Instituto de Ciências Matemáticas e de Computação
Departamento de Sistemas de Computação da Universidade de São Paulo
Av. Trab. São Carlense, 400 - Centro, São Carlos - SP, 13566-590 – Fone (16) 3373-9700

Atividade 2

SSC0951 - Desenvolvimento de Código Otimizado

Alunos:
Felipe Guilermmo Santuche Moleiro - 10724010
Matheus Tomieiro de Oliveira - 10734630

OUTUBRO/2021

Introdução

Neste trabalho, nosso objetivo é testar quatro diferentes tipos de algoritmos de ordenação e analisar sua eficiência utilizando a ferramenta **gprof**.

Desenvolvimento

Para as análises foi escrito um código em C com os quatro algoritmos de ordenação modularizados em funções, são eles: **bubble sort**, **counting sort**, **merge sort** e o **quick sort**.

O interesse é analisar o tempo gasto em cada uma dessas funções de ordenação, e analisar se seus tempos são condizentes com suas complexidades. Lembrando que as complexidades de tempo de cada uma delas são:

bubble sort: $O(N^2)$, N é o tamanho do vetor
merge sort: O(NlogN) , N é o tamanho do vetor
quick sort: O(NlogN) , N é o tamanho do vetor
counting sort: O(N+K) , N é o tamanho do vetor e K é o intervalo entre o maior e menor elemento do vetor

E as complexidades de espaço são:

bubble sort: O(1)
merge sort: O(N) , N é o tamanho do vetor
quick sort: O(1)
counting sort: O(K) , K é o intervalo entre o maior e menor elemento do vetor

Na main, é definido um vetor de ***n*** posições, e em cada posição será sorteado um número aleatório que a ela será atribuído. É necessário, a cada ordenação ao vetor, limpar a cache entre os processos, uma vez que o lixo presente ali poderia atrapalhar durante a mensuração dos resultados.

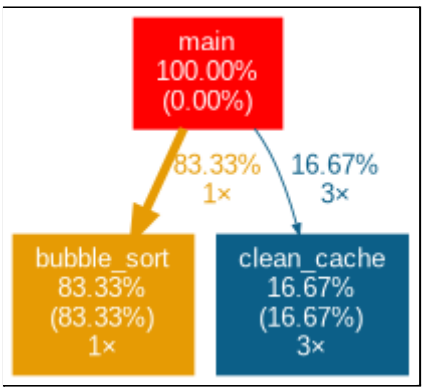
O código final então será uma função de criar/copiar o vetor, uma chamada para a função de ordenação, e uma chamada para limpar a cache, repetindo esses passos para cada uma das diferentes funções de ordenação.

Resultados

Primeira execução

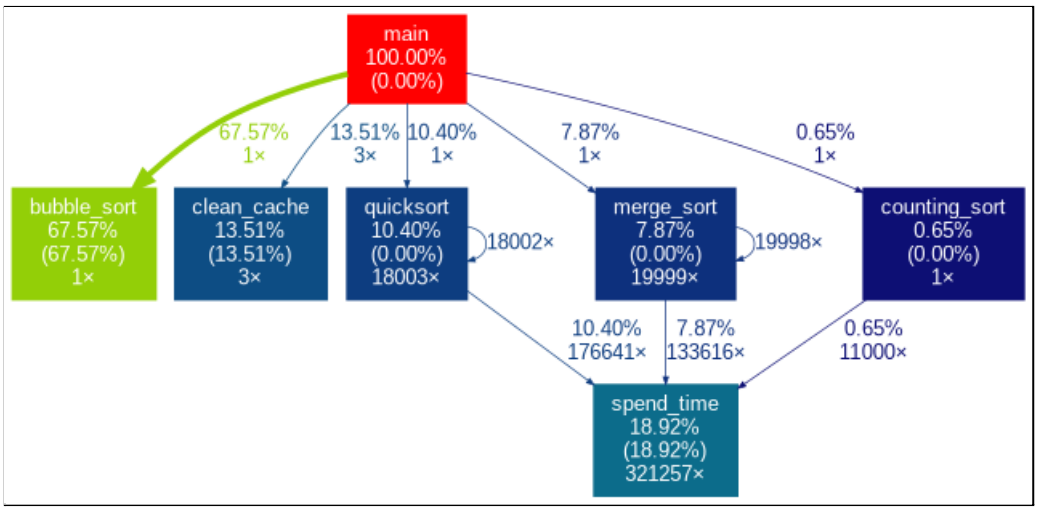
Para a primeira execução, vamos rodar o código normal para um vetor de tamanho 10000 e com intervalo(K) de 1000 e ver o que temos:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 82.03      0.25     0.25        1   246.09   246.09  bubble_sort
 16.74      0.30     0.05        3    16.74    16.74  clean_cache
  0.00      0.30     0.00        4     0.00     0.00  cpy_vet
  0.00      0.30     0.00        1     0.00     0.00  counting_sort
  0.00      0.30     0.00        1     0.00     0.00  merge_sort
  0.00      0.30     0.00        1     0.00     0.00  quicksort
```



Podemos ver pelos valores gerados pelo gprof que o tempo de execução do programa é principalmente gasto no bubble sort e limpando a cache, enquanto não gastamos praticamente nada de tempo nas outras funções. Isso se dá pelo fato de que o bubble sort tem complexidade N^2, enquanto as outras funções tem complexidade NlogN e N+K, e por este motivo escalam mais rapidamente o tempo conforme aumentamos o tamanho do vetor. Para contornar um pouco isso e termos uma visualização de tudo, vamos adicionar uma função que passa o tempo fazendo nada para incluir na visualização.

Utilizando função spend time e contando número de operações

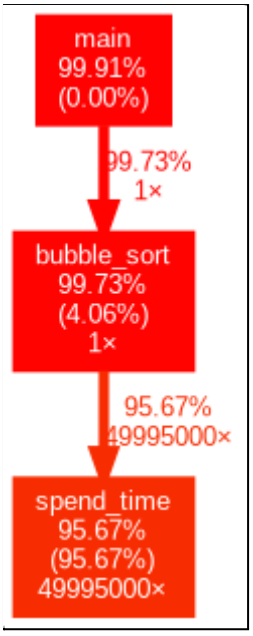


Agora é possível ver todas as funções na visualização, e apesar da questão do tempo de execução não fazer mais tanto sentido, o que fizemos foi tentar chamar a função spend time para toda operação realizada dentro de cada função ao invés de para cada chamada de função. Então é possível ver como o counting sort realizou 1100 chamadas para a função spend time, 1000 percorrendo o vetor de tamanho N e 1000 percorrendo o vetor de tamanho 1000 do intervalo, o que é condizente com sua complexidade de O(N+K).

O mesmo vale para o mergesort, que chamou spend time 133616 vezes que é aproximadamente 10000*log(10000) ≅ 132877 , ou seja, podemos confirmar a complexidade desses algoritmos contando as operações realizadas.

O quicksort chamou a spend time 176641 vezes, o que é um pouco mais que NlogN, mas isso se dá pelo fato de que o quicksort é um algoritmo que tem complexidade média de NlogN mas no pior dos casos ele é um algoritmo N^2, tudo isso varia dependendo da heurística da escolha do pivô dentro do algoritmo (no nosso caso escolhemos o último elemento do vetor). Se tivéssemos uma heurística perfeita que encontra sempre o pivô que estaria exatamente no centro do vetor, teríamos o mesmo número de chamadas que o mergesort, mas na prática essa escolha nunca é perfeita mas é em geral boa o bastante para manter a complexidade média perto de NlogN.

Por fim, será analisado o número de execuções do bubble sort, realizando uma execução separada.(Ela foi feita separadamente pois se incluímos a spend time dentro do bubble sort o tempo gasto nas outras funções acaba sendo ignorado).



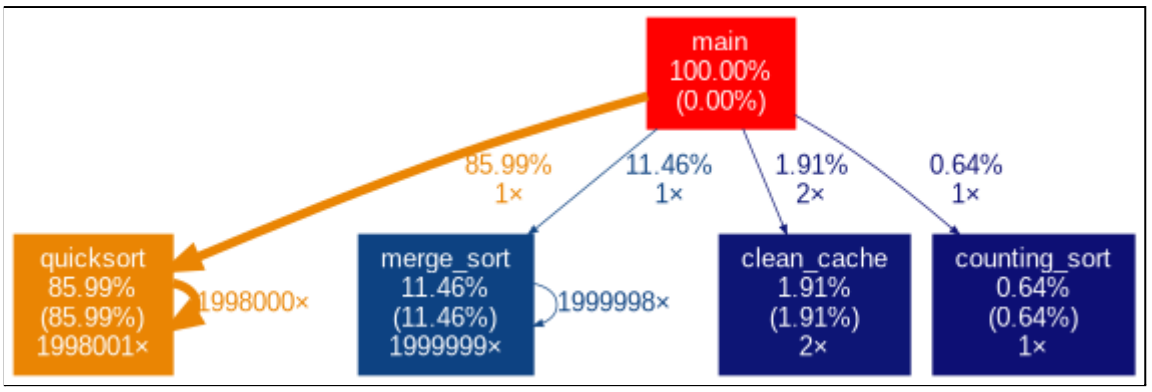
Aqui podemos ver que o número de chamadas foi de 49995000, que é exatamente N*(N-1)/2 , que é o número de operações que o bubble sort realiza pois ele realiza operações da seguinte forma, primeiro percorre o vetor inteiro até o penúltimo elemento(N-1), depois até o antepenúltimo item(N-2), e continua assim até o vetor unitário(1), ou seja, o número de operações é (N-1) + (N-2) + … (1) , que utilizando a fórmula de soma de PA é igual a N*(N-1)/2.

Desta forma, podemos observar como a contagem de operações é compatível com a complexidade desses algoritmos.

Comparando os algoritmos NlogN e N+K

Já vimos como o bubble sort acaba sendo o que tem a pior execução em questão de tempo gasto, o que faz sentido já que possui complexidade N^2 e o tempo de execução cresce rapidamente comparado aos outros algoritmos. Por este motivo vamos ter essa seção do relatório dedicada aos algoritmos de menor complexidade e voltaremos a falar do bubble sort na análise final de tempo de execução.

Vamos fazer algumas execuções agora para analisar a eficiência de cada algoritmo. O que podemos esperar pelas análises anteriores é que o mergesort e o quicksort executem mais lentamente que o counting sort, desde que o K não for extremamente maior que N. Por este motivo, vamos rodar um caso em que N = 10^6 e K=10^3:

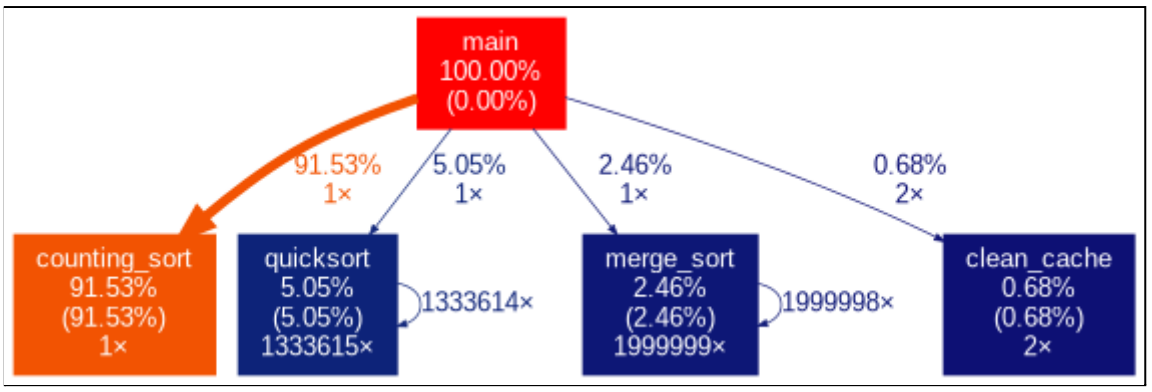


Bom, aqui podemos ver como o quicksort é onde nós gastamos a maior parte do tempo dessa execução, em segundo temos o mergesort e por último temos o counting sort, assim como prevemos. Aqui estão os tempos em cada função:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 86.28      1.35     1.35        1     1.35     1.35  quicksort
 11.59      1.53     0.18        1     0.18     0.18  merge_sort
  1.93      1.56     0.03        2     0.02     0.02  clean_cache
  0.64      1.57     0.01        1     0.01     0.01  counting_sort
  0.00      1.57     0.00        3     0.00     0.00  cpy_vet
```

Aqui podemos nos perguntar porque o quicksort foi pior que o mergesort, e o motivo é o que foi discutido anteriormente, sobre em alguns casos específicos o quicksort tender a realizar mais operações que o mergesort, entretanto, ele tem certas vantagens sobre o mergesort. Enquanto o mergesort precisa de memória auxiliar do tamanho do vetor original(O(N)), o quicksort não necessita de memória adicional proporcional ao vetor(O(1)), o que pode fazer muita diferença para vetores grandes em máquinas que não tem memória suficiente para alocar uma cópia do vetor. Além disso, para vetores maiores, essa questão de não ter um vetor auxiliar alocado na memória pode dar uma performance melhor devido a ter menos acessos à memória e assim um melhor acesso à cache dos dados. Vamos ver isso mais a fundo na análise de tempo, pois se considerarmos todo o tempo que a máquina gasta acessando a memória extra(o que causa vários miss na cache) isso irá mostrar que em média o quick executará mais rápido.

Vamos agora aumentar o valor de K para demonstrar problemas do counting sort. Rodando para o caso de N = 10^6 e K=MAX_INT(2x10^9):



Nesse caso podemos ver como o counting sort é terrivelmente mais lento que os outros algoritmos. Isso se dá pelo fato de que o counting sort é impactado pelo intervalo dos valores no vetor(variável K), e quando K é muito maior que N, o counting sort sofre perda de performance(pois tem que percorrer um vetor de tamanho K) enquanto os outros algoritmos não se importam com o intervalo de valores do vetor, mas apenas com a quantidade de elementos. Aqui estão os valores obtidos em questão de segundos:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 91.97      6.70     6.70        1     6.70     6.70  counting_sort
  5.10      7.08     0.37        1     0.37     0.37  quicksort
  2.41      7.25     0.18        1     0.18     0.18  merge_sort
  0.69      7.30     0.05        2     0.03     0.03  clean_cache
  0.28      7.32     0.02        3     0.01     0.01  cpy_vet
```

Além disso, neste caso com K = MAX_INT(2x10^9) foi necessário alocar um vetor de tamanho 2x10^9 inteiros, que é equivalente a 8GB de memória, o que foi o limite do que o computador que executa esse código foi capaz de executar, e muito provavelmente algum tipo de pagamento da memória em disco(swap) ocorreu. Ou seja, podemos ver que se fossemos ordenar um vetor de longs em que o K=9x10^18, já seria impossível ter memória suficiente para isso. Além disso vale notar que o tipo de dados desse vetor auxiliar tem que ser capaz de contar até o valor máximo N, ou seja, não poderíamos ter um vetor de unsigned chars como contador nesse caso pois só conseguimos contar até 255, e o valor de N vai até 10^6.

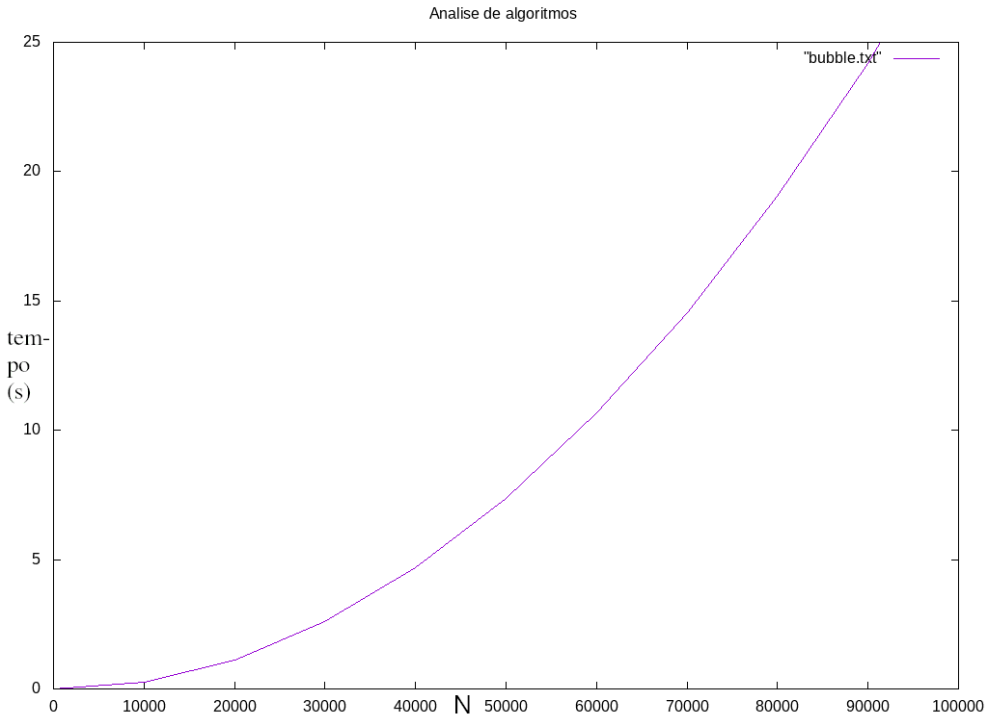
E por último, outro problema deste algoritmo é que ele só funciona para valores inteiros. Existem variações dele para ordenar outros tipos de dados como por exemplo pontos flutuantes, entretanto ele acaba tendo uma complexidade maior que o counting sort pois ele é impactado pela precisão do número e o número de casas decimais existentes antes e depois da vírgula e muitas vezes não vale a pena comparado com os algoritmos NlogN.

Então podemos ver que para casos de vetores inteiros com valores com pouca variação(K pequenos), o counting sort é muito mais eficiente, mas para vetores com muita variação em que K é muito maior que N, ele pode ter problemas.

Análise de Tempo De Execução

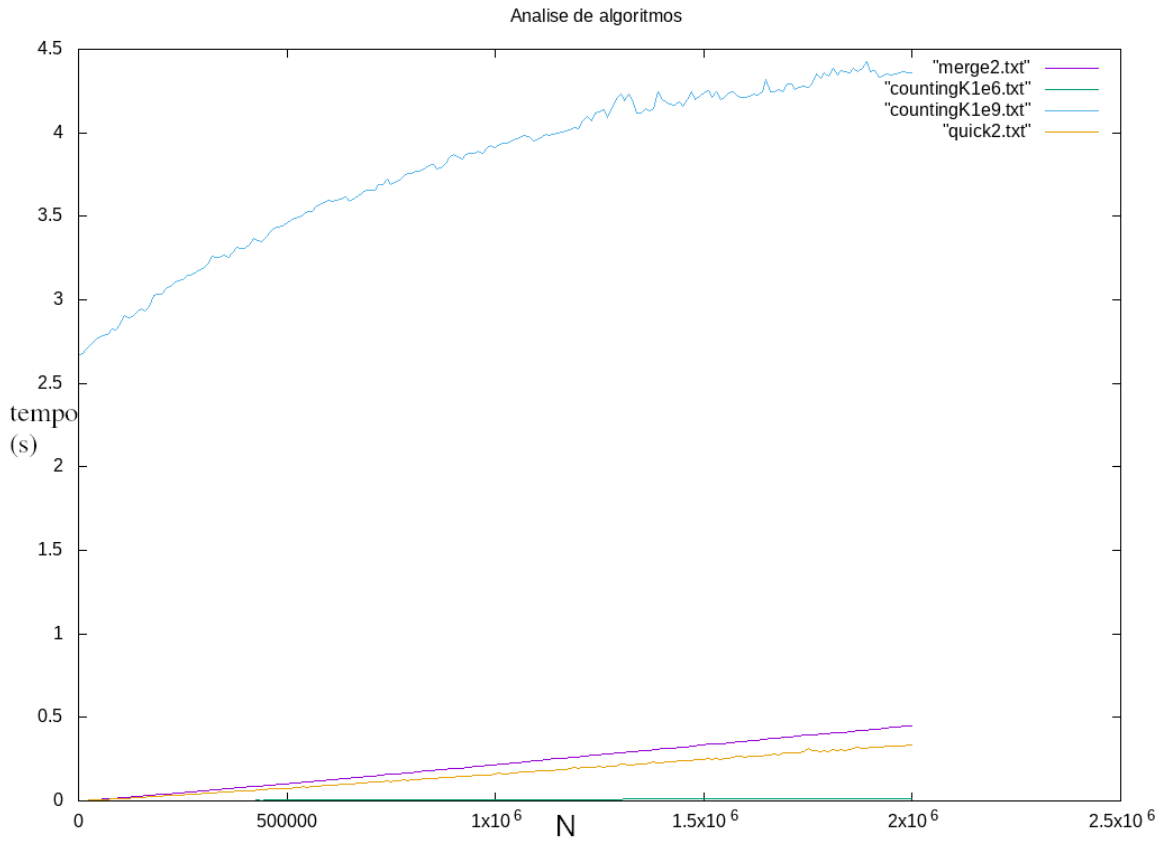
Agora vamos mostrar alguns gráficos analisando o tempo de execução. Todos os gráficos a seguir têm o eixo x representando o valor de N e o eixo y representando o tempo de execução.

Primeiramente temos o bubblesort, que como sabemos é N^2, portanto esperamos ver esse comportamento ao aumentarmos o N. Fazendo diversas execuções em série para diversos Ns nós temos:



Podemos ver claramente o comportamento quadrático do algoritmo expressado no gráfico, e como o tempo de execução cresce muito rapidamente. A visualização do bubble sort comparada com os outros foi separada pois o tempo que ele levava para executar foi muito maior, e a visualização dos outros algoritmos era impossível.

Vamos analisar agora os outros algoritmos:

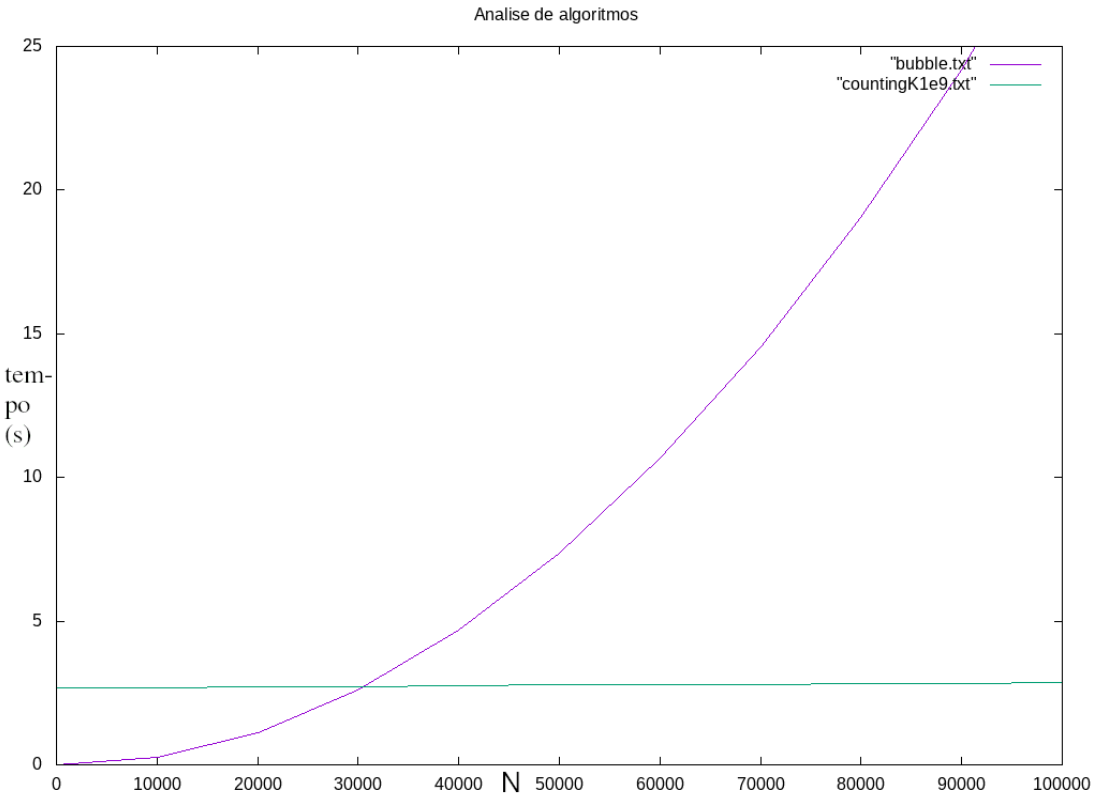


Aqui temos o mergesort, quicksort e dois countings(o nome merge2 e quick2 se referem apenas a implementação interna feita, já o countingK1e6 significa o counting sort com k=10^6, e countingK1e9 significa counting sort com k=10^9).

Podemos ver que neste caso temos tempos muito melhores que o bubble, mas ainda temos um algoritmo que demora muito mais que os outros, este é o counting sort com k=10^9. Devido ao enorme tamanho do vetor auxiliar de contagem que deve ser percorrido, a sua execução demora muito mais que as outras, tanto pelo fato de ter que iterar por diversos elementos como pelo fato da cache não conseguir carregar o vetor auxiliar inteiro, e ser necessário fazer acessos a partes diferentes do vetor e carregá-los na cache.

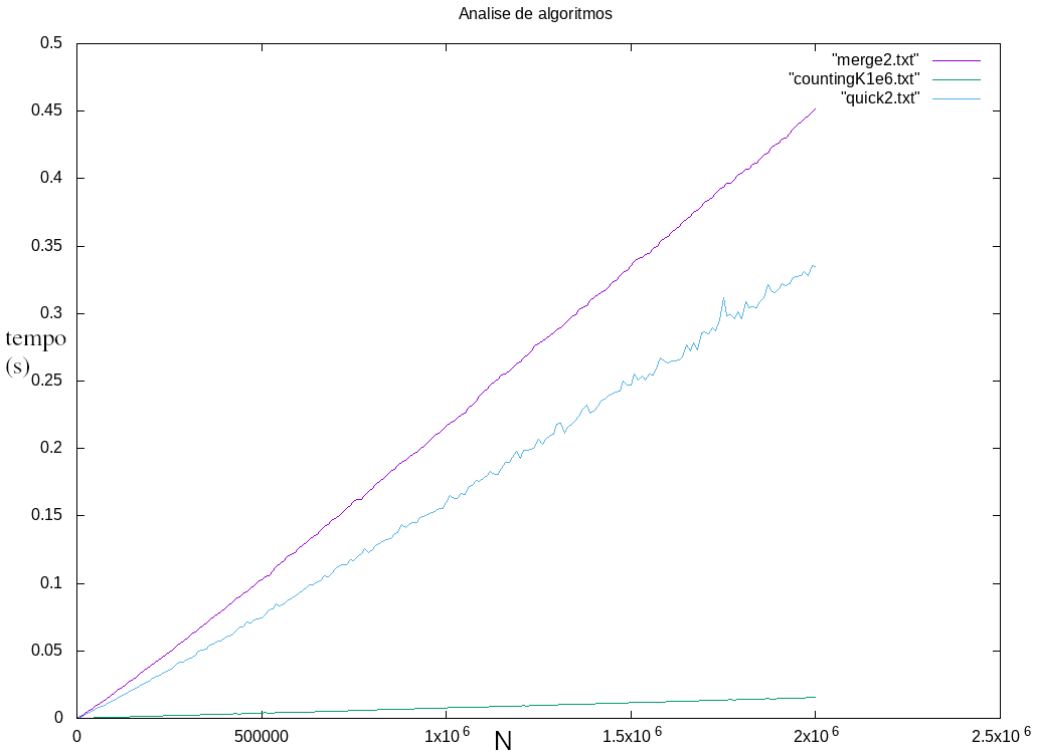
Nós podemos ver que a execução do counting sort não é tão afetada pelo N quanto ela é pelo K neste exemplo, pois o outro counting sort com k=10^6 está com valores de tempo tão pequenos que quase não conseguimos ver.

Uma comparação que podemos fazer é entre o counting sort de k=10^9 com o bubble sort, e desta forma nós temos:



Aqui podemos ver como o bubble é melhor que o counting sort com K grande até por volta de 30000 no N, após isso o bubble explode, enquanto o counting sort com K grande fica relativamente estável. Entretanto, estes dois ainda são piores que as outras opções que temos.

E analisando agora somente os algoritmos mais eficientes, nós temos:



Aqui podemos notar claramente que o counting sort é o algoritmo mais eficiente para casos em que o valor de K não é tão grande. E uma coisa interessante nesta análise é que o mergesort demorou mais tempo para executar em geral do que o quick. Isto se dá pelo fato de que apesar do quicksort ter em média mais operações a serem realizadas pelo que foi discutido anteriormente sobre o algoritmo ser apenas no caso médio NlogN, o fato dele não precisar alocar memória extra ajuda muito no tempo de execução, pois além do tempo perdido alocando memória, com um vetor auxiliar e mais memória sendo necessária para executar o algoritmo, necessitamos acessar mais regiões diferentes de memória, o que consome espaço da cache levando a mais misses em geral, o que faz com que o quick em média seja mais rápido.

Conclusões

Podemos ver como a ferramenta gprof pode ser muito útil para analisar regiões consideradas como códigos quentes, e desta forma descobrir que partes do código estão tomando mais tempo de execução para desta forma considerar maneiras de otimizar estas regiões em específico.

Já sobre o código analisado em específico, foi interessante relembrar os diversos algoritmos de ordenação, e principalmente a questão de quando vale a pena utilizar um algoritmo NlogN como o quicksort e o mergesort, e quando utilizar um algoritmo N+K como o counting sort.

Obs: Todo o código utilizado encontra-se alocado no GitHub: **https://github.com/FelipeMoleiro/MateriaCodigoOtimizado**