­Este projeto possui como objetivos evidenciar a compreensão do aluno sobre o tema Análise de algoritmos e sobre análise assintótica de algoritmos. Além disso o projeto visa proporcionar uma experiência para a prática neste tema.

O trabalho pode ser realizado em grupos de até 3 alunos e deve ser entregue um relatório com:

a) um breve resumo sobre os conceitos gerais de análise assintótica de algoritmos (contendo no mínimo uma e no máximo três páginas);

b) a descrição de um algoritmo que possa ser analisado com base nos recursos de análise assintótica de algoritmos;

c) a análise realizada sobre o algoritmo indicado.

O trabalho pode ser baseado em artigos científicos que descrevem este tipo de utilização, bem como pode ser gerado a partir de uma situação particular de interesse do aluno.

Os critérios de correção considerados serão: a clareza na descrição dos conceitos e do problema, juntamente com a correção da análise realizada.

a) um breve resumo sobre os conceitos gerais de análise assintótica de algoritmos (contendo no mínimo uma e no máximo três páginas);

Um determinado algoritmo pode ter diferentes tempos de execução de acordo com o ambiente no qual está sendo rodado. Se for em um “PC gamer”, por exemplo, possivelmente vá ser mais performático do que em um Raspberry Pi. Mas, além do ambiente, ainda existe a linguagem. Um código escrito em, por exemplo, Java e adaptado para Python pode também apresentar dois tempos de execução diferente, dado a natureza das linguagens, por mais que apresentem a mesma linha de raciocínio, uma vez que a implementação vai ser ligeiramente diferente.

Dessa forma, afim de julgar se um algoritmo é rápido ou não, a maneira encontrada pelos teóricos da computação foi remover todas essas variáveis estrangeiras da conta, e se apegar única e exclusivamente ao código e a como ele está implementado. Isso é a Análise Assintótica de Algoritmos.

Segundo o Wikibooks, análise assintótica é: *(…) a way of expressing the main component of the cost of an algorithm, using idealized (not comparable) units of computational work.* Como descrito anteriormente, é uma forma de julgar a eficiência de um algoritmo, independente dos recursos que o cercam (processador, memória, ping, etc).

Essa análise consiste em descobrir a ordem de grandeza do tempo de processamento do algoritmo, dado o crescimento da entrada. Mas matematicamente falando, como isso funciona?

Do ponto de vista matemático, análise assintótica seria analisar uma função, como por exemplo f(n) = n+1 ou f(n) = n²+2, para casos onde n tende ao infinito. Nesses casos, a única parte da função que realmente importa é aquela que dita seu crescimento. Por exemplo: na função f(n) = n+1, com n = infinito, o 1 torna-se completamente irrelevante. Em outra palavras, temos que sua ordem de crescimento é n. A mesma coisa vale para f(n) = n² \* 7382 + 3. Aqui, a constante que multiplica n e o valor que soma, tornam-se praticamente irrelevante para valores massivos de n, fazendo com que sua taxa de crescimento seja da ordem n².

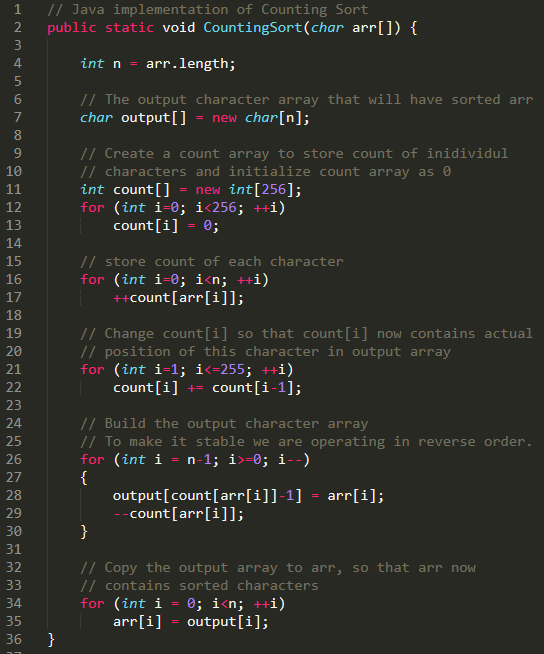
Aplicando isso à computação, nos importa medir a ordem de crescimento de execução de um algoritmo baseando-se em três casos (lembrando que nos três casos a quantidade de valores de entrada, isso é, o valor de n, está tendendo ao infinito):

1. ordem O, o pior caso possível
2. ordem Ômega, o melhor caso possível
3. ordem Theta, o caso médio

Os três títulos são auto explicativos, e serão abordados com um exemplo mais detalhado nas próximas seções do trabalho.

b) a descrição de um algoritmo que possa ser analisado com base nos recursos de análise assintótica de algoritmos;

O algoritmo escolhido pelo grupo é o Counting Sort, que está implementado abaixo:



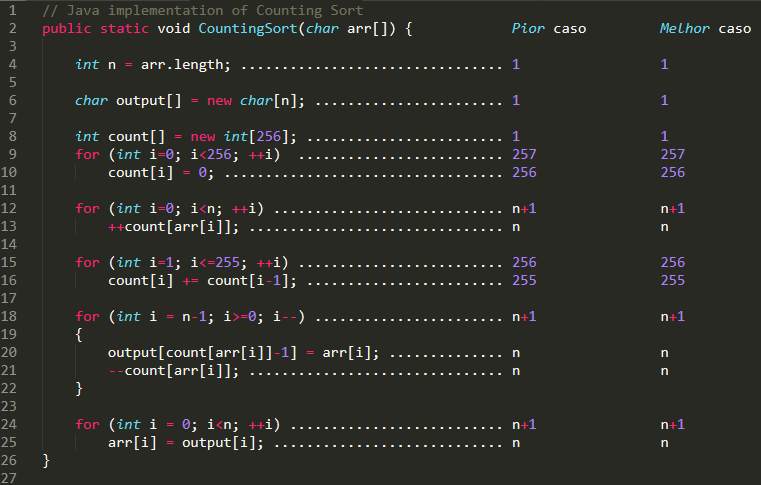
Descrição de funcionamento do algoritmo em steps:

1. criar uma array extra chamada OUTPUT cujo tamanho é igual ao tamanho da array de entrada. Aqui, já identificamos que a complexidade de espaço desse

algoritmo é de no mínimo, O(n), pois exige um espaço extra na memória de igual tamanho à array de entrada. Essa array vai ser onde guardaremos o nosso resultado. Isso acontece nas linhas 4 e 7.

1. Criar uma terceira array, agora chamada COUNT, de tamanho 256, não coincidentemente os mesmos 256 da tabela ASCII. Essa array vai servir para contar a quantidade de vezes que cada valor da ASCII se repete dentro da array que se deseja ordenar. Além disso, essa array é inicializada com todas suas entradas zeradas. Nesse momento, a complexidade de espaço do algoritmo subiu para O(n) + C, onde C representa a array COUNT, pois é uma constant. Isso acontece nas linhas 11~13.
2. agora o algoritmo vai iterar em todas as entradas da array que se deseja ordenar, e para cada uma das entradas, vai somar +1 no código da tabela ASCII. Por exemplo: se a array fosse {1, 1, 1}, teremos COUNT[1] = 3. Isso significa que para o valor ASCII ‘1’, temos três entradas correspondentes no array que gostaríamos de ordenar. Isso acontece nas linhas 16 e 17.
3. Agora soma-se o valor de cada index de COUNT com o valor da posição anterior, contando de 1 até 255. Isso vai fazer com que o index de COUNT indique em que posição o valor deve ser populado na array OUTPUT. Por exemplo, se tivermos COUNT[5] = 3. Significa que devemos passar COUNT[5] para OUTPUT[3]. De maneira grossa, podemos dizer que é aqui que o sorting acontece, pois ao final dessa lógica, a array COUNT já possui todas as posições esperadas, de maneira ordenada, de todas as entradas da array que se deseja ordenar. Isso acontece nas linhas 21 e 22.
4. Agora, passamos os valores para a array OUTPUT. Faz-se da seguinte forma: o valor ASCII do index específico da array que se quer ordenar é verificado dentro da array COUNT. Essa array, como descrito no passo d), armazena o index que o valor deve ser impresso dentro da array OUTPUT. Por exemplo: arr[I] = 10, count[10] = 15, output[15] = arr[I] = 10. É como se COUNT se comportasse como um hash, nos entregando em que index o determinado valor deve ir.
5. Por fim, atribuem-se os valores de OUTPUT, agora ordenados, à array ARR, que é a que veio por parâmetro e deve ser ordenada.

c) a análise realizada sobre o algoritmo indicado.



T(n) melhor caso =

T(n) pior caso =

Conclusão: como podemos constatar, esse algoritmo de ordenação possui complexidade de tempo na ordem de N, tanto para o melhor quanto para o pior caso. Inclusive, o tempo de execução deve ser exatamente o mesmo, seja com o algoritmo já ordenado ou ainda não ordenado, uma vez que todas as iterações sob o array se dão nele inteiro e em valores constantes, como 255 ou 256 vezes.