Relatório 1 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.

Na matéria de LICC2, aprendemos diversos métodos, entre eles: Bubble, Insertion e Merge Sort. Para testar a eficiência dos Sorts, deve-se efetuar uma análise conjunta entre as equações e tempos de execução em diferentes tamanhos de vetores e situações (pior e melhor caso).

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=10).

```
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;
}
```

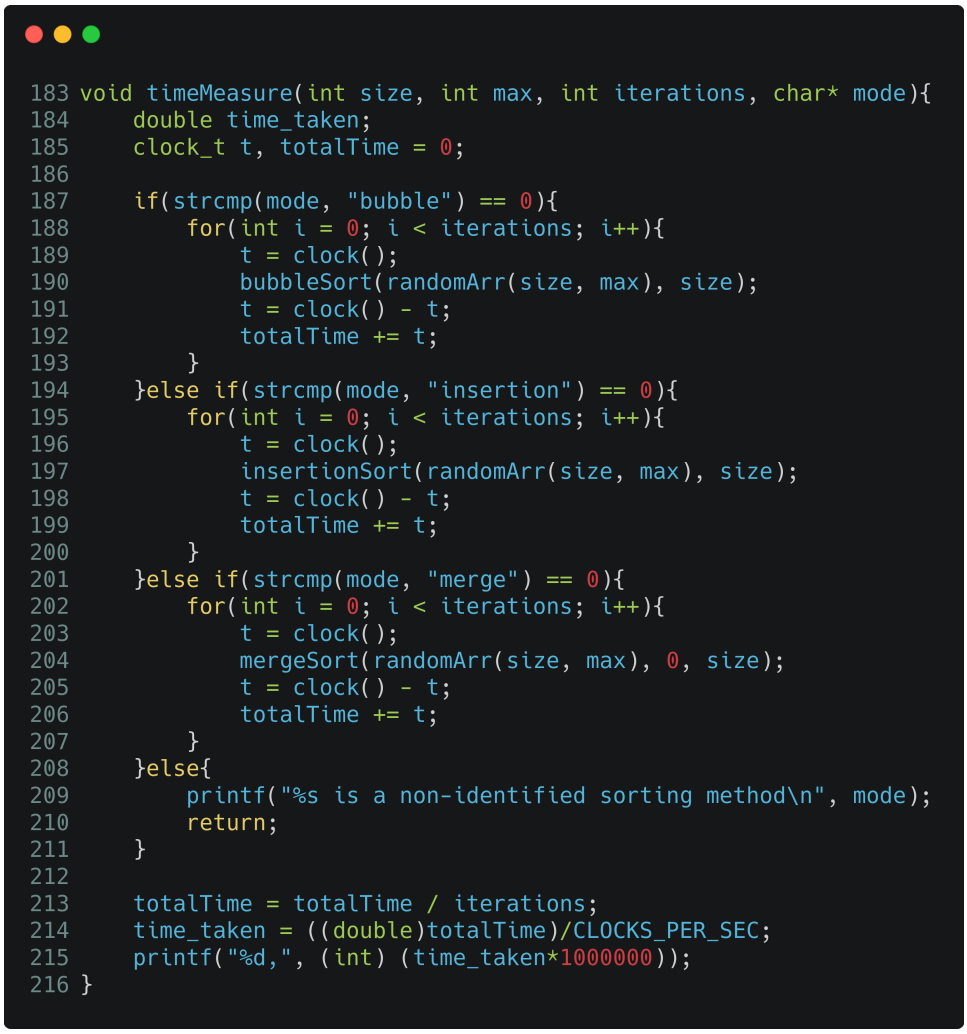
Cria um vetor de arraySize elementos aleatórios

```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
void timeMeasure(int size, int max, int iterations, char* mode){
    double time_taken;
    clock_t t, totalTime = 0;

    if(strcmp(mode, "bubble") == 0){
        for(int i = 0; i < iterations; i++){
            t = clock();
            bubbleSort(randomArr(size, max), size);
            t = clock() - t;
            totalTime += t;
        }
    }else if(strcmp(mode, "insertion") == 0){
        for(int i = 0; i < iterations; i++){
            t = clock();
            insertionSort(randomArr(size, max), size);
            t = clock() - t;
            totalTime += t;
        }
    }else if(strcmp(mode, "merge") == 0){
        for(int i = 0; i < iterations; i++){
            t = clock();
            mergeSort(randomArr(size, max), 0, size);
            t = clock() - t;
            totalTime += t;
        }
    }else{
        printf("%s is a non-identified sorting method\n", mode);
        return;
    }

    totalTime = totalTime / iterations;
    time_taken = ((double)totalTime)/CLOCKS_PER_SEC;
    printf("%d,", (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, "insertion");
```

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.

Bubble Sort

Entre os 3 métodos de ordenação, o Bubble é o mais simples.

O método consiste em, para cada elemento do vetor, conferir todos os outros elementos do vetor e *swappar* aqueles que estão na ordem errada. É considerado um algoritmo pouco eficiente.

```c
void bubbleSort(int arr[], int n)
{
   int i, j, swapCheck;

   for (i = 0; i < n-1; i++){
     swapCheck = 0;
     for (j = 0; j < n-i-1; j++){
        if (arr[j] > arr[j+1]){
           swap(&arr[j], &arr[j+1]);
           swapCheck = 1;
        }
     }
     if (swapCheck == 0)
        break;
   }
}
```

A função bubbleSort

Ao realizar a contagem de operações, obtém-se

$b(x) = x(a + c) \times x(2a + 2c)$
$b(x) = x^2(a + c)(2a + 2c) = x^2$
Simplificando, encontramos que $b(x) = x^2$. Esse é o pior e médio caso da função, logo ela é $O(n^2)$.

O código do Bubble Sort pode ser otimizado, adicionando uma variável que aborta a execução caso não sejam efetuados swaps (o vetor já está ordenado). Assim, no melhor caso, o vetor deverá ser percorrido no mínimo uma vez e a complexidade é $\Omega(x)$.

Insertion Sort

Ao inverso do bubbleSort que executa um swap por vez, o insertion percorre todo o vetor até encontrar um elemento menor, e somente então realiza a troca. Com essa evolução, consegue ser bem mais eficiente que o método anterior.

```c
void insertionSort(int* arr, int n){
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;

        /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

A função insertionSort

Realizando a contagem de operações:

$$i(x) = x(6a + c) \times x(4a + 2c)$$
$$i(x) = x^2(6a + c)(4a + 2c) = x^2$$

Assim, o insertionSort é $O(n^2)$. No entanto, quando o vetor está totalmente ordenado, ocorre o melhor caso, onde o percurso é feito somente uma vez, definindo o algoritmo como $\Omega(n)$.

Ainda que na fase de metodologia, é importante ressaltar: foi cravado que o Insertion é mais eficiente que o Bubble. Porém, eles possuem notação assintótica idêntica. Os motivos serão apresentados junto com os testes na seção seguinte do relatório.

Merge Sort

Por último, o Merge traz uma abordagem completamente diferente, advinda do lema dividir para conquistar.

```c
void mergeSort(int* arr, int l, int r){
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

Função de mergeSort

Essa primeira parte da função recursivamente divide o vetor em vetores menores de metade do tamanho até que cada vetor seja unitário. Então, chama a função `merge()`, que mescla os vetores, ordenando-os.

```c
void merge(int* arr, int l, int m, int r){
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
    are any */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
    are any */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Função merge

A imagem abaixo ajuda a entender melhor como funciona o processo de ordenação.

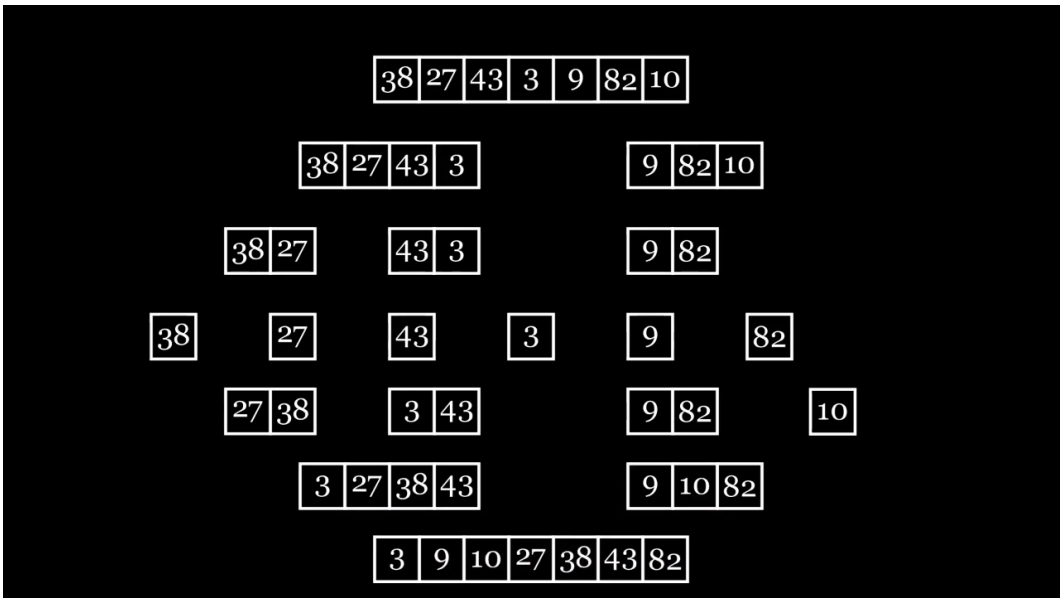


Imagem que mostra o processo de ordenação de um vetor via Merge

A complexidade da função pode ser analisada com ajuda da imagem. Mas primeiro, vamos analisar a complexidade da função `merge()`.

$$m(x) = 5a + 2x(3a + c) + 3a + x(4a + c)$$
$$m(x) = 3x(7a + 2c) + 8a = x$$

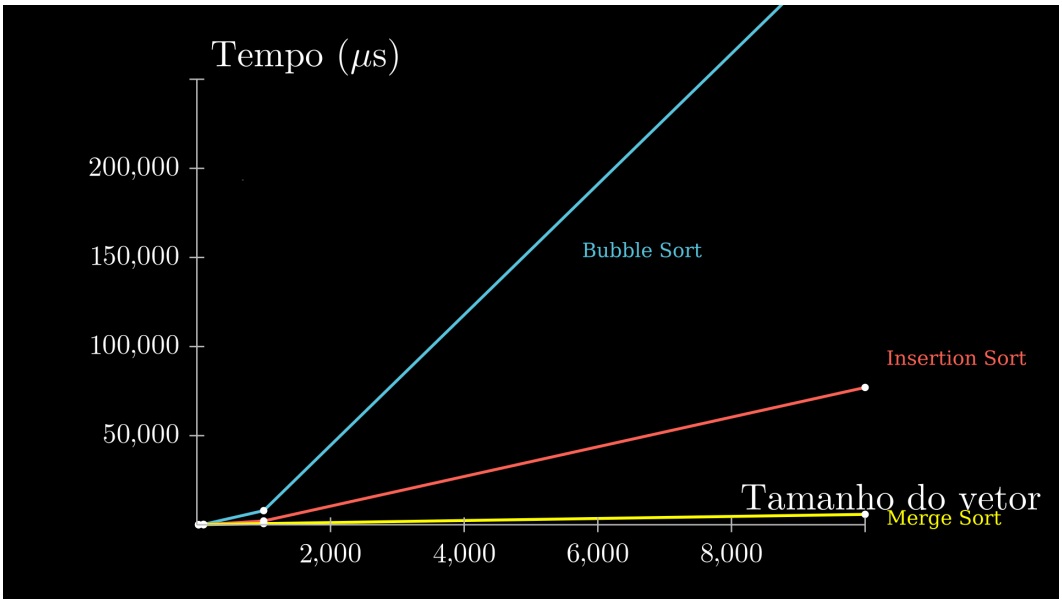
Simplificando, a função merge é $O(n)$. Agora, basta analisar quantas vezes o merge será chamado dentro da função `mergeSort()`. Considere o tamanho no vetor como **n**. É possível observar, pela imagem, que o vetor é dividido em 2 diversas vezes, e o merge será aplicado em todas as linhas, com processamento total de tamanho **n**. As sucessivas divisões podem ser algebricamente definidas por $log(n)$.

Assim, a complexidade final é definida por $x \log(x)$. Espera-se portanto, que o gráfico do Merge não cresça tanto quanto os anteriores.

É importante citar ainda, que o Merge Sort gasta muita memória devido ao número excessivo de vetores auxiliares criados e acessados.

Resultados

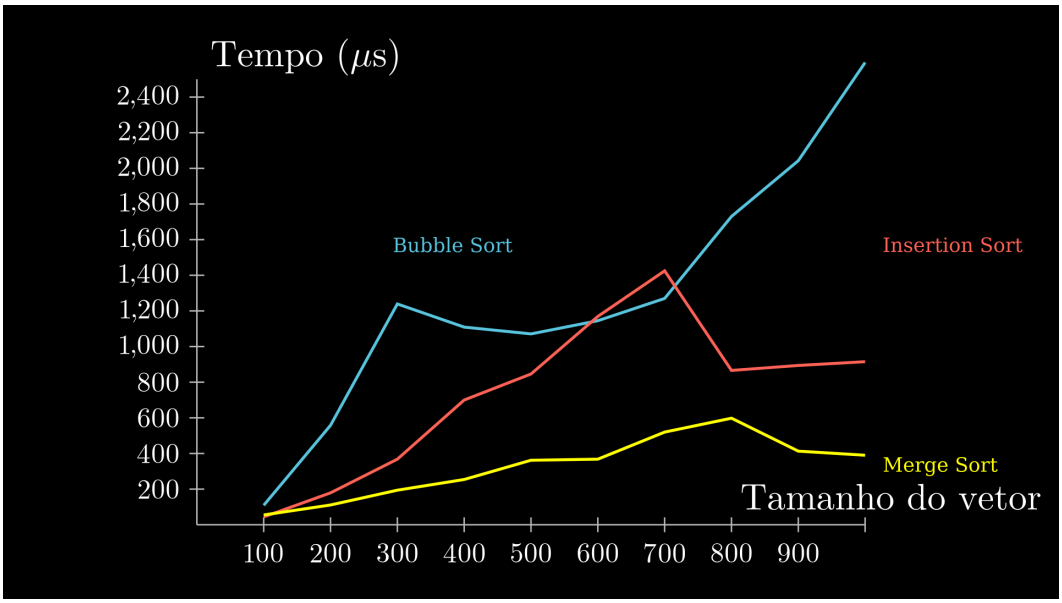
A sugestão do modelo de relatório foi trabalhar inicialmente com os tamanho de vetores 25, 100, 1000 e 10000. Entretanto, isso gera um gráfico com poucos detalhes.



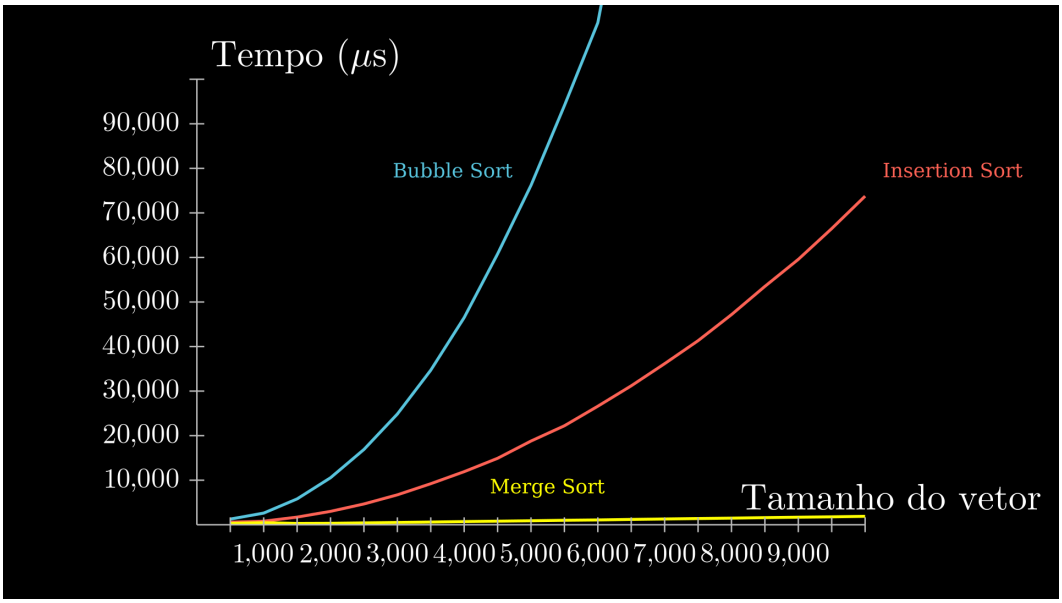
Já é possível observar tendência dos vetores: Bubble é o pior e Merge claramente é o melhor.

Novos gráficos

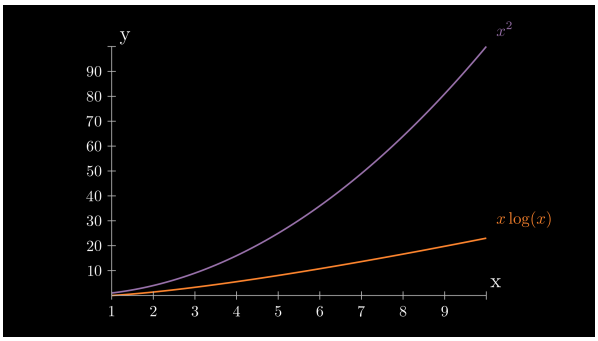
Portanto, outros gráficos com maior definição podem ser criados.

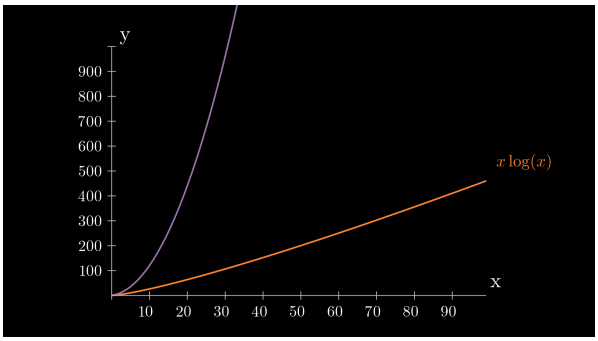


Para tamanhos de vetores muito pequenos, os gráficos podem parecer brigar a primeira vista, mas logo se definem, como dito acima.



Um gráfico com tamanhos de vetor até 10000. Nele, as funções estão bem definidas.





Os dois gráficos acima mostram funções x^2 e xlog(x) nos intervalos [0,10] e [0,100], respectivamente. Ao comparar essas funções com o gráfico das funções dos métodos de ordenação, percebe-se a semelhança entre Bubble, Insertion e a função quadrática, e entre Merge e a função logarítmica. A análise assintótica foi bem feita nesse caso, pois de forma simples, demonstrou a complexidade do algoritmo.

No podium dos métodos de ordenação, Merge é ouro com sobra e o Insertion fica com a prata. Mas...

Por que Bubble e Insertion não possuem o mesmo gráfico se suas funções de eficiência são iguais?

Simples.

Apesar de os dois serem $O(n^2)$, as constantes e coeficientes não são considerados nessa notação. Assim, o Bubble, que possui coeficientes de n maiores que o Insertion, cresce muito mais que seu companheiro.

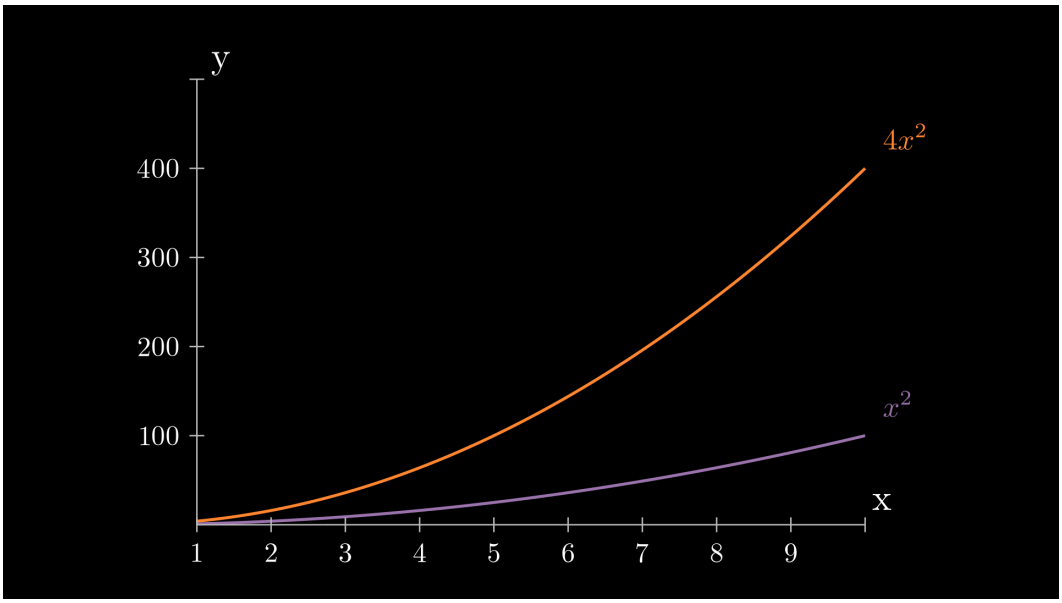


Gráfico das funções 4x² e x²

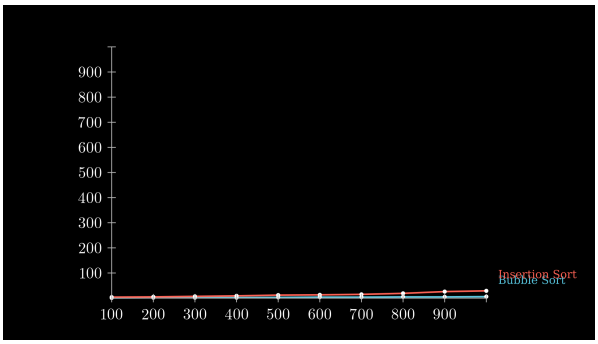
No gráfico acima, as curvas representam funções de mesmo grau, porém o coeficiente do termo faz com que um dos dois cresça muito mais rápido a curto prazo.

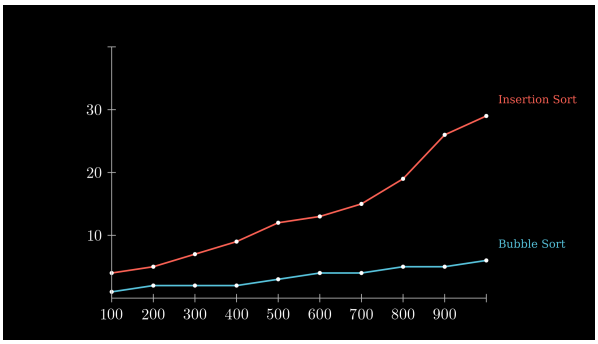
Melhor e pior casos

Para análise dos melhor e pior casos é importante notar que o Merge Sort é $\Omega(n) = O(n)$, 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 Insertion e Bubble.

O melhor caso: vetor ordenado

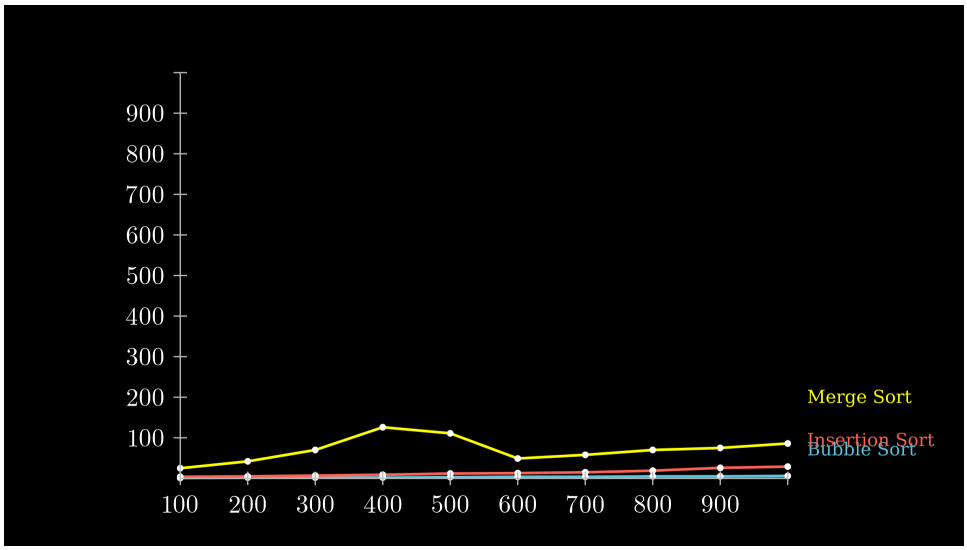
Quando o vetor já está ordenado, basta uma passagem dos métodos mais simples para perceber o resultado e abortar a função. Assim, como esperado da parte de metodologia, as funções $\Omega(n)$ tendem a ser mais rápidas.





Ao diminuir a escala em y é possível analisar melhor o que acontece no gráfico

No melhor caso, o Bubble foi melhor que Insertion. Mas como observado no primeiro gráfico, a diferença é ínfima



Adição do Merge

Com o Merge no gráfico, nota-se que sua eficiência em vetores quase ordenados não é boa.

O pior caso: vetor inversamente ordenado

Ou vetor ordenado de forma decrescente. Enfim.

Espera-se que as duas funções sejam muito ruins agora, já que terão que percorrer o vetor diversas vezes.

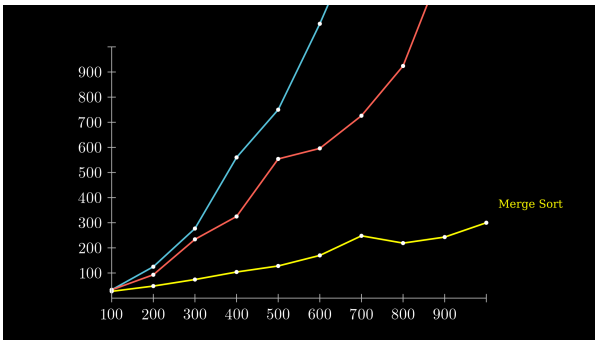
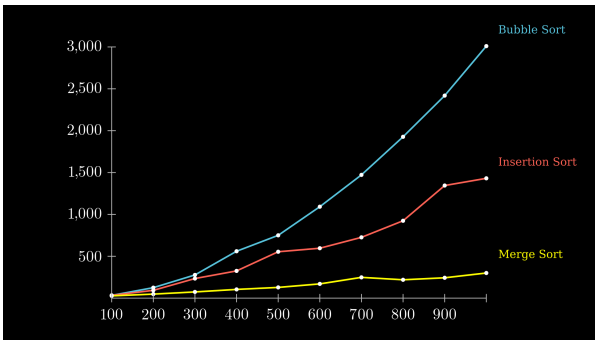


Gráfico das funções no pior caso



Novamente, é necessário mudar a escala (dessa vez aumentar) para análise

O merge sort performa bem de qualquer forma. Já os outros métodos mostrados não são tão efetivos para seus piores casos.

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.

Em suma, é possível observar que o Bubble Sort é completamente inútil. Já o Insertion Sort pode ser utilizado em casos de, por isso o nome do método, inserção, isto é: vetores quase ordenados, os vetores que estão recebendo novos valores. Finalmente, entre os 3 algoritmos analisados, Merge é o melhor deles, simplesmente por sua eficiência superior, e deve ser utilizado em todos outros casos, observando sempre a memória disponível.

Referências

- Khan Academy
- Geeks for Geeks
- Merge-Sort with Transylvanian-Saxon (German) folk dance
- Merge Sort Example
- 15 Sorting Algorithms in 6 minutes