Relatório do Projeto 1

Gabriel de Andrade Dezan – NUSP: 10525706

Ivan Mateus de Lima Azevedo – NUSP: 10525602

Exercício 1

A abordagem utilizada foi a seguinte: primeiro foi realizada a ordenação das *n* cargas. Os critérios de ordenação são: os volumes são ordenados de forma decrescente. Se houver um ou mais volumes iguais, a carga que tiver menor ID será priorizada. Após isso, são retornadas as *p* primeiras cargas.

Para a ordenação foi utilizado o algoritmo QuickSort, pois ele funciona bem dentro da faixa de valores que a quantidade de cargas pode assumir (tanto para os valores pequenos quanto para os grandes).

```c
int* solucao(struct entrada *entradas, int n, int p) {
  int *ret = (int *) malloc(p * sizeof(int));
  quick_sort_volumes_ids(entradas, 0, n - 1);     //ordena as entradas de acordo com os criterios ja definidos acima
  for (int i = 0; i < p; ++i) {
    ret[i] = entradas[i].id;       //retorna os p primeiros ids
  }
  return ret;
}
```

Figura 1: trecho de código da solução do primeiro exercício.

Então, como pode-se observar na figura acima, a complexidade do algoritmo é composta pela complexidade da alocação do vetor de inteiros (que é constante, logo é *O(c)*), mais a complexidade do QuickSort que é em média *O(nlog(n))*, mais a complexidade da atribuição dos elementos, que não é *O(n)* e sim, *O(p)*, que é o número de cargas que cada motorista pode transportar. Então a complexidade fica:

$$T(n,p)=O(c)+O(n\cdot\log(n))+O(p)$$

Mas como p é sempre menor que ou igual a n, então no pior caso (onde p = n), a complexidade *O(nlog(n))* ainda seria maior que *O(p)* = *O(n)* e que *O(c)*. Logo, a complexidade da solução se resume à complexidade da ordenação em si:

$$T(n)=O(n\cdot\log(n))$$

Exercício 2

Para fazer o exercício 2 era necessário inicialmente utilizar o BubbleSort para ordenar os hobbies de cada usuário. Isso foi necessário, pois o próprio problema foi posto dessa forma (quanto menor as trocas adjacentes do BubbleSort, maior a afinidade). A fim de não modificar a lista de hobbies de cada usuário, dentro do código do BubbleSort é feita uma cópia do vetor e a cópia é que é ordenada. Como a complexidade do BubbleSort é *O(h²)* e a complexidade de uma cópia de um vetor é *O(h)* (onde h é o número de hobbies de cada usuário), então a complexidade da ordenação continua sendo *O(h²).* Após cada ordenação é retornado o número de inversões feitas.

```
68 int *solucao(struct entrada *entradas, int n, int h)
69 {
70   int i = 0;
71   int *ret = (int *) malloc(n * sizeof(int));
72   int *inversoes = (int *) malloc(n * sizeof(int));
73   for(i = 0; i < n; ++i) {  //guarda o numero de trocas de cada usuario
74     inversoes[i] = inversoes_bubble_sort(entradas[i].hobbies, h);
75   }
76   quick_sort_afinidades_ids(entradas, inversoes, 0, n - 1); //ordena os usuarios de acordo com os quesitos ja especificados
77   for(i = 0; i < n; ++i) {  //guarda os ids
78     ret[i] = entradas[i].usuario_id;
79   }
80   return ret;
81 }
```

Figura 2: trecho de código da solução do segundo exercício.

Como pode ser visto na figura acima, temos uma complexidade O(c) para as linhas 70 a 72. Logo após elas, a ordenação e a atribuição do número de inversões já são feitas em um loop de modo que o vetor que guarda esses números é chamado *inversoes.* Visto que essa atribuição é feita *n* vezes, onde *n* é o número de usuários, então a complexidade desse loop é *O(n)*O(h²) = O(n*h²).*

Logo após isso, é feita uma ordenação da lista de usuários de acordo com o número de inversões (em ordem crescente). Como essa ordenação é feita usando o QuickSort, então a complexidade é *O(nlog(n)).*

Por fim, os ID's dos *n* primeiros usuários são guardados em um vetor que é retornado. Então temos uma complexidade *O(n)*. Considerando todos esses termos, temos:

$$T(n,h)=O(c)+O(n\cdot h^2)+O(n\cdot \log(n))+O(n)$$

Visto que não se pode estabelecer uma ordem de quem será maior que quem entre *n* e *h*, então a complexidade vai depender de quem é maior. Se *n* >> *h,* então a complexidade fica:

$T(n)=O(n\cdot \log(n))$, pois *O(c)*, *O(n)* e *O(nh²)* serão menores que *O(nlog(n))*.

Porém, se *n* << *h*, então a complexidade é:

$T(h)=O(h^2)$, pois *O(c)*, *O(n)* e *O(nlog(n))* serão menores que *O(nh²)* e *nh² ≈ h².*

Agora, se *n* e *h* forem relativamente do mesmo tamanho, então a complexidade é:

$$T(n,h)=O(n\cdot h^2)$$

Exercício 3

No exercício 3, foi feita primeiro uma análise dos jogos para atribuir os pontos aos respectivos times. Foi necessária uma iteração pelo vetor 'entradas', logo, complexidade

O(m). É, também, preenchido o vetor de retorno com os times em ordem crescente, de forma que sendo o vetor de times ‘ret’ e o vetor de pontos ‘team_points’, temos que o time ret[x] terá a quantidadade de pontos team_points[x].

```c
int *solucao(struct entrada *entradas, int m, int n_teams) {
    int *ret = (int *)malloc(n_teams * sizeof(int));
    // Implemente aqui sua solução

    // Vetor com a pontuação dos times
    int *team_points = (int *)malloc(n_teams * sizeof(int));

    // Adiciona os pontos de cada time por jogos ao vetor team_points com o
    // auxílio do método get_points
    for (int i = 0; i < m; i++) {
        team_points[entradas[i].x - 1] += get_points(entradas[i])[0];
        team_points[entradas[i].z - 1] += get_points(entradas[i])[1];
    }

    // Popula o vetor ret com os times em ordem crescente
    for (int i = 0; i < n_teams; i++) {
        ret[i] = i + 1;
    }

    sort_both(entradas, team_points, ret, n_teams, m);
    return ret;
}
```

Figura 3: trecho de código da solução do terceiro exercício.

Em seguida, esse vetor com o resultado final de cada time é ordernado inversamente utilizando o algoritmo de QuickSort. Ele fica ordernado de forma inversa pois o time com maior quantidade de pontos deve vir primeiro, fazendo com que os pontos fiquem em ordem decrescente. Como os vetores ‘ret’ e ‘team_points’ devem andar paralelamente, quando uma troca é feita no team_points, ela é feita também no ret, para que eles estejam “espelhados”. Complexidadade do QuickSort: $T(n)=O(n \cdot \log(n))$

Durante o QuickSort, porém, existe uma comparação sendo feita no caso de empate que itera pelas partidas para fazer a média de cestas de um time. Essa iteração tem complexidade de O(m). O que deixaria nosso QuickSort da forma: $T(n)=O(m \cdot n \cdot \log(n.m))$.

Temos então na complexidade final:

$$T(n)=O(m \cdot n \cdot \log(n \cdot m)+m)$$

$$T(n)=O(m \cdot n \cdot \log(m \cdot n))$$

Como $m=\frac{n(n-1)}{2}$,

$$T(n)=O(\frac{n(n-1)}{2} \cdot n \cdot \log(m \cdot n))$$

$$T(n)=O(n^3 \cdot \log(n^3))$$

$$T(n)=O(3 \cdot n^3 \cdot \log(n))$$

$$T(n)=O(n^3 \cdot \log(n))$$

Exercício 4

Para o exercício 4, primeiro os itens foram ordenados usando um QuickSort padrão, o que nos dá a complexidade: $T(n)=O(n \cdot \log(n))$

E em seguida é feita uma busca binária nos itens, utilizando as consultas dos clientes. Cada busca tem a complexidade: $T(n)=O(\log(n))$

```c
int *solucao(struct entrada entrada, int n, int c) {
    int *ret = (int *)malloc(c * sizeof(int));

    // Implemente aqui sua solução
    // Para as consultas, caso o item não exista, retorne -1
    quick_sort(entrada.itens, n);

    for (int i = 0; i < c; i++) {
        // Popula o vetor de resultados com os resultados das buscas
        ret[i] = busca(entrada.itens, 0, n - 1, entrada.consultas[i]);
    }

    return ret;
}
```

Figura 4: trecho de código da solução do quarto exercício.

Temos que a complexidade final é:

$$T(n,c)=O(n \cdot \log(n)+c \cdot \log(n))$$

$$T(n,c)=O((n+c) \cdot \log(n))$$

Onde c pode ser ignorado caso seja muito menor que n, tornando-se irrelevante para o cálculo.

Exercício 5

Nesse exercício é feito um MergeSort normal, porém após os subvetores de uma iteração dessa ordernação estarem ordenados, é feita uma contagem cada vez que um item do subvetor da "direita" é selecionado, pois ele conta como uma troca.

Quanto ao valor incrementado nas trocas, será (meio - l1), pois se l[l1] > l[l2], então significa que todos os valores da posição (l1 + 1) até a posição meio também vão ser maiores que o valor da posição l2, pois os subvetores são ordenados. Então é como se l[l2] fizesse trocas com todos esses (meio - l1) elementos.

```c
int solucao(int *arr, int n)
{
    // Implemente aqui sua solução
    int *copia = (int *)malloc(n * sizeof(int));
    int ret = merge_sort_trocas(arr, copia, 0, n - 1); // Atribua aqui o número de inversões
    return ret;
}
```

Figura 5: trecho de código da solução do quinto exercício.

No fim, o algoritmo final é um MergeSort padrão, com algumas checagens.

Temos então a complexidade final sendo a complexidade apenas do MergeSort.

$$T(n)=O(n \cdot \log(n))$$