SCC0220 - Laboratório Introdução à Ciência da Computação II

Relatório de execução da aula prática 4

Alunos	**NUSP**
Felipe Camargo Cerri	15451119
Gabriel Campanelli Iamato	15452920

Exercício 4 – Notáveis

Item a

➔ Comentário

O exercício da semana nos deu a liberdade para escolher a forma de implementação da solução do problema. Dessa forma, partimos de duas abordagens semelhantes que resolvem o exercício com complexidades de tempo diferentes a fim de compará-las e demonstrar na prática a eficiência do método escolhido: a primeira utiliza o algoritmo Quick Sort para ordenar o vetor de alunos enquanto a outra inicialmente realiza o algoritmo Quick Select para diminuir a complexidade de tempo em situações específicas. O gerenciamento de memória por sua vez foi implementado da mesma maneira em ambos os casos. Vale ressaltar que utilizamos uma struct “ALUNO”, que contém o nome e aumento referente a cada indivíduo para facilitar as operações.

De início, a nossa decisão acerca da utilização de memória foi balancear o tempo de demora das realocações sem utilizar muito espaço desnecessário. Para contornar ausência do conhecimento da quantidade de dados tratados a priori, escolhemos alocar em um vetor uma quantidade inicialmente fixa de memória (1000 structs ALUNO) e através de uma função realizar a verificação do espaço disponível antes de inserir algum elemento no vetor, caso não haja espaço a função realloc é utilizada para alocar mais 1000 structs. Além disso, a escolha do Quick Sort traz a complexidade de espaço como vantagem em relação a outros algoritmos de ordenação a exemplo do Merge Sort, uma vez que não aloca memória extra para realizar a reordenação, fazendo as operações de troca dentro do próprio vetor a ser ordenado.

Antes de mais nada, precisamos discutir a diferença entre os dois métodos implementados para a resolução do problema. Ambos iniciam-se lendo toda a entrada e alocando os alunos em um vetor, a complexidade da ação é O(n). Para a estratégia convencional o próximo passo é ordenar todos os dados e imprimir todos os elementos do array até k, incluindo os empates no último aumento, assim a complexidade de tempo será impactada pelo tempo de ordenação sendo O(nlogn) para o caso médio do Quick Sort, resultando em O(nlogn + n) = O(nlogn). A maneira otimizada por sua vez utiliza do Quick Select que acha o k-ésimo maior elemento de um vetor não ordenado em complexidade média O(n) com o objetivo de reduzir o tamanho do array que deve ser ordenado, no entanto, como o algoritmo não ordena o vetor, a procura dos empates com o aumento no último elemento terá complexidade O(n - k), portanto, temos a complexidade final O(n + n + n - k + klogk) = O(n + klogk), dessa forma o algoritmo se mostra muito eficiente para um k<<n (como nos casos testes do runcodes) pois a complexidade será O(n), enquanto para um k equivalentemente grande em relação a n, a complexidade será O(klogk).

Finalmente podemos adentrar na implementação dos algoritmos. Tanto o Quick Sort quanto o Quick Select utilizam a abordagem de partição, isto é, estabelecer um pivot em um dado array e reorganizar todos os elementos maiores de um lado e os menores do outro, a diferença entre os dois reside no fato de que o método de ordenação realiza divisão e conquista, ao passo que faz chamadas recursivas para os vetores resultantes a esquerda e à direita de cada pivot encontrado a fim de ordenar completamente os dados, o algoritmo de busca por sua vez se restringe a realizar a partição novamente em apenas um dos lados a fim de encontrar o pivot correto, ambos fazem chamadas recursivas até que o caso base seja satisfeito. Por fim, a implementação da função de partição utilizou da técnica de estabelecer o pivot como a mediana entre o elemento central, inicial e final, para evitar o pior caso ($O(n^2)$) e depois iterar sobre todos o elementos para partí-lo.

➔ Código

Gerenciamento de memória:

```c
void gerencia_memoria(ALUNO **alunos, int *tam, int *alocado) {
    if (*tam == *alocado) {
        ALUNO *temp = realloc(ptr: *alunos, size: ((*alocado + 1000)*sizeof(ALUNO)));
        if (temp == NULL) exit(status: -1);

        *alunos = temp;
        *alocado += 1000;
    }
    (*tam)++;
}
```

Quick select:

```c
int quick_select(ALUNO *alunos, int k, int inicio, int fim) {
    int i = particao(alunos, inicio, fim);

    if (i == k-1) //caso base
        return i;

    if (i < k-1)
        return quick_select(alunos, k, inicio: i+1, fim); //refaz para o lado direito(maior)

    else
        return quick_select(alunos, k, inicio, fim: i-1);//refaz para o lado esquerdo(menor)

}
```

Quick Sort:

```c
void quick_sort(ALUNO *alunos, int inicio, int fim) {
    if (inicio >= fim) //caso base
        return;

    int pivot = particao(alunos,inicio,fim); //ajusta pivot em sua posicao e organiza vetor
    quick_sort(alunos, inicio, fim: pivot-1); //ordena esquerda
    quick_sort(alunos, inicio: pivot+1, fim);  //ordena direita

}
```

Partição:

```
int particao(ALUNO *alunos, int inicio, int fim) {
    swap(a: &alunos[mediana(a: inicio, b: (fim+inicio)/2, c: fim)], b: &alunos[fim]); //evita o pior caso
    int i = inicio; ALUNO pivot = alunos[fim];


    for (int j = inicio; j<fim; j++) { //organiza os elementos e estabelece o local certo para o pivot
        if (alunos[j].aumento > pivot.aumento) {
            swap(a: &alunos[i], b: &alunos[j]);
            i++;
        }
        else if (alunos[j].aumento == pivot.aumento) { //desempate por nome
            if (strcmp(s1: alunos[j].nome , s2: pivot.nome) < 0) {
                swap(a: &alunos[i], b: &alunos[j]);
                i++;
            }
        }
    }
    swap(a: &alunos[fim], b: &alunos[i]); //coloca o pivot em sua posicao
    return i; //index do pivot
}
```

Adiciona empates (no método do Quick Select):

```
int adiciona_empates(ALUNO *alunos, int id, int fim, int menor) {
    for (int j = id+1; j<=fim; j++) { //percorre o restante do vetor adicionando empates
        if (menor == alunos[j].aumento) {
            swap(a: &alunos[id+1], b: &alunos[j]);
            id++;
        }
    }
    return id; //retorna o index ajustado do ultimo aluno incluso
}
```

Adiciona empates (no método de apenas ordenar):

```
id = k-1;
while(id < count && alunos[id+1].aumento == alunos[k-1].aumento) id++;
```

Struct ALUNO:

```
typedef struct aluno_ {
    int aumento;
    char nome[52];
} ALUNO;
```

➔ Saída

Para comprovar a eficiência dos dois métodos de ordenação, foi utilizada a biblioteca <time.h> da linguagem C para medir o tempo de execução do algoritmo. Os casos teste foram obtidos através dos códigos

disponibilizados para a construção de arquivos de entrada, foram feitas algumas alterações para permitir a criação de arquivos com a quantidade de alunos até 10^6. O tempo de leitura e alocação dos dados foi incluído na contagem.

Entrada: 100 alunos, k = 5;

Apenas Quick Sort - Tempo: 0.000105

```
Ines Torres Silva
Kleber Elias Borges
Alvaro Pires Yoshida
Fabio Hernandes Costa
Zeca Campos Bittencourt
Tempo de Execução: 0.000105
```

Quick Select + Quick Sort - Tempo: 0.000089

```
Ines Torres Silva
Kleber Elias Borges
Alvaro Pires Yoshida
Fabio Hernandes Costa
Zeca Campos Bittencourt
Tempo de Execução: 0.000089
```

- Tempo de execução mais curto: **Apenas Quick Sort**

Entrada: 100 alunos, k = 90;

Apenas Quick Sort - Tempo: 0.000135

```
Denise Kawasaki Valente
Murilo Rodrigues Oliveira
Heitor Klein Gomes
Rodrigo Dantas Machado
Wesley Klein Silva
Simone Ximenez Xavier
Ana Souza Ximenes
Cristiano Oliveira Dantas
Tempo de Execução: 0.000135
```

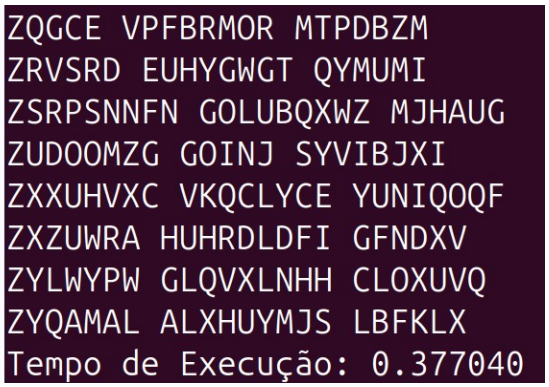
Quick Select + Quick Sort - Tempo: 0.000114

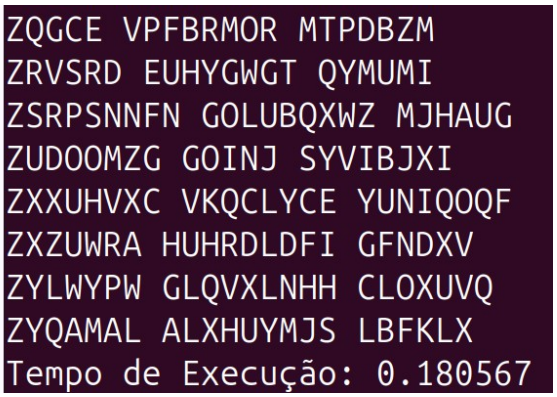
```
Denise Kawasaki Valente
Murilo Rodrigues Oliveira
Heitor Klein Gomes
Rodrigo Dantas Machado
Wesley Klein Silva
Simone Ximenez Xavier
Ana Souza Ximenes
Cristiano Oliveira Dantas
Tempo de Execução: 0.000114
```

- Tempo de execução mais curto: **Quick Select + Quick Sort**

Entrada: 10^6 alunos, k = 10^3;

Apenas Quick Sort - Tempo: 0.377040 **Quick Select + Quick Sort - Tempo: 0.180567**

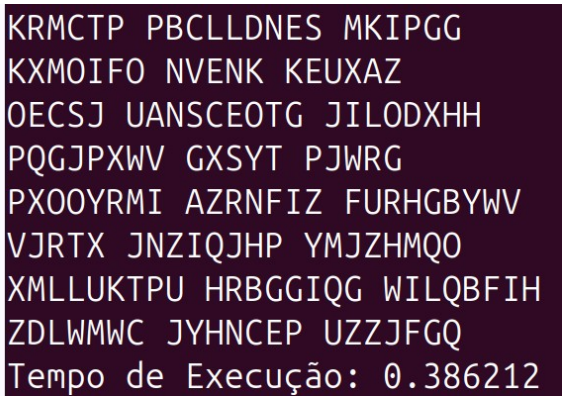




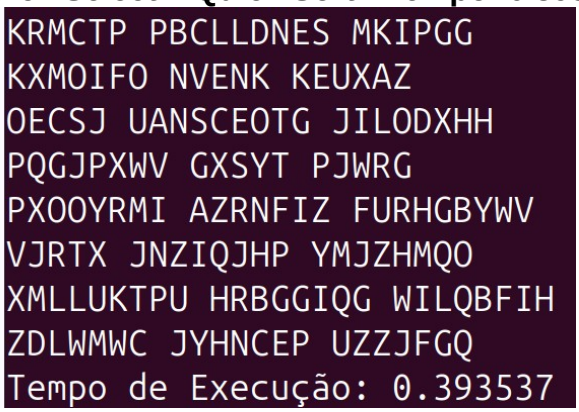
- Tempo de execução mais curto: **Quick Select + Quick Sort**

Entrada: 10^6 alunos, k ≈ 10^6;

Apenas Quick Sort - Tempo: 0.386212



Quick Select + Quick Sort - Tempo: 0.393537



- Tempo de execução mais curto: **Apenas Quick Sort**

Como podemos verificar, o método de apenas ordenar não possui impacto significativo no tempo de execução com relação a escolha do k, diferentemente do método que utiliza do Quick Select, no qual os casos com k muito menor que n demonstram grande melhoria no tempo de processamento. Para os dados casos teste foi possível concluir que o método mais eficiente num contexto geral é o do Quick Select, uma vez que o tempo de execução é muito menor nos casos favoráveis (verificado por O(n) < O(nlogn)) e mesmo para k muito próximos de n a perda de eficiência não é tão grande visto que o Big O nessa circunstância para os dois será o mesmo.