O problema da agenda de viagens de Rick Sanchez

Felipe Matheus Guimarães dos Santos[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

2018098688

felipe.guimaraes@dcc.ufmg.br

1. Introdução

A situação problema do presente trabalho é a programação das viagens interplanetárias de Rick Sanchez. Ele deseja visitar uma quantidade p de planetas e ficar neles um tempo determinado. Porém, Rick não quer viajar mais que t horas durante um mês, ou seja, caso o tempo das visitações seja maior que t, Rick visitará os demais planetas em um mês posterior. Além disso, deseja fazer essa viagens em ordem crescente de tempo de visitação. Por fim, cada mês, já ordenado por tempo, será visitado em ordem alfabética.

2. Implementação

Para resolver esse problema, serão usados algoritmos de ordenação vistos em aula: Merge Sort, Couting Sort e Radix Sort. O Merge será usado para ordenar o tempo das visitações, pois Rick especificou que o algorimo utilizado deveria ser estável e de ordem de complexidade $O(n\log_2(n))$. Por outro lado, para ordenar as visitações em cada mês alfabeticamente, foi usado o algorimo Radix Sort, o qual opera em conjunto com o Counting Sort, de ordem de complexidade O(nk), sendo k o comprimento dos nomes.

O programa principal roda em loop infinito, recebendo três inteiros de entrada, separados por espaço: tempo máximo de visitação, número de planetas e comprimento da string. O retorno do programa é a agenda de viagens de Rick, o qual é impresso na saída padrão. O loop do programa principal é interrompido por EOF (Ctrl + C no CMD do Windows ou Ctrl + D no Terminal do Linux).

2.1. Planetas

Tipo abstrato de dados usado para armazenar os atributos de cada planeta: tempo de visitação, mês de visitação e nome do planeta. Tempo e mês são do tipo inteiro e nome é uma string.

- **LePlanetas:** Recebe um vetor de planetas e a quantidade de planetas. Lê pela entrada padrão os dados dos planetas e os salvas no vetor.
- **SeparaMeses:** Recebe um vetor de Planetas, a quantidade de planetas, p, e o valor máximo de horas por mês, t. Para calcular a quantidade de meses, essa função apenas soma cada valor do vetor já ordenado e verifica se o valor da soma é maior que t. Se não for, continua somando. Se for maior ou for a posição sentila, a soma volta para 0 e a quantidade de meses é incrementado.
- **PrintaPlanetas:** Recebe um vetor de planetas e a quantidade de planetas. Printa na saída padrão todos os dados de planetas separados por tab: mês de visitação, nome e tempo de visitação. Cada planeta é separado por \n.

2.2. Merge Sort

A implementação do Merge Sort foi baseada nos slides disponibilizados pelo professor da disciplina Estrutura de Dados (EDs) Luiz Chaimowicz e algumas moficações baseadas no algoritmo do site Code*Codex*[1].

Separado em duas funções: Merge e MergeSort.

- **MergeSort:** Recebe um vetor de Planetas, a posição da esquerda, iniciada em 0, e da direita, iniciada em p-1.

  Divide o problema ao meio, chamando-se recursivamente duas vezes: passando apenas a metade esquerda do vetor e, após todas as operações da esquerda, passando apenas a metade direita.

  Após tudo ser empilhado, chama a função **Merge** e desempilha.

- **Merge:** Recebe como parâmetro um vetor de Planetas, a posição mais à esquerda, a posição central e a posição mais à direita.

  Para a execução, cria um vetor de Planetas auxiliar com direita - esquerda +1

1 http://www.codecodex.com/wiki/Merge_sort#C.2B.2B, acesso em 31 out. 2019

posições. Após, faz as verificações para ordenação: verifica se a posição mais à esquerda do vetor é menor ou igual, para ser estável, que o elemento na posição central mais um. Se a condição for satisfeita, adiciona o valor da esquerda na primeira posição do vetor auxiliar e passa para a posição seguinte da esquerda. Se não for satisfeita, adiciona o valor da posição central mais um na primeira posição do vetor. Esse loop é iterado até que o cursor da esquerda ultrapasse a posição central ou o cursor da direita ultrapasse o limite da direita. Em suma, compara cada posição da metade da esquerda com todas as posições da segunda metade, passando para o vetor auxiliar o menor entre essas comparações.

Se sobrarem valores que não foram comparados, eles apenas serão inseridos no vetor de forma sequencial, pois, para isso acontecer, precisa-se de ter uma quantidade desigual de valores menores nos dois lados do vetor.

Copia os valores do vetor auxilar no vetor principal e deleta o auxiliar.

Após fazer isso, continua desempilhando os valores recursivamente até ordenar o vetor.

2.3. RadixSort e CountingSort

A implementação do RadixSort foi baseada nos slides disponibilizados pelo monitor da disciplina Estrutura de Dados (Eds), Allison Svaigen, e algumas moficações foram baseadas no algoritmo do site *StackOverflow*[2].

- **RadixSort:** Recebe um vetor de Planetas, o tamanho do vetor (*p*) e o comprimento dos nomes, todos os nomes têm mesmo tamanho, *c,* $\log_2 p$.

  O Radix funciona apenas com um loop, iterando da posição c-1 do vetor até a posição 0. A cada iteração, chama o CountingSort, para ordenar o vetor da posição menos relevante para a mais relevante, isto é, da última letra para a primeira.

- **Counting Sort:** Recebe um vetor de Planetas, o tamanho do vetor e a posição iterada do nome dos Planetas.

  Primeiro, cria um vetor auxiliar com 26 posições, uma para cada letra, e o preenche com zeros. Após, contabiliza cada letra na determinada posição, isto é

2https://stackoverflow.com/questions/32543683/radix-sort-array-of-strings, acesso em 8 nov. 2019

a cada letra '*a*' encontrada, soma um na posição 0 do vetor e assim por diante. A letra a minúscula corresponde ao valor 97 na tabela ASCII, ou seja, sempre que recebemos uma letra, ela será mapeada para a posição alfabética dela sendo subtraída pelo número 97. Com a quantidade de letras salvas, passa para o procedimento de somar à posição i a posição i-1, necessário para o CountingSort. Por fim, cria-se um novo vetor com a mesma ordem do vetor ordenado. Para preencher esse vetor, auxiliar[contador[principal[i]]-1] e iguala a principal[i], i indo da posição mais à direita até a esquerda. Após adicionar esse valor ao vetor auxiliar, subtrai-se um da posição i do vetor contador.

Após todas as operações, copia os valores ordenados do vetor auxiliar nas mesmas posições do vetor principal. Deleta os vetores contador e auxiliar.

- **RadixMes:** Recebe o vetor de planetas já ordenado por tempo, a quantidade p de planetas, a quantidade c de caracteres por nome de planeta e o valor t, número máximo de horas de visitação por mês. Chama a função SeparaMeses para calcular a quantidade de meses. Após, verifica se a quantidade de meses é maior que um. Se não for, todos os planetas poderão ser visitados em apenas um mês, desta forma, apenas passa-se o vetor inteiro para o RadixSort. Porém, se a quantidade de meses for maior que um, precisa-se separar em subvetores e passar cada um desses vetores na função de ordenação alfabética. Para tal, foi criado um loop que roda enquanto a variável contador for menor que a quantidade de meses. O loop soma os tempos de visitação, parando quando for maior que t. Assim que essa última condição é satisfeita, cria-se um vetor de Planetas com n posições, sendo n a quantidade de planetas no mês. Passa-se os dados dos respectivos planetas para o auxiliar e chama a função RadixSort para esse vetor auxiliar. Assim que os planetas foram ordenados alfabeticamente, atribui essa nova ordenação no vetor principal, copiando do auxiliar para o principal e deletando-o.

2.4. Main

Funciona em loop infinito, recebe três variáveis inteiras: t, p e c; tempo máximo de visitação por mês, número de planetas e quantidade de caracteres por nome, respectivamente. Interrompa a execução com EOF, ctrl+c no CMD do Windows.

Após receber os três valores, cria um vetor com p+1 planetas, a posição final fica como sentinela para somas durante a execução, e lê os dados desses planetas. A posição sentinela recebe tempo 0.

Depois que todos os dados dos planetas foram lidos, chama-se a função *MergeSort* para ordenar os vetores por tempo, de forma estável e com complexidade $n\log_2 n$.

Por fim, chama a função RadixMes, para separar os meses e ordená-los.

Após fazer a ordenação alfabética de todos os meses, printa os planetas na saída padrão com a função PrintaPlanetas.

2.5. Makefile

Para compilar e rodar o programa foi gerado um Makefile. Este arquivo facilita a compilação, pois elimina a necessidade de dar os comandos do g++ no terminal, apenas digitamos *make test* para rodar os testes ou *make run* para rodar o programa.

3. Instruções de compilação e execução

Para compilar e executar o programa usa-se o Makefile. Para tal, abra o Terminal e entre no diretório onde se encontra o programa, i.e., *.../felipe_guimaraes/src*. Dentro do diretório, dê o comando *make test* para rodar os testes ou *make run* para rodar o programa. Após a execução, dê o comando *make clean* para apagar os arquivos temporários. *Há* um arquivo *README.txt* com as especificações.

Os compiladores utilizados foram: g++ (MinGW.org GCC-8.2.0-3) 8.2.0*,* no Windows 10, e g++ (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0, no Ubuntu.

4. Análise de complexidade

- **Merge Sort:** Sua complexidade de tempo é $\theta(n\log_2 n)$ em todo caso, pois não há variações de acordo com a entrada. Além disso, tem complexidade de espaço linear, O(n), pois cria subvetores de tamanho no máximo 2*n/2, ou seja, n, a cada divisão.
- **Radix Sort:** Para cada letra, executa um Counting Sort. A complexidade do Counting Sort é O(n+k), sendo k o tamanho máximo do vetor. Neste programa,

o valor máximo é 25, pois temos 26 letras no alfabeto. Logo, a complexidade do Counting Sort é linear, O(n), pois k se torna insignificante para grandes valores de n. Portanto, a complexidade do Radix Sort é O(nk), sendo k o tamanho do caractere que nomeia os planetas, todavia, esse valor foi especificado como $\log_2 n$, logo, a complexidade do Radix Sort é $O(n\log_2 n)$.

A complexidade de espaço é constante, $\theta(1)$, pois não são feitas recursões.

- **Main:** Comparando as complexidades do Radix Sort e do Merge Sort, percebe-se que ambos têm a mesma complexidade. Portanto, a complexidade total do algoritmo é $O(n\log_2 n)$.

5. Conclusão

Portanto, para concluir o presente trabalho foram usados três algoritmos de ordenação: Merge Sort, Counting Sort e Radix Sort. A utilização de cada um desses algoritmos foi pensada de acordo com as especificações do trabalho e com o auxílio do monitor Allison Svaigen.

A maior dificuldade do trabalho estava nos ponteiros da separação dos meses e aplicação do Radix Sort, porém, com um pouco de paciência e teste de mesa, o problema foi consertado.

Como dito anteriormente, foi criada uma posição sentinela no fim do vetor de planetas, pois grande partes das verificações e somas são feitas entre as posições i e i+1, tornando a posição necessária para não gerar segmentation fault.

Logo, o trabalho cumpre o que era esperado: utilizar todo o conhecimento que foi acumulado desde o ingresso na Universidade e o conhecimento de todos os algoritmos de ordenação, sabendo quando usar determinado algoritmo, respeitando as especificações.

O código e a documentação deste trabalho estão disponíveis no GitHub: https://github.com/FelipeGuimaraes42/TP2-Planetas

6. Bibliografia

TEODOROWITSCH, Roland. Adaptação para OpenOffice.org 1.1 feita em 29 mar. 2005. Disponível em: http://www.sbc.org.br/documentos-da-sbc/summary/169-templates-para-artigos-e-capitulos-de-livros/878-modelosparapublicaodeartigos. Acesso em 26 set. 2019.