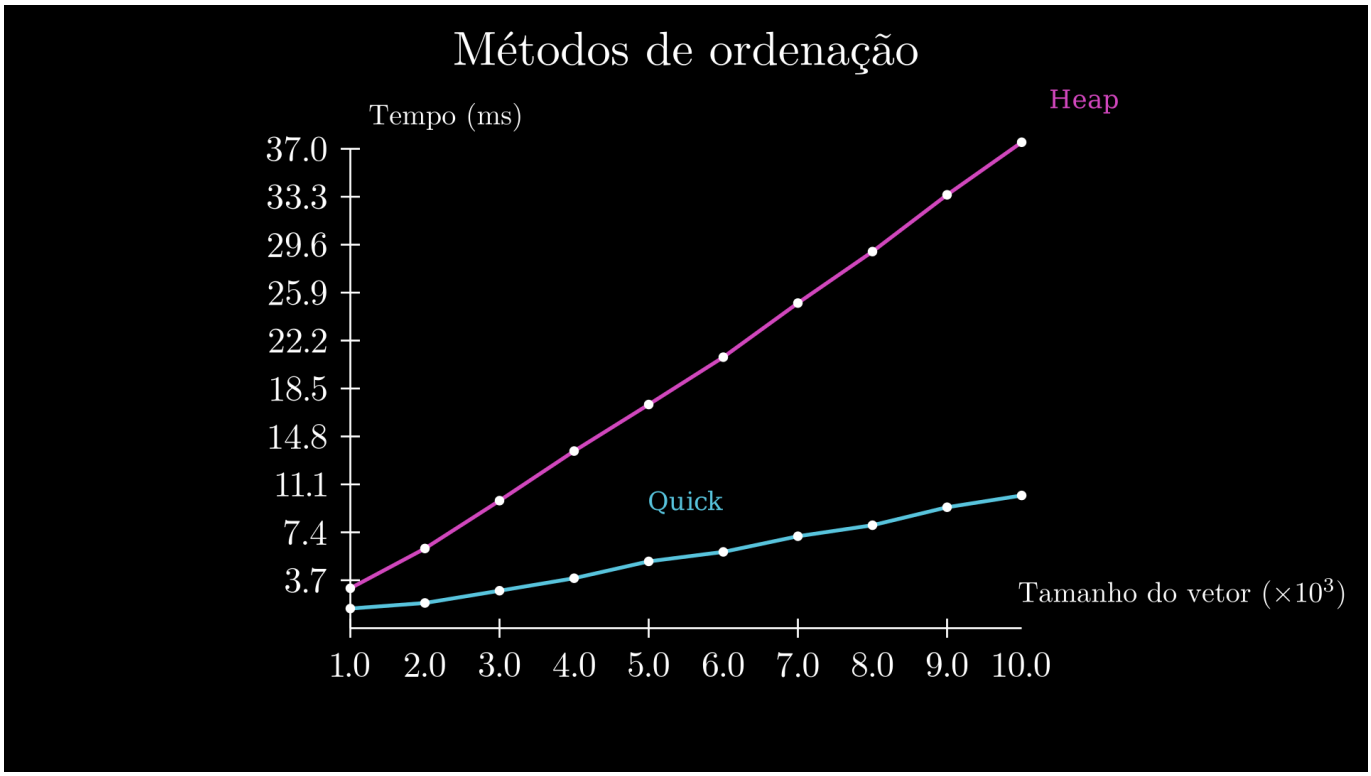

Figura 1: Ordenação de vetores randômicos com até 10000 elementos

A primeira coisa que chama atenção nesse gráfico é a disparidade entre o Heap e o Quick. Isso se deve ao fato de que as constantes do primeiro são muito maiores que as do segundo. Ainda é possível notar que apesar de parecer uma

reta, o gráfico possui uma leve inclinação, originária do $\log_2 n$ na função de complexidade.

Com o tempo, a função logarítimica tende a encontrar uma inclinação menor, portanto quanto maior o valor de n, mais $O(n \log_2 n)$ se aproxima, proporcionalmente, de uma função linear. Observar o gráfico com valores até 100000 permite melhor visualização.

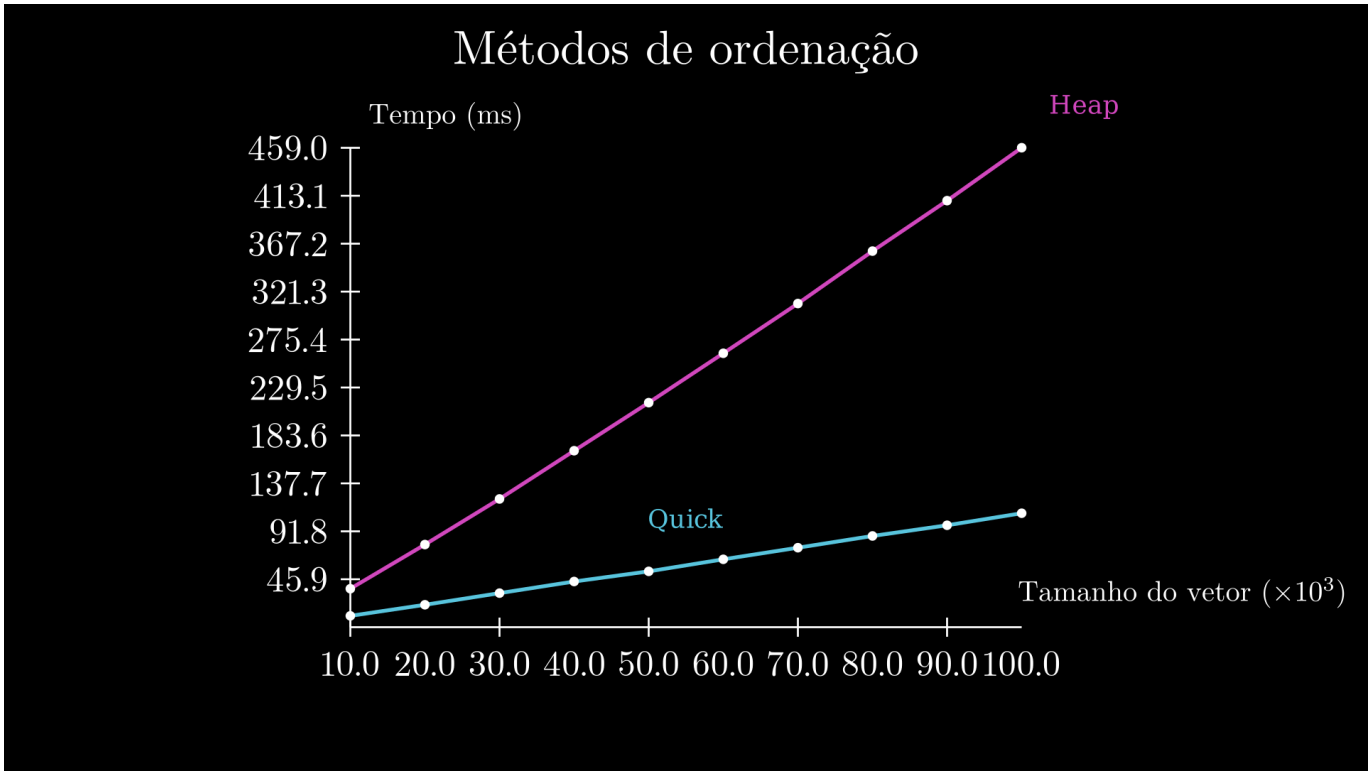

Figura 2: Ordenação de vetores randômicos com até 100000 elementos

3.2 Pior Caso

Entre os dois algoritmos estudados, somente o Quick apresenta pior caso. Assim, o gráfico busca comparar seu pior caso com o caso médio do Heap.

```
1    //pivo = averagePivo(array, began, end); //3a + 3b + 9
2    pivo = array[0];
```

O pivo será escolhido como o primeiro elemento do vetor, o qual estará inversamente ordenado.

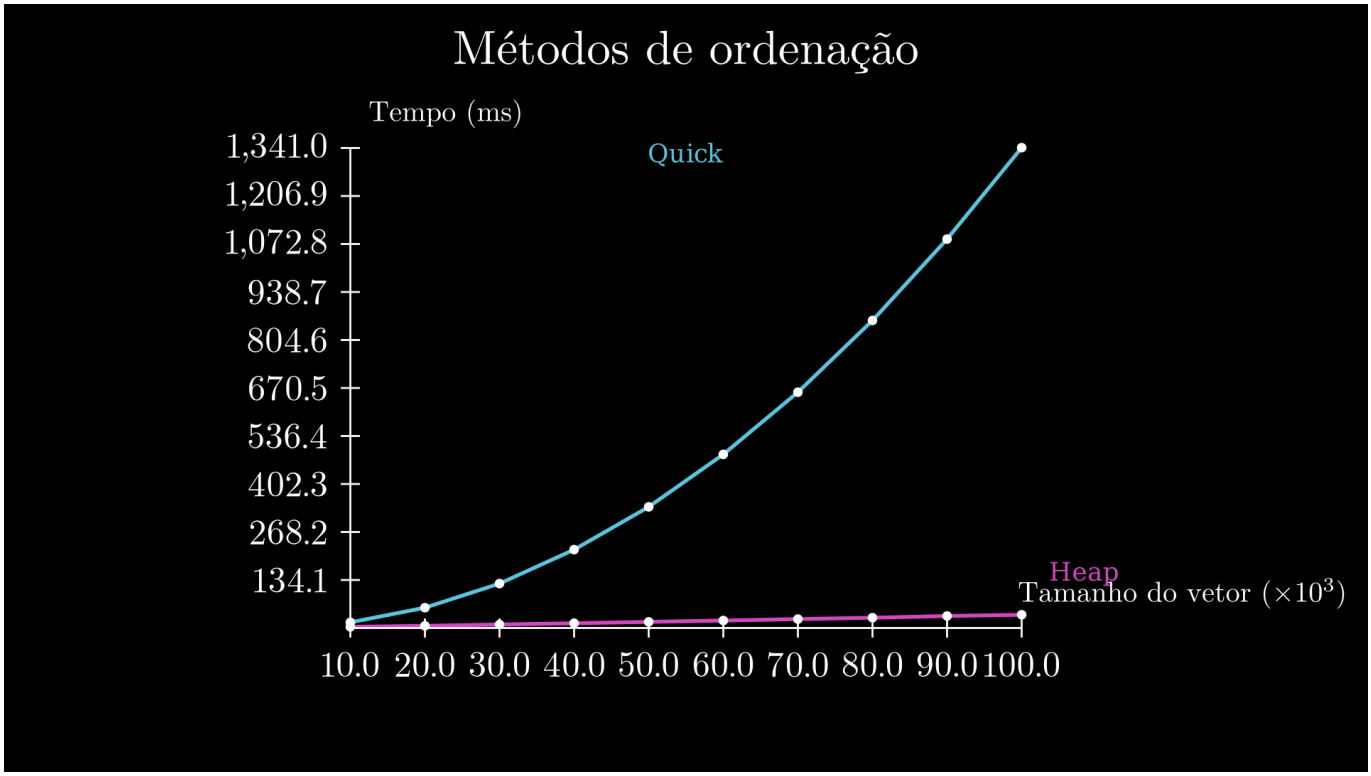

Figura 3: Pior caso do Quick Sort

Assim, pode se observar que em seu pior caso, o Quick Sort realmente apresenta complexidade $O(n^2)$, enquanto o Heap continua com pior caso $O(n \log n)$.

3.3 Comparação com métodos anteriores

Ao comparar com os métodos anteriores, o gráfico do caso médio se porta como esperado Nota-se que o Insertion dispara, enquanto os métodos que possuem complexidade $O(n \log n)$ ficam próximos. A pequena diferença entre eles se dá pelas constantes, como já comentado anteriormente.

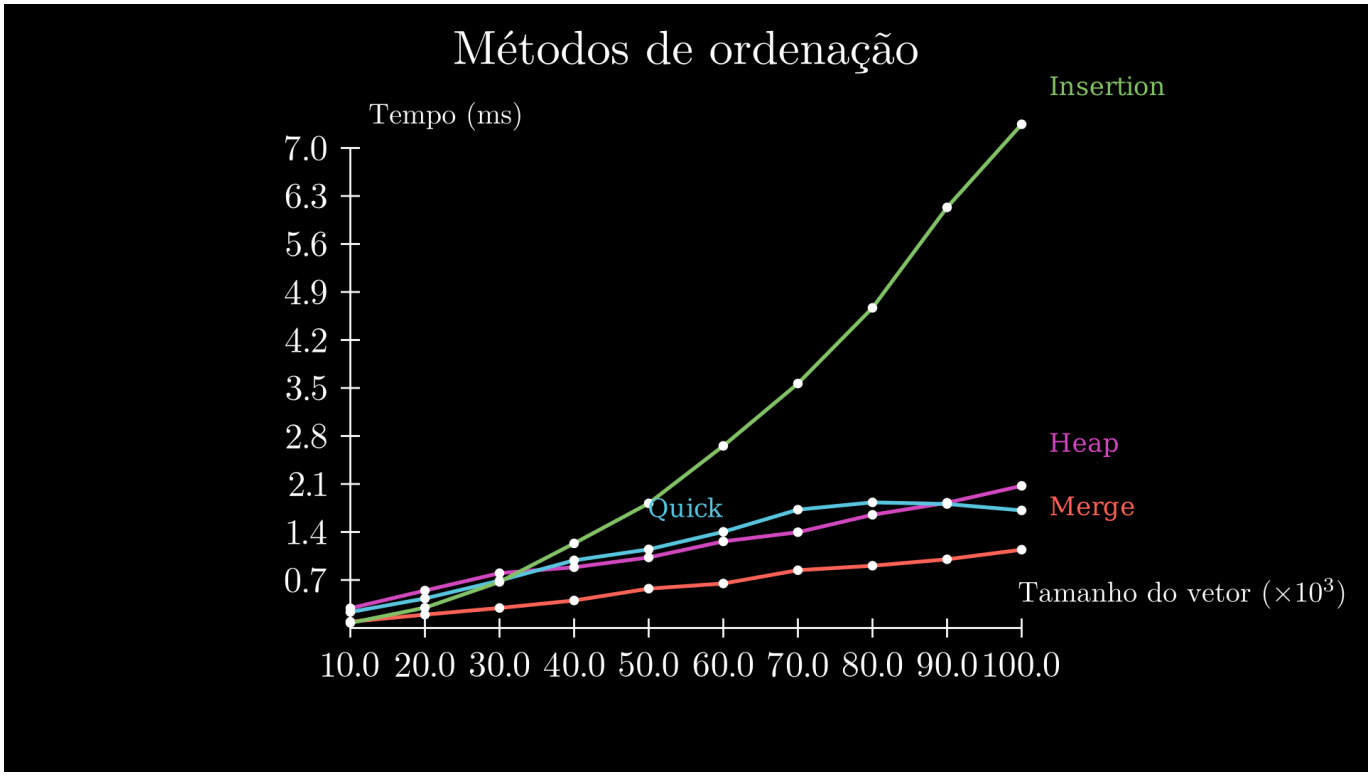

Figura 4: Todos métodos de ordenação

4 Conclusão

Concluímos ao longo do relatório que os dois algoritmos são muito eficientes, porém, o Heap é tem uma garantia maior de complexidade baixa. Para usar o Quick Sort, é essencial garantir que o pivô escolhido seja bom, caso contrário sua complexidade dispara, tornando-se $O(n^2)$.

É também possível observar que as análises feitas na seção de Metodologias tiveram seus resultados confirmados, sendo mais fácil constatar ao comparar os algoritmos do segundo módulo com os do primeiro.

Ainda assim, cada método possui sua particularidade, e analisar um problema e suas nuances antes de escolher um algoritmo para resolver é essencial.

Referências

[1] Srini Devadas. *4. Heaps and Heap Sort*. MIT OpenCourseWare, 2013. URL: `https://www.youtube.com/watch?v=B7hVxCmfPtM`.

[2] Ew Dijkstra. *Heap Sort*. 2014. URL: `http://wiki.c2.com/?HeapSort`.

[3] Moacir Antonelli Ponti. *ICC2 (2) 5 - Quicksort: análise de pior e melhor caso*. Universidade de São Paulo, 2021. URL: `https://www.youtube.com/watch?v=A5eywcvvNEQ`.