[Instituto Superior Manuel Teixeira Gomes](http://www.ismat.pt/)

Licenciatura de Engenharia Informática, 2º Semestre

*Algoritmia*

Prof. Doutor António dos Anjos

Trabalho Prático de Avaliação nº 1

Trabalho elaborado por:

Carlos Manuel Pacheco Soares, Nº 21103408

Índice

Introdução2

O Problema3

O Algoritmo4

O Programa6

Otimização do Programa7

Soares Counting Sort Vs. Quick Sort8

Conclusão11

Introdução:

No âmbito da disciplina de Algoritmia foi proposto a implementação de um algoritmo de ordenação de valores inteiros, não decrescentes, nosso ou já existente com o objetivo de ordenar *arrays* com grandes dimensões o mais rápido possível. Para tal é pedido a criação de um programa com pelo menos três funções: uma que gera o *array* com valores aleatórios, uma com o algoritmo de ordenação e outra que irá validar se o algoritmo está ordenado.

Para este problema criei um algoritmo irá procurar a distribuição dos dados e depois os coloca na devida ordem. Em que mais tarde encontrei uma semelhança com um outro algoritmo chamado *“Counting Sort”*, e consequentemente chamei o meu de *“Soares Counting Sort”*, visto que existe também um *“Tim Sort”*.

O Problema:

Existem várias razões para se ordenar uma sequência de um tipo de dados. Uma delas seria para lhes poder aceder de forma mais fácil e rápido. Isto numa situação de análise ou para tomar uma decisão de negócio pode ser crucial. O problema é quando a quantidade de dados são demasiados para se ordenar manualmente.

Para o resolver começaram-se a criar diversos algoritmos para podermos utilizar os computadores em nosso favor. Isso funciona geralmente muito bem para situações em que temos poucos dados. Mas qual será a situação se tivermos por exemplo cem milhões de dados diferentes?

Nesta situação, como na maioria delas, já irá contar muito a rapidez do algoritmo utilizado, pois se tivermos a espera da ordenação de dados que irão ser utilizados para uma decisão de negócios e se esta ordenação demorar, por exemplo, um ano, provavelmente não irá servir de muito.

No entanto, existem diferentes tipos de dados a ordenar e de diferentes formas cada uma com um intuito diferente. Mas irei explora-los a todos. Somente uma situação relativamente simples que foi o proposto, ou seja, a ordenação de grandes números de dados numéricos. Estes dados são números inteiros do qual sabemos à partida o valor máximo e o valor mínimo.

Na criação do algoritmo para o problema proposto tentei melhorar o algoritmo criado de forma otimizá-lo o máximo possível para estas condições específicas.

O Algoritmo:

O algoritmo que eu criei e mantive com escolha para este problema foi porque o mesmo é muito eficiente. Com eficiente quero dizer que é rápido, tem uma boa gestão de recursos e tem resultados corretos. Para além disso é muito fácil de explicar, entender e implementar, o que o torna uma excelente opção de utilização. No entanto, como em qualquer situação este algoritmo também tem contrapartidas.

Para começar só ordena números inteiros, o que para a nossa situação é o suficiente mas o que poderia pedir alterações ou até a excluir a sua utilização em outros tipos de dados a ordenar. Isto no entanto o coloca em vantagem com alguns, como é o caso do *Counting Sort*, que só consegue ordenar números inteiros positivos. Necessita que é conhecido o valor mínimo e o máximo dos itens mas isso é um problema que se necessário é facilmente resolvido, com um algoritmo auxilia de busca, que nos retorna valor mais alto e o mais baixo. Por fim, poderá para algumas situações ocupar muita memória do computador, mas esse tipo análise não será aprofundada para este trabalho, visto que não é pedido para o mesmo.

No entanto, poderei dizer, relativamente a análise da utilização de memória, que por usar *arrays* de comprimentos ‘n’ e ‘k + 1’, em que ‘n’ é o número de elementos diferentes a ordenar e ‘k + 1’ o número diferente de elementos que poderão ser necessários para a ordenação, pois ‘k +1’ é igual ao número de diferentes valores possíveis entre o intervalo limitado pelo máximo e o mínimo, então a memória que o algoritmo irá utilizar será . O que nos dá a concluir que a utilização da memória será bastante eficiente se ‘k’ for pequeno.

Para poder prosseguir a análise do algoritmo terei primeiro o explicar. O algoritmo irá criar um *array* auxiliar com o número diferente de itens possíveis no dado intervalo de o mínimo ao máximo inclusive, neste *array* cada item corresponde a um valor e será utilizado para contar o número de ocorrências de cada valor, cada vez que aparece no *array* a ordenar um valor o item correspondente no *array* auxiliar é incrementado. De seguida, como queremos ordenar os valores de ordem não decrescente e o primeiro valor do *array* auxiliar corresponde ao mínimo e o último ao máximo é só coloca-los por ordem e pelo número de ocorrências, ou seja, se um dado valor aparecer duas vezes e for o terceiro mais baixo é colocado a seguir do último valor, segundo mais baixo, duas vezes. Repetindo se este processo até se ter colocado por fim o valor máximo as vezes que ela ocorre.

Em pseudocódigo, o algoritmo representa-se da seguinte forma:

SoaresCountingSort (Array, n, max, min){

Array[[1 ... n] //É introduzido pelos argumentos (Colocado aqui somente para facilitar a compreensão do algoritmo).

ArrContador[1 ... (max – min + 1)] //É um array auxiliar (Contador), elementos inicializados a zero.

for i 1 to (max – min + 1)

do ArrContador[i] 0

for i 1 to n

do ArrContador[Array[i] – min + 1] ArrContador[Array[i] – min + 1] + 1

j 1

for i 1 to n

do if (ArrContador[j] > 0)

do Array[i] j + min – 1

ArrayContador[j] ArrayContador[j] – 1

else

do j j + 1

i i – 1

}

A análise assintótica do algoritmo torna-se extremamente fácil, temos três ciclos de que dois são obviamente de tamanha ‘n’ e ‘k’. O terceiro é de tamanho ‘n + k - 1’, no pior caso, caso tenha de passar pelos elementos todos que contou. Mas se só existe um elemento e este for o valor mais baixo possível, repetido ‘n+1’ vezes, o ciclo será de tamanho ‘n’, que será o melhor caso.

Então para o pior caso:

Para o melhor caso a análise assintótica será muito semelhante:

Como o melhor e o pior caso são iguais, não será necessário calcular o caso intermédio que será igual.

Em suma, sabemos que o algoritmo que criei tem para a análise de memória como para a análise assintótica, no pior, melhor e no caso intermédio .

O Programa:

Para a implementação do algoritmo, criei um programa com quatro funções, as três essências para o trabalho e um para auxiliar o controle dos parâmetros. Irei explicar as funções de forma simples e breve.

* int isnum(char \* str)

Controla se a *string* introduzida é um número inteiro, devolve um inteiro ‘1’ se for verdade e ‘0’ se falso. É a função auxiliar do controle de parâmetros.

* int \* genArray(int n, int min, int max)

Gera um *array* de ‘n’ valores de ‘min’ a ‘max’, devolve o apontador para o *array* gerado.

* void SoaresCountingSort(int \* A, int n, int min, int max)

Ordena o *array* de inteiros ‘A’, com ‘n’ elementos com valores de ‘min’ a ‘max’ de forma não decrescente.

* int ControlSort(int \* A, int n)

Verifica se o *array* de inteiros ‘A’, com ‘n’ elementos está ordenado de forma não decrescente. Devolve um inteiro ‘1’ se tiver ordenado e ‘0’ caso contrário.

Para além destas funções ainda coloquei algumas verificações para que os parâmetros introduzidos aceites fizessem sentido e para seguir a limitação proposta pelo professor, de que para um *array* de ‘n’ valores pode ter um valor máximo de ‘n’ e um valor mínimo de ‘-n’.

Relativamente às livrarias utilizadas somente utilizei três das livrarias *standart*, o “stdio.h”, o “stdlib.h” e o “time.h”.

Otimização do Programa:

O objetivo é criar um programa que ordenasse um *array* o mais rápido possível a otimização do programa é também extremamente importante. Para tal, baseie-me no autor Paul Hsieh (<http://www.azillionmonkeys.com/qed/optimize.html>), que constata que substituir ciclos “for” por “do while’s”, incrementações por decrementações (porque devida arquitetura de alguns computadores isto se torna mais rápido enquanto no outros pouco importa), “if else” por um simples “if”, várias variáveis auxiliares por uma global e algumas outras técnicas farão grande diferença na rapidez de execução de um programa.

Sabendo isso, para otimização criei primeiro uma condição que é o seguinte se o máximo for igual ao mínimo ou se só houver um elemento no *array*, o mesmo está ordenado. Isto por razões óbvias. De seguida, substitui todos os ciclos que tinha, que por facilidade e hábito de utilização eram ciclos “for” a incrementar por “do while’s” a decrementar. Por fim declarei uma variável ‘i’ global e retirei uma variável auxiliar a cada função.

Isto tudo aumentou em muito a rapidez do programa. Os motivos por trás desta otimização não irei aprofundar, por considerar que isso já se torna um pouco fora do contexto do trabalho.

Soares Counting Sort Vs. Quick Sort:

Por ter sido convictamente confrontado por vários colegas que constatavam que o famoso Quick Sort é o algoritmo de ordenação mais rápido de todos decidiu compara-lo com o meu. Para tal, irei fazer 5 réplicas de cada uma para valores diferentes de ‘n’, com o máximo igual a 10 e o mínimo igual a -10, isto para poder começar com ‘n’ igual a 10. Para este teste irei utilizar a função que pedi ao meu colega Isac Pimpao por confiar nele e achar que ele tenha feito uma boa implementação do Quick Sort.

Os testes foram feitos no meu portátil, um ACER ASPIRE 5741G, com o Windows 7 Home Premium 64 Bits (SP1), no MinGW Shell. As características principais do computador são as seguintes:

* Intel® Core™ i5 CPU M430 (2.27GHz, 3MB L3 cache);
* NVIDIA® GeForce® GT 320M CUDA™ - 1GB (2747 MB TurboCache™);
* 4 GB DDR3 Memory;
* 640GB.

Os resultados foram os seguintes (apesar de sabermos que com um ‘k’ muito pequeno terei uma grande vantagem, relativamente ao Quick Sort, por ter feito mais testes com ‘k’s maiores considerei os seguintes resultados relevantes):

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N: | Soares Counting Sort: | | | | | Quick Sort: | | | | |
| 10 | 0.050s | 0.049s | 0.045s | 0.051s | 0.045s | 0.047s | 0.050s | 0.051s | 0.046s | 0.047s |
| Média = 0.0480s | | | | | Média = 0.0482s | | | | |
| 100 | 0.046s | 0.045s | 0.044s | 0.043s | 0.043s | 0.047s | 0.050s | 0.055s | 0.049s | 0.043s |
| Média = 0.0442s | | | | | Média = 0.0488s | | | | |
| 1.000 | 0.043s | 0.043s | 0.048s | 0.044s | 0.044s | 0.048s | 0.044s | 0.047s | 0.045s | 0.045s |
| Média = 0.0444s | | | | | Média = 0.0458s | | | | |
| 10.000 | 0.048s | 0.049s | 0.042s | 0.046s | 0.053s | 0.049s | 0.043s | 0.045s | 0.052s | 0.044s |
| Média = 0.0476s | | | | | Média = 0.0466s | | | | |
| 100.000 | 0.051s | 0.050s | 0.049s | 0.049s | 0.050s | 0.058s | 0.058s | 0.062s | 0.064s | 0.059s |
| Média = 0.0498s | | | | | Média = 0.0602s | | | | |
| 1000.000 | 0.097s | 0.097s | 0.092s | 0.101s | 0.089s | 0.207s | 0.195s | 0.199s | 0.208s | 0.197s |
| Média = 0.0952s | | | | | Média = 0.2012s | | | | |
| 10.000.000 | 0.490s | 0.498s | 0.494s | 0.498s | 0.493s | 1.753s | 1.773s | 1.758s | 1.749s | 1.744s |
| Média = 0.4946s | | | | | Média = 1.7554s | | | | |
| 100.000.000 | 4.602s | 4.505s | 4.515s | 4.584s | 4.579s | 18.789s | 18.920s | 18.764s | 18.747s | 18.894s |
| Média = 4.557s | | | | | Média = 18.8228s | | | | |

Podemos concluir então que o algoritmo que criei é mais rápido que o Quick Sort para a grande maioria dos casos e como o mesmo é considerado um dos algoritmos mais rápidos existentes até hoje, o Soares Counting Sort também fará parte desse grupo.

Ainda para poder demonstrar a diferença obtida a partir da otimização do programa utilizando o mesmo algoritmo. Irei comparar, nas condições iguais às anteriores, como mesmo programa mas antes de ser otimizado:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| N: | Depois | | | | | Antes: | | | | |
| 10 | 0.050s | 0.049s | 0.045s | 0.051s | 0.045s | 0.049s | 0.051s | 0.046s | 0.049s | 0.048s |
| Média = 0.0480s | | | | | Média = 0.0486s | | | | |
| 100 | 0.046s | 0.045s | 0.044s | 0.043s | 0.043s | 0.046s | 0.050s | 0.051s | 0.049s | 0.047s |
| Média = 0.0442s | | | | | Média = 0.0486s | | | | |
| 1.000 | 0.043s | 0.043s | 0.048s | 0.044s | 0.044s | 0.048s | 0.051s | 0.046s | 0.043s | 0.045s |
| Média = 0.0444s | | | | | Média = 0.0466s | | | | |
| 10.000 | 0.048s | 0.049s | 0.042s | 0.046s | 0.053s | 0.049s | 0.046s | 0.054s | 0.047s | 0.048s |
| Média = 0.0476s | | | | | Média = 0.0488s | | | | |
| 100.000 | 0.051s | 0.050s | 0.049s | 0.049s | 0.050s | 0.053s | 0.055s | 0.050s | 0.055s | 0.051s |
| Média = 0.0498s | | | | | Média = 0.0528s | | | | |
| 1000.000 | 0.097s | 0.097s | 0.092s | 0.101s | 0.089s | 0.089s | 0.089s | 0.096s | 0.093s | 0.090s |
| Média = 0.0952s | | | | | Média = 0.0914s | | | | |
| 10.000.000 | 0.490s | 0.498s | 0.494s | 0.498s | 0.493s | 0.511s | 0.513s | 0.511s | 0.508s | 0.515s |
| Média = 0.4946s | | | | | Média = 0.5116s | | | | |
| 100.000.000 | 4.602s | 4.505s | 4.515s | 4.584s | 4.579s | 4.591s | 4.586s | 4.574s | 4.650s | 4.673s |
| Média = 4.557s | | | | | Média = 4.6148s | | | | |

Conclusão:

Durante a execução do presente trabalho considerei que ultrapassei facilmente as dificuldades que foram surgindo. A elaboração do mesmo proporcionou uma noção mais aprofundada acerca da criação, análise e comparação de Algoritmos, mais especificamente os de ordenação.

Em última análise, achei que o presente trabalho contribuiu para a consolidação dos nossos conhecimentos adquiridos em aula, considerando-o, deste modo, uma mais-valia.