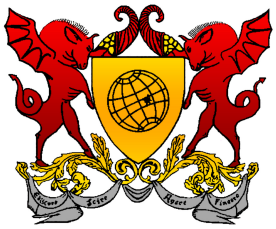
Universidade Federal de Viçosa
Campus Florestal
Ciência da Computação



2 que somam X

1º Trabalho Prático de Projeto e Análise De Algoritmos

Grupo Victor (02658)
Professor Daniel Mendes Barbosa

Belo Horizonte, 20 de setembro de 2017

Sumário

Resumo

Nesse trabalho irei analisar o desempenho de diferentes algoritmos na tarefa de encontrar, em um vetor, dois números que somados resultam em um dado valor X. Por fim será proposta uma modificação no método de pesquisa em busca de um desempenho melhor.

Abstract

This work is based on an analysis of different algorithms for finding two numbers in a vector whose sum is a given value X. Then i'll offer some changes in the searching method seeking better performance.

1 Introdução

Em sala de aula, o professor Daniel propôs uma solução para o problema de encontrar dois números em um vetor que somam um dado valor x. Tal solução era composta pela ordenação do vetor e a execução de uma busca binária em função de $(x - A_i) \forall i / 0 \leq i \leq t$, sendo t o tamanho do vetor, e $i \in Z$.

Para a primeira parte foi pedido que implementássemos essa solução utilizando diferentes algoritmos de ordenação com grau de complexidade $O(nlog(n))$. Eu escolhi os algoritmos QuickSort, HeapSort e MergeSort.

Para a segunda parte foi pedido que propuséssemos alguma alteração que não estivesse relacionada a ordenação para aumentar o desempenho do programa. Para essa parte levantei duas soluções as quais implementei e comparei os resultados mais a frente, são essas: troca do algoritmo de busca binária por um algoritmo de busca por interpolação e troca de toda a lógica para uma solução que utilizasse hash — o que reduziria a complexidade do algoritmo de $O(nlog(n))$ para $O(n)$.

2 Desenvolvimento

2.1 Ferramentas utilizadas

Durante a fase de codificação foram utilizadas as seguintes ferramentas: IDE KDevelop 5.1.1 para a codificação, o CMake 3.7.2 para a montagem e

configuração, o git 2.13.5 para o controle de versões e o GCC 5.4.0rev3[1] para a compilação. Durante a fase de documentação foi utilizada a plataforma online sharelatex que permite a escrita em LaTeX de forma fácil e cômoda por esta se encontrar na nuvem e o servidor github, que é compatível com a tecnologia git anteriormente citada, para hospedar o repositório de desenvolvimento — encontrado no link: https://github.com/primary157/2EmArraySomamX.

2.2 Decisões de projeto

Durante o desenvolvimento do algoritmo de Hash me deparei com uma certa inconsistência no desempenho quando o número de entradas crescia muito, logo depois descobri que esse problema estava relacionado tanto ao valor escolhido para M, sendo M o tamanho da Hash, pois muitas vezes o valor não era primo, quanto ao metodo de tratamento de colisões adotado, o qual optei por usar o método de endereçamento fechado por lista encadeada. Como a solução destes problemas encontrados seria muito trabalhosa e isso fugiria do escopo do trabalho, optei por abandonar a idéia de utilizar Hash como minha solução. Apesar disso mantive o código da Hash para futuramente eu continuar o que comecei.

2.3 Arquivos produzidos

2.3.1 binarysearch

Arquivo que implementa o algoritmo de busca binaria em vetor de inteiros.

2.3.2 buscacomhash

Arquivo que implementa o algoritmo que utiliza uma hash para inserir itens de um vetor de inteiros e assim efetuar a busca do numero de valor $(x - A_i) \forall i / 0 \leq i \leq t$, sendo t o tamanho do vetor, e $i \in Z$.

[1] para a compilação foram utilizadas as flags -O0 -march=native, o que permitiu uma análise independente de otimizações que interfeririam na análise do desempenho código em si

2.3.3 doissomamx

Arquivo que implementa as funções diretamente relacionadas ao trabalho, as quais possuem suporte a qualquer algoritmo de ordenação e busca que sigam o padrão de parâmetro e retorno.

2.3.4 hash

Arquivo que implementa o TAD Hash de endereçamento fechado por lista encadeada.

2.3.5 heapsort

Arquivo que implementa o algoritmo de ordenação por heap, ou heapsort, de um vetor de inteiros.

2.3.6 quicksort

Arquivo que implementa o algoritmo de ordenação por quicksort de um vetor de inteiros.

2.3.7 mergesort

Arquivo que implementa o algoritmo de ordenação por mergesort de um vetor de inteiros.

2.3.8 interpolationsearch

Arquivo que implementa o algoritmo de busca por interpolação em vetor de inteiros.

2.3.9 listaencadeada

Arquivo que implementa o TAD de lista encadeada auxiliar ao TAD Hash.

2.4 Códigos de terceiros incorporados

2.4.1 Quick Sort

O Algoritmo foi retirada da solução proposta por Nivio Ziviani em seu livro Projeto e Analise de Algoritmos em C e Pascal.

2.4.2 Heap Sort

Algoritmo foi retirado do site: https://gist.github.com/marcoscastro/6877436826b6a8a9ccd3#file-heap_sort-c

2.4.3 Merge Sort

O Algoritmo foi retirada da solução proposta por Nivio Ziviani em seu livro Projeto e Analise de Algoritmos em C e Pascal.

2.4.4 Interpolation Sort

O Algoritmo baseado na adaptação do código em pascal do site: https://users.dcc.uchile.cl/~rbaeza/handbook/algs/3/322.srch.p.html

3 Bateria de Testes

3.1 *geraraleatorio.py*

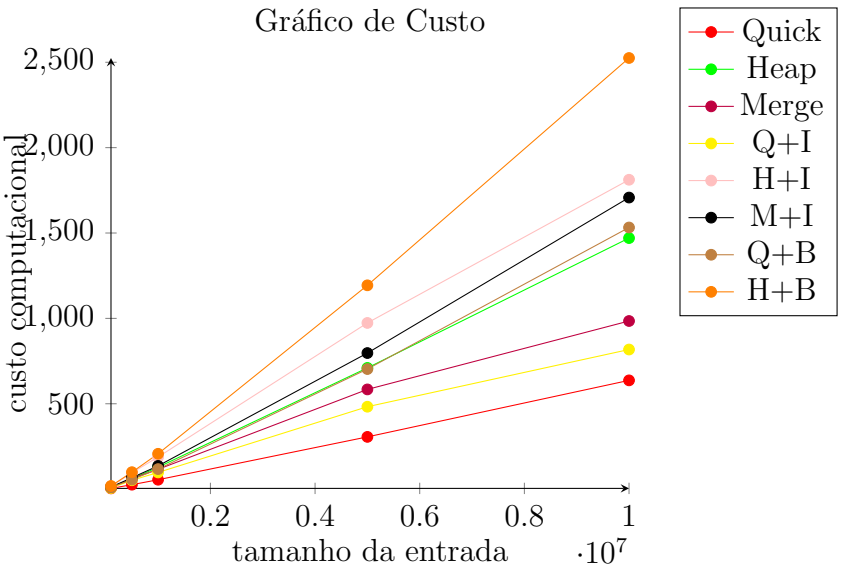
Para gerar os valores de entrada sem interferir no código, desenvolvi um programa em python que ao ter a saída padrão redirecionada para a entrada padrão do programa alimenta-o com dados os que esse necessita gerados aleatoriamente. Quando eu precisava de uma entrada 10x maior que 100.000 eu apenas precisava rodar o comando:

```
$ python geraraleatorio.py 10 | build/encontrar2quesomamx
```

Quando eu precisava de uma entrada de tamanho 100.000 eu apenas precisava rodar o comando:

```
$ python geraraleatorio.py 1 | build/encontrar2quesomamx
```

3.2 Gráfico de custo



4 Análise de complexidade

A maior parte dos algoritmos executados são algoritmos de grau de complexidade já conhecidos, logo cabe a nós examinar o grau de complexidade das combinações destas complexidades.

E como provamos no Exercício 6 da 1ª lista da matéria de PAA: "algoritmos de ordem de complexidade distintas quando somados resultam em um algoritmo de ordem de complexidade igual a maior ordem de complexidade dentre os algoritmos". Tomando isso como verdade, apresentarei a complexidade de cada algoritmo executado no programa.

4.1 Solução QuickSort + Busca Binária

Os algoritmos utilizados são Busca Binária e QuickSort(veja: 2.4.1), seus graus de complexidade são, respectivamente, O($log(n)$) e O($nlog(n)$) no caso médio. Como O(a) + O(b) = O(max(a,b)), então O($log(n)$) + O ($nlog(n)$) = O($nlog(n)$).

4.2 Solução QuickSort + Busca por Interpolação

Os algoritmos utilizados são a Busca por Interpolação(veja: 2.4.4) e o QuickSort(veja: 2.4.1), seus graus de complexidade são, respectivamente, O($loglog(n)$)— como pode ser visto na wikipedia — e O($nlog(n)$). Como O(a) + O(b) = O(max(a,b)), então O($loglog(n)$) + O ($nlog(n)$) = O($nlog(n)$).

4.3 Solução HeapSort + Busca Binária

Os algoritmos utilizados são Busca Binária e HeapSort(veja: 2.4.2), seus graus de complexidade são, respectivamente, O($log(n)$) e O($nlog(n)$) no caso médio. Como O(a) + O(b) = O(max(a,b)), então O($log(n)$) + O ($nlog(n)$) = O($nlog(n)$).

4.4 Solução HeapSort + Busca por Interpolação

Os algoritmos utilizados são a Busca por Interpolação(veja: 2.4.4) e o HeapSort(veja: 2.4.2), seus graus de complexidade são, respectivamente, O($loglog(n)$)— como pode ser visto na wikipedia — e O($nlog(n)$). Como O(a) + O(b) = O(max(a,b)), então O($loglog(n)$) + O ($nlog(n)$) = O($nlog(n)$).

4.5 Solução MergeSort + Busca Binária

Os algoritmos utilizados são Busca Binária e MergeSort(veja: 2.4.3), seus graus de complexidade são, respectivamente, O($log(n)$) e O($nlog(n)$) no caso médio. Como O(a) + O(b) = O(max(a,b)), então O($log(n)$) + O ($nlog(n)$) = O($nlog(n)$).

4.6 Solução MergeSort + Busca por Interpolação

Os algoritmos utilizados são a Busca por Interpolação(veja: 2.4.4) e o MergeSort(veja: 2.4.3), seus graus de complexidade são, respectivamente, O($loglog(n)$)— como pode ser visto na wikipedia — e O($nlog(n)$). Como O(a) + O(b) = O(max(a,b)), então O($loglog(n)$) + O ($nlog(n)$) = O($nlog(n)$).

4.7 Solução Hash

O algoritmo de Hash segundo material da ufabc todas operações — inserção, busca e remoção — são: no caso médio, O(1); no pior caso, O(n).

Se a Hash fosse otimizada teríamos uma complexidade total equivalente a $n*O(1)+n*O(1)$, equivalente a n inserções e n buscas no máximo, ou seja, $O(n)+O(n) = O(n)$.

Porém a Hash situada no trabalho provou-se ineficiente, o que nos leva ao pior caso de hash, que tem complexidade $O(n)$ para todas as operações, logo $n*O(n)+n*O(n)=O(n^2)+O(n^2)=O(n^2)$, equivalente a n inserções e n buscas no máximo.

5 Conclusão

Diante das funções de complexidade encontradas e do resultado empírico obtido nos testes, evidencia-se que o método de busca por interpolação juntamente ao algoritmo de ordenação heapsort possuem na média os melhores desempenhos. Com esse trabalho pude perceber o crescente *gap* de desempenho existente entre diferentes algoritmos de ordenação e busca, além de descobrir casos não tradicionais em que os algoritmos e TADS aprendidos durante a disciplina de Algoritmos e Estrutura de Dados são amplamente utilizados para se obter um resultado com baixo custo computacional.