Relatório 2 de Laboratório de Introdução à Ciências da Computação

Introdução

A ordenação é, em ciências da computação, uma ferramenta importante para solucionar vários problemas. Assim, é esperado que durante a produção de um código, o algoritmo mais apropriado para a situação seja utilizado.

No módulo passado, estudamos o comportamento de métodos como Bubble, Insertion e Merge Sort, através de medições, análises assintóticas e gráficos. É necessário fazer a mesma análise com os novos métodos aprendidos: Heap e Quick Sort.

Como no relatório 1, para a produção dos gráficos, foi utilizada a linguagem Python, em conjunto com a biblioteca Manim, que permite criar gráficos flexíveis de alta qualidade.

Metodologia

Antes de começar a falar dos métodos em si, é preciso detalhar funções que fizeram essa análise possível.

Medições

Para garantir que não haveriam grandes desvios nas medições, todos os valores foram obtidos por meio de médias (no caso do trabalho, max=100).

```
for(int i = step; i <= max; i += step)
        timeMeasure(i, max, iterations, argv[4]);
```

main.c

```c
int* randomArr(int arraySize, int max){
    srand(time(NULL));
    int* array = malloc(sizeof(int) * arraySize);
    for(int i = 0; i < arraySize; i++){
        array[i] = random() % max;
    }
    return array;
}
```

Cria um vetor de arraySize elementos aleatórios

```c
int* increasingArr(int arraySize, int max){
    int* array = malloc(sizeof(int) * arraySize);
    for(int i = 0; i < arraySize; i++){
        array[i] = i;
    }
    return array;
}
```

Cria um vetor de elementos crescentemente ordenados

```c
int* decreasingArr(int arraySize, int max){
    int* array = malloc(sizeof(int) * arraySize);
    for(int i = 0; i < arraySize; i++){
        array[i] = arraySize-i;
    }
    return array;
}
```

Cria um vetor de elementos inversamente ordenados

```c
int timeMeasure(int size, int max, int iterations, char* mode){
    double time_taken;
    clock_t t, totalTime = 0;
    int *array = randomArr(size, max);
    //printArray(array, 0, size);
    //quicksort(array, 0, size);
    if(strcmp(mode, "quick") == 0){
        for(int i = 0; i < iterations; i++){
            t = clock();
            quicksort(array, 0, size);
            t = clock() - t;
            totalTime += t;
        }
    }else if(strcmp(mode, "heap") == 0){
        for(int i = 0; i < iterations; i++){
            t = clock();
            heapSort(array, size);
            t = clock() - t;
            totalTime += t;
        }
    }else{
        printf("%s is a non-identified sorting method\n", mode);
        return 0;
    }
    //printArray(array, 0, size);
    free(array);
    totalTime = totalTime / iterations;
    time_taken = ((double)totalTime)/CLOCKS_PER_SEC;
    //printf("\ntime taken = %d", (int) (time_taken*1000000));
    printf("%sSort took %f seconds to execute the sort on %d elements\n", mode,
time_taken, size);
    return (int) (time_taken*1000000);
}
```

Função timeMeasure( )

A função timeMeasure recebe o tamanho, valor máximo, número de iterações e tipo de método e retorna o tempo médio gasto para ordenar crescentemente um vetor com as especificações inseridas em microssegundos (10^{-6}s).

```
timeMeasure(1000, 1000, 10, "quick");
```

Análise assintótica

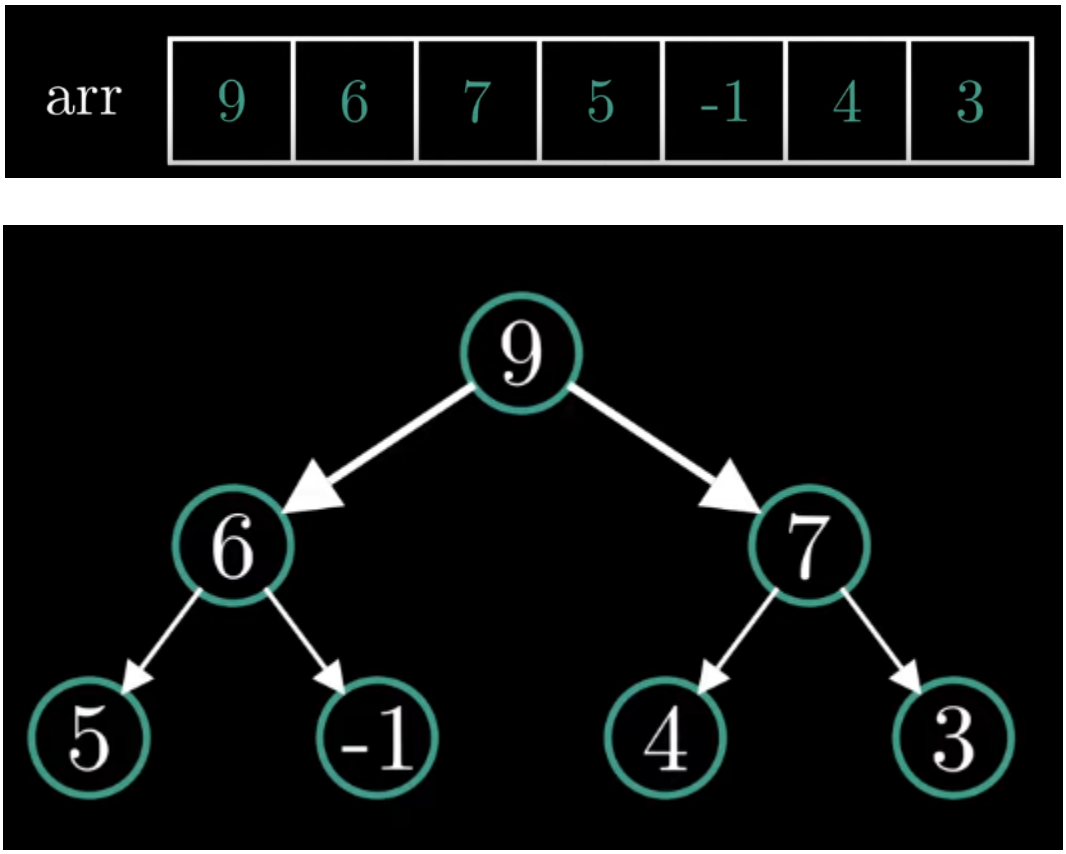
No relatório será a usada a análise assintótica afim de determinar a eficiência dos algoritmos.

Assim, ao analisar a função do algoritmo, serão cortadas constantes e expoentes de menor grau.

Além disso, as notações Big-O e Big-Omega serão usadas para determinar os melhores casos.

Heap Sort

O heapSort é um algoritmo que se utiliza de uma estrutura em formato de árvore binária para realizar as operações. Primeiro, é necessário organizar o vetor em maxHeap.



Estrutura de maxHeap

Observe que há uma ordenação dos elementos, para então realizar trocas chamadas recursivas e ordenar o vetor.

A função heapSort( ) faz chamadas iterativas de heapify, que força o maxHeap. Ao analisar a função heapify $h(x)$

$h(n) = a + 2(3a) + n(a) + n(a - i) + c * h(n - i)$

$h(n) = 7a + 2n(a)(a - i) + h(n - i)$

$h(n)$ é, portanto, $\theta(n)$

Como o heap é uma estrutura de árvore com chamadas recursivas, as chamadas de $h(n)$ ocorrem $log(n)$ vezes. Portanto, o heapSort é $\theta(nlog(n))$.

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

    if (l < size && array[l] > array[largest])
        largest = l;

    if (r < size && array[r] > array[largest])
        largest = r;

    if (largest != i) {
        swap(&array[i], &array[largest]);
        heapify(array, size, largest);
    }
}
```

```c
void heapSort(int *array, int size){
    for (int i = size / 2 - 1; i >= 0; i--)
        heapify(array, size, i);

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

O ponto forte do Heap é sua eficiência mesmo nos piores casos, isto é, o pior caso tem complexidade igual ao caso médio.Entretanto, **não** é um algoritmo estável.

Um algoritmo **estável** é aquele que preserva a ordem de registros de chaves iguais. Isto é, se tais registros aparecem na sequência ordenada na mesma ordem em que estão na sequência inicial.

Quick Sort

O Quick Sort é um algoritmo que utiliza pivôs para ordenação. Após a escolha de cada pivô, os elementos menores que o valor indexado devem estar atrás no vetor, enquanto os elementos maiores devem estar a frente. Após sucessivas escolhas de pivô, o vetor fica ordenado, e então a função é chamada recursivamente para a lista de elementos menores e maiores.

A complexidade do algoritmo pode ser expressa por:

$$Q(n) = 2a + 2a + n(2(2c + 2a) + 5a) + Q$$

$Q(n)$ tem complexidade $\theta(n)$

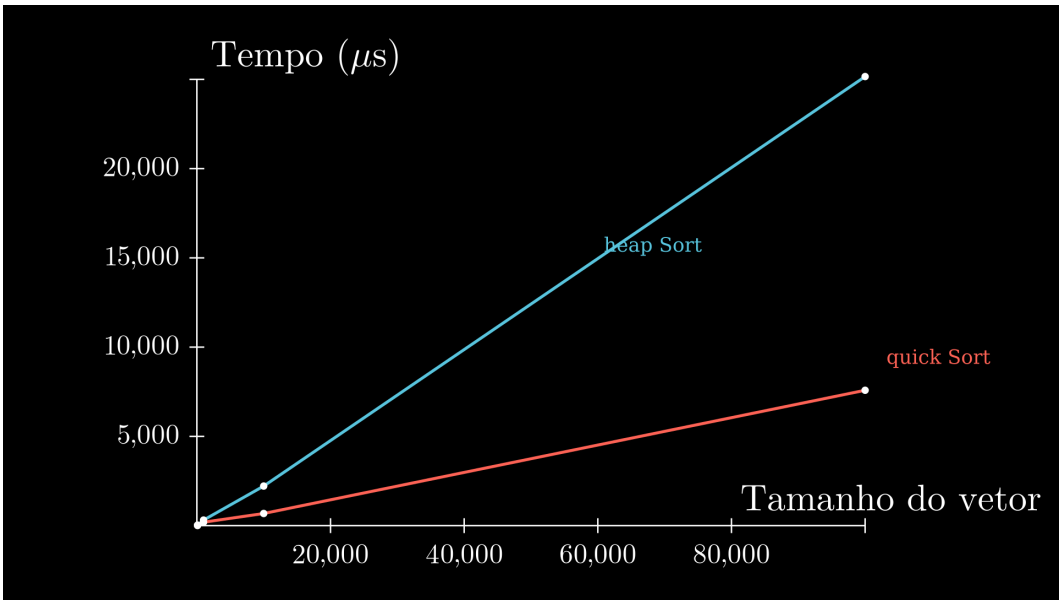
```c
void quicksort(int* array, int began, int end){
    int i, j, pivo, aux, n;
    n = end - began;

    //escolha média de 3 do pivô
    pivo = averagePivo(array, began, end);
    //escolha do último elemento como pivô para pior caso
    pivo = array[end-1];

    i = began;
    j = end-1;
    while(i <= j){
        while(array[i] < pivo && i < end)
            i++;

        while(array[j] > pivo && j > began)
            j--;

        if(i <= j){
            aux = array[i];
            array[i] = array[j];
            array[j] = aux;
            i++;
            j--;
        }
    }

    if(j > began)
        quicksort(array, began, j+1);

    if(i < end)
        quicksort(array, i, end);

}
```

Como no Heap Sort, as chamadas recursivas dividem o vetor em diversas partes, sendo esse resultado igual a $logn$. Logo, a complexidade média do Quick é $\theta(nlogn)$.

Porém, no pior caso, quando o pivô é mal escolhido, deve-se efetuar as chamadas recursivas um total de n vezes, aumentando a complexidade do algoritmo para $\theta(n^2)$ no pior caso ou $O(n^2)$.

Resultados

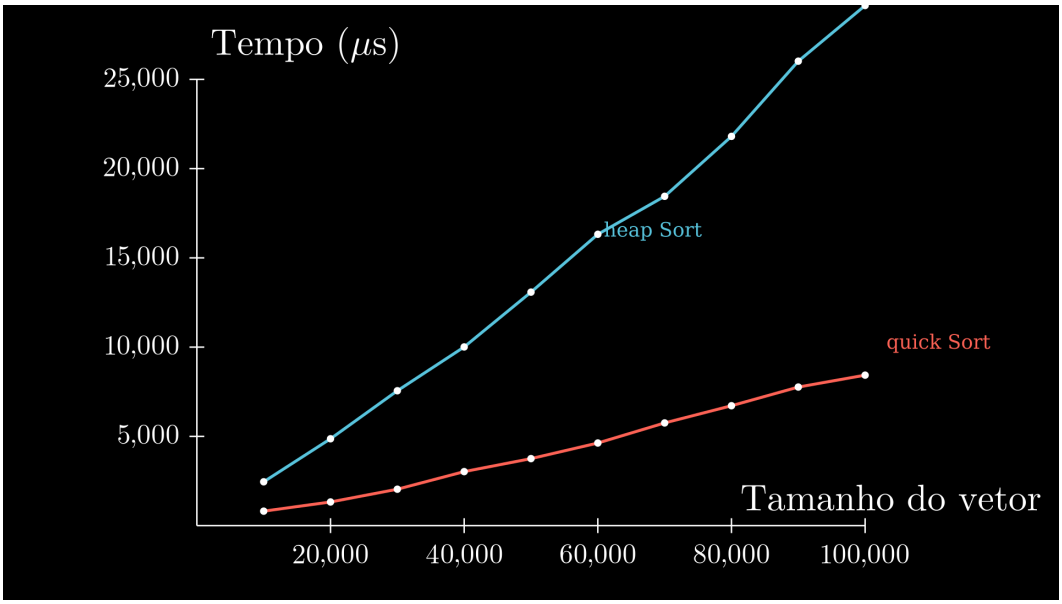
A sugestão do modelo de relatório foi trabalhar inicialmente com os tamanho de vetores 100, 1000, 10000 e 100000. Entretanto, isso gera um gráfico com poucos detalhes.



Já é possível observar tendência dos vetores: Apesar dos dois métodos serem muito bons, o quickSort se destaca

Novos gráficos

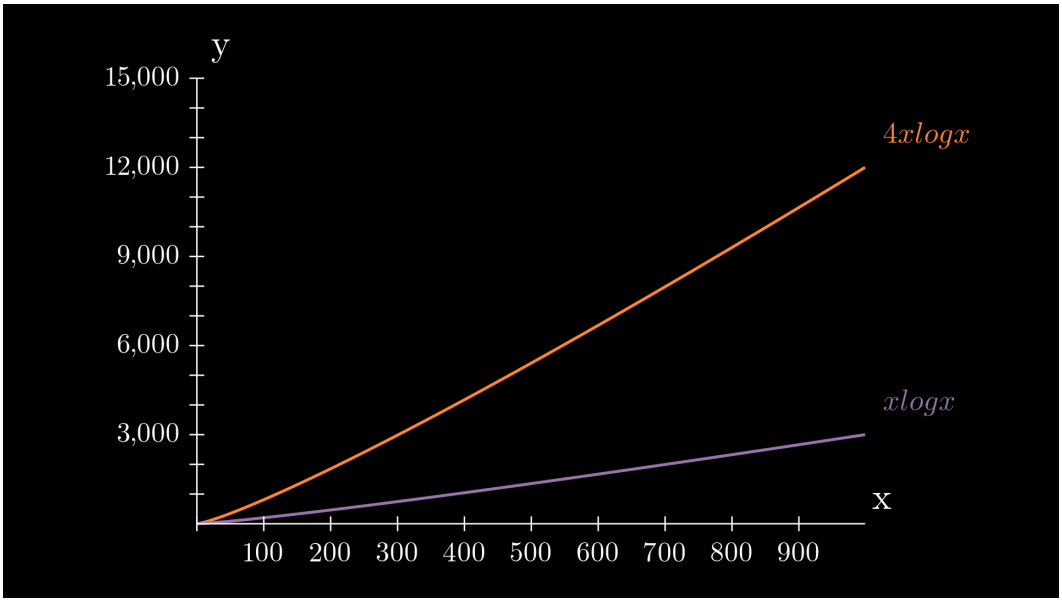
Portanto, outros gráficos com maior definição podem ser criados.



Para tamanhos de vetores muito pequenos, os gráficos podem parecer brigar a primeira vista, mas logo se definem, como dito acima.

Apesar das funções possuirem complexidade igual, o Quick Sort continua performando melhor. E analogamente ao Insertion e Bubble, que apesar de possuirem mesma complexidade, um é

muito melhor que o outro, isso acontece com Quick e Heap, devido aos coeficientes.



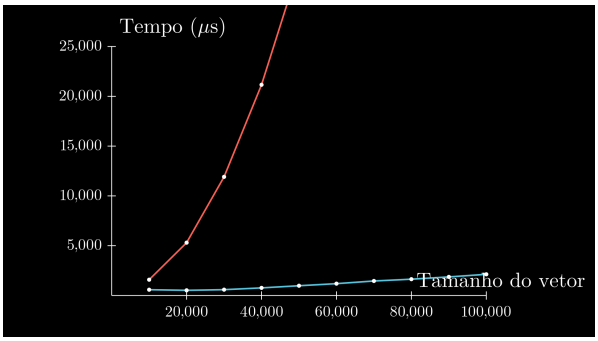
O coeficiente 4 faz a função em laranja crescer muito mais que a função em roxo

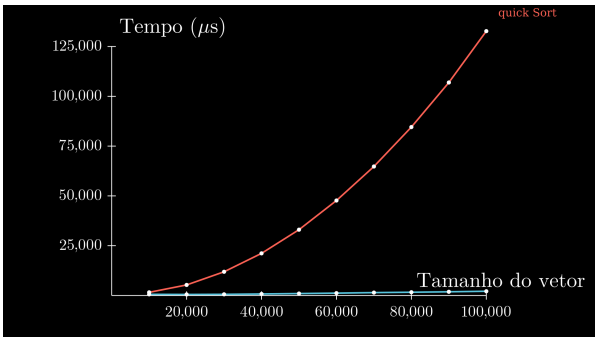
Melhor e pior casos

Para análise dos melhor e pior casos é importante notar que o Heap Sort é $\Theta(nlogn)$, isto é: o melhor caso se iguala ao pior caso. Isso pois independente do vetor estar ou não ordenado, o algoritmo vai cumprir a mesma função. Assim, a análise vai focar no Quick.

O pior caso do Quick Sort ocorre quando a escolha do pivô é ruim. Por exemplo, em um vetor ordenado, escolher o primeiro ou último elemento. Nessa situação, o algoritmo é $O(n^2)$.

Escolha ruim de vetor

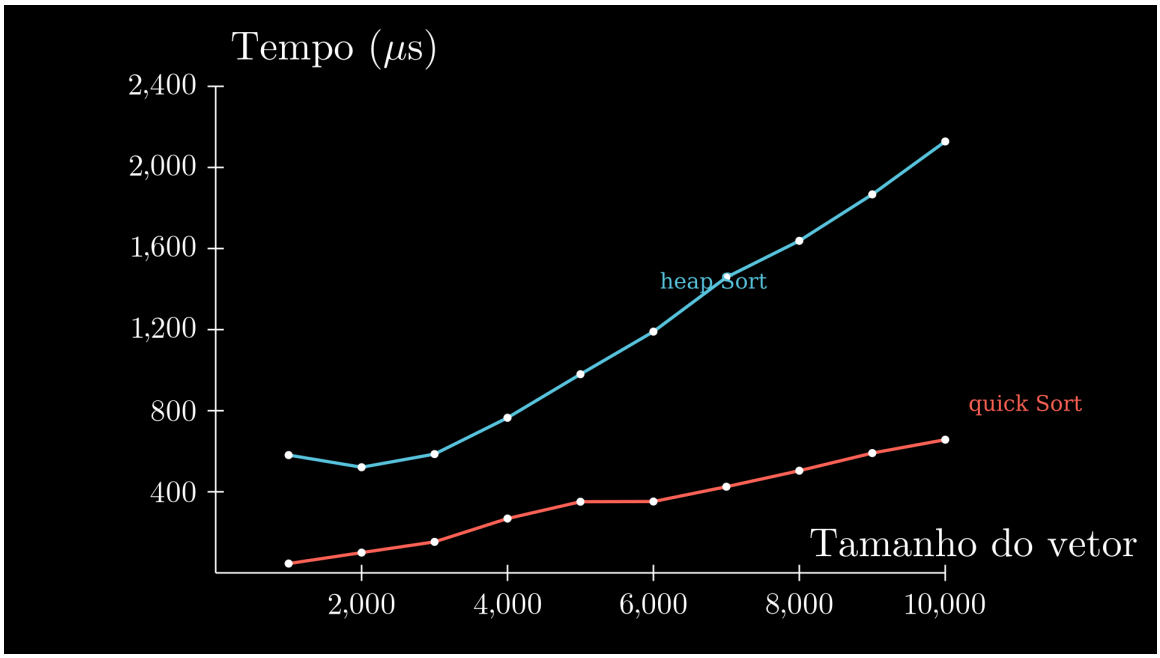




O quickSort, em vermelho, dispara. Para comparação, o heap está em azul para o mesmo vetor.

Assim, é essencial que, ao usar o quickSort, haja uma escolha boa de pivô. Um bom jeito de fazer isso é através da mediana de 3.

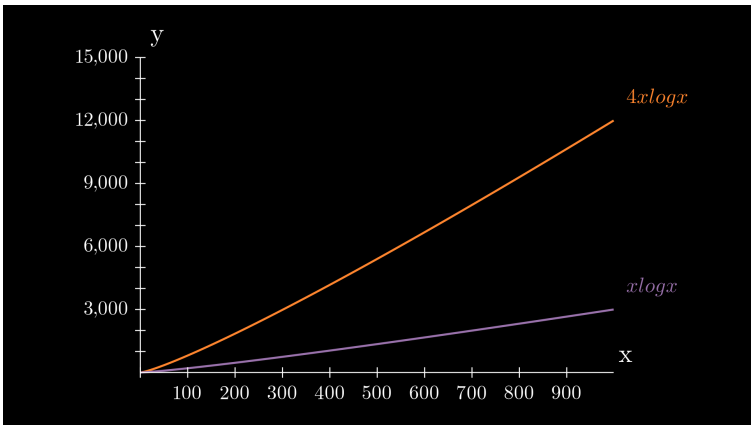
```c
int averagePivo(int *array, int began, int end){
    int averageArray[3];
    averageArray[0] = array[began];
    averageArray[1] = array[(began + end) / 2];
    averageArray[2] = array[end-1];
    insertionSort(averageArray, 0, 3);
    //printArray(averageArray, 0, 3);
    return averageArray[1];
}
```



Quick sort quando se escolhe um bom pivô.

O melhor caso

O melhor caso desses métodos é sempre $n \log n$. Pela diferença na quantidade de operações, apesar da complexidade igual, espera-se que o quick performe melhor, como mostrado no



O coeficiente 4 faz a função em laranja crescer muito mais que a função em roxo

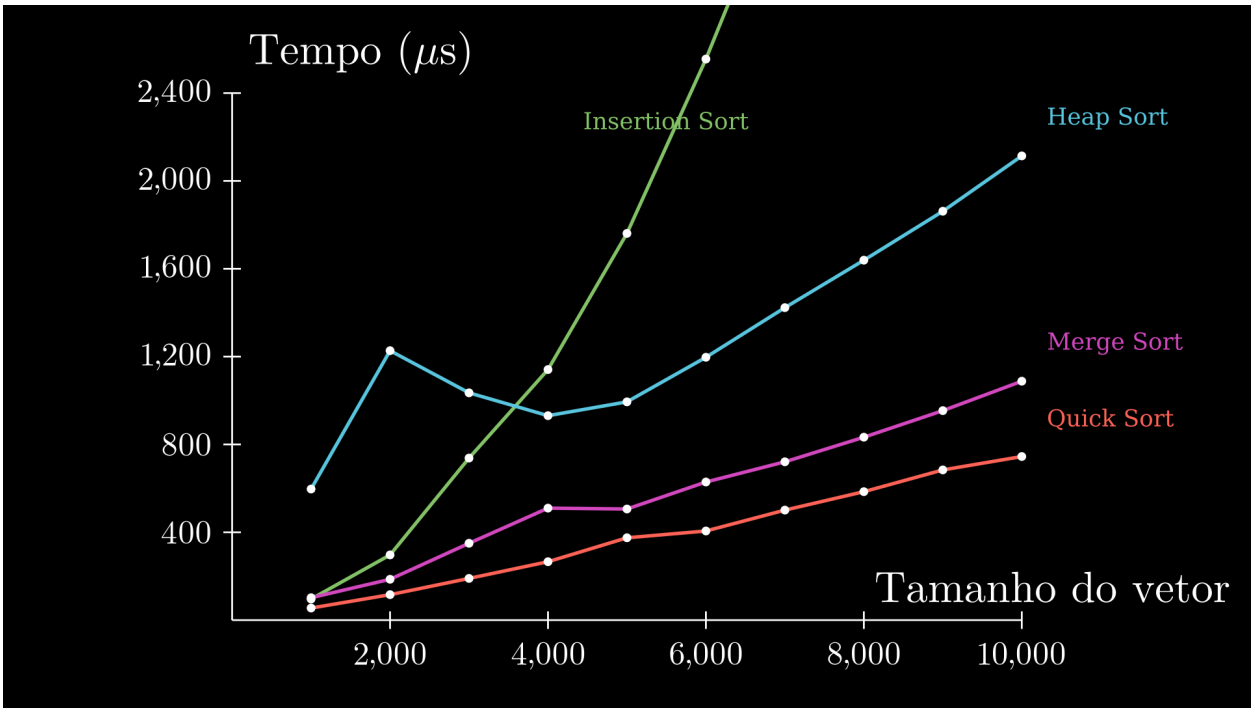
gráfico de funções iguais com coeficientes > 1.

Comparação dos métodos 2 com Merge e Insertion

Primeiramente é importante frisar que **ninguém** se importa com o bubble. Porém, ao longo das análises pudemos concluir que:

- O Quick Sort é o melhor algoritmo no geral
- Quando não for possível usar o Quick, por exemplo em vetores com elementos repetidos ou quando a escolha de um bom pivô seja difícil, e com vetores pequenos, pode se usar o Heap.
- Os dois métodos descritos acima não são estáveis. Assim, em casos que a estabilidade for essencial, deve se usar o Merge Sort.
- Por fim, o Insertion é um algoritmo que funciona muito bem em vetores quase ordenados, ou quando houver inserções.

Ao colocar todos os métodos num gráfico com vetores randômicos, obtém-se a imagem abaixo



Conclusão

A análise dos gráficos permitiu confirmar que a notação assintótica feita na seção de metodologia foi bem sucedida. Assim, espera-se usar disso em próximas atividades para calcular com antecedência a eficácia do código. Ainda é possível usar a notação Big-O e Big-Omega para detalhar os melhores e piores casos.

Dentre os algoritmos estudados, sempre que possível é interessante usar o Quick sempre que possível.

Esperava que o Heap Sort performasse melhor ou igual ao Merge, porém não foi o que ocorreu. Talvez por conta das constantes, ainda que, como é possível observar no gráfico, a complexidade é igual

Referências

- Merge-Sort with Transylvanian-Saxon (German) folk dance
- 15 Sorting Algorithms in 6 minutes
- Quick Sort in 4 minutes

- O Relatório 1