INF1721 - Primeiro Trabalho de Implementação

Carlos Mattoso (1210553)
Michelle Valente (1312828)
Gabriel Barros (1111061)

Sumário

1 Produção de Frascos de Vidro 3

1.1 Descrição do Algoritmo 3

1.2 Análise

1.3 Resultados

1.4 Conclusão

2 Problema da Mochila Fracionária

2.1 Descrição do Algoritmo

2.2 Análise

2.3 Resultados

2.4 Conclusão

Produção de Frascos de Vidro

Descrição do Problema: O problema trata do desafio de se fazer o controle de qualidade de k frascos iguais lançados de uma escadaria com até n degraus. Deseja descobrir, com o menor número possível de tentativas, o degrau x a partir do qual os frascos quebram.

Descrição do Algoritmo: No mundo real, a estratégia ótima a ser adotada é de subdividir-se o espaço de tentativas em $n^{1/k}$ segmentos de degraus. Feito isto, basta tentar lançar o frasco do degrau mais alto de cada segmento: caso quebre, sabemos que o primeiro degrau encontra-se neste segmento, e aplicamos a mesma estratégia recursivamente no mesmo; caso não quebre, tentamos no próximo segmento até atingirmos o topo. Quando nos restar apenas um frasco, trivialmente decorre que devemos começar do primeiro degrau até o último degrau do segmento eleito, para encontrar onde o frasco quebra.

1. Tratemos o caso particular em que temos $k = 2$ frascos. Neste caso, temos $\sqrt{n}$ segmentos de degraus cada um com $\sqrt{n}$ degraus. No pior caso, os frascos quebram apenas no último degrau. Assim, teremos que testar em todos os $\sqrt{n}$ segmentos para constatar que o frasco finalmente quebra no último segmento. Neste, usando o frasco remanescente, começamos no primeiro degrau até atingir o último, relizando $\sqrt{n}$ operações. Portanto, uma estratégia de complexidade $2 \times \sqrt{n} = O(\sqrt{n})$.

2. Em se tratar do problema prosposto, não determinamos x exatamente, jà que este valor nos é dado. Assim sendo, simulamos a busca por tal valor. Pensemos em sua representação na base binária, como nos é passada a entrada. Seja N o número de bits necessários para representar x. Ora, temos um espaco de 2^N possibilidades de valores que podem ser assumidos por x. Deste modo, nossa estratégia deve explorar esse domínio de alguma forma mais inteligente que pura força bruta, afinal esta nos custaria $O(2^N)$. Ainda, devemos respeitar ao máximo as limitações do modelo: ao avaliar os *bits* de x é preciso sempre começar dos menos signficativos, pois o contrário no mundo real implicaria verificar se um frasco quebra acima de seu limite mais de uma vez, um erro fatal.

Decorre que a estratégia ótima consiste em segmentar x em k blocos de *bits*. Feito-se isto, incrementa-se uma variável de 0 até o valor deste bloco. Após repetir este processo em todos os blocos, teremos encontrado o valor de x. Analisemos este algoritmo: temos N bits, os quais dividimos em k segmentos, sendo que em cada um destes devemos explorar, no pior caso, um espaço de possibilidades de no máximo $2^{N/k}$ *bits*. Logo, ao todo realizamos $k \times 2^{N/k} = k \times n^{1/k}$ operações. Segue abaixo o algoritmo central de desta estratégia:

```
ret drop(int n_bits, int k, int * data ){
    int step, i, answer[10], start, end, result;
    for(i =0; i<10;i++)
        answer[i]=0;

    step = n_bits/k;
    start = 0;
    end = step;
    while(end <= n_bits){
        while(1){
            add(1,answer,start,end,INTBIT);
            result = compare(answer,data,start,end,INTBIT);
            if (result == 0 )
                break;
            else if (result >0){
                add(-1,answer,start,end,INTBIT);
                break;
            }
        }
        start +=step;
        end += step;
    }
}
```

Para os casos particulares de $k = 3$ e $k = 4$, bastaria apenas fixar este valor acima. Caso houvesse algum resto na divisão, seria ainda preciso adicionar alguma lógica para tratar disto, mas a lógica essencial não varia.

Análise de complexidade do algoritmo: Como apresentado acima, este algoritmo tem complexidade de $O(k \times n^{1/k})$, onde n é o número de degraus e k, o número de frascos. Observe no caso particular de $k = 16$ como o tempo necessário para executar instâncias maiores. Os dados de execução completos encontram-se em *Grficos.xlsx* sob a pasta *Frascos*.

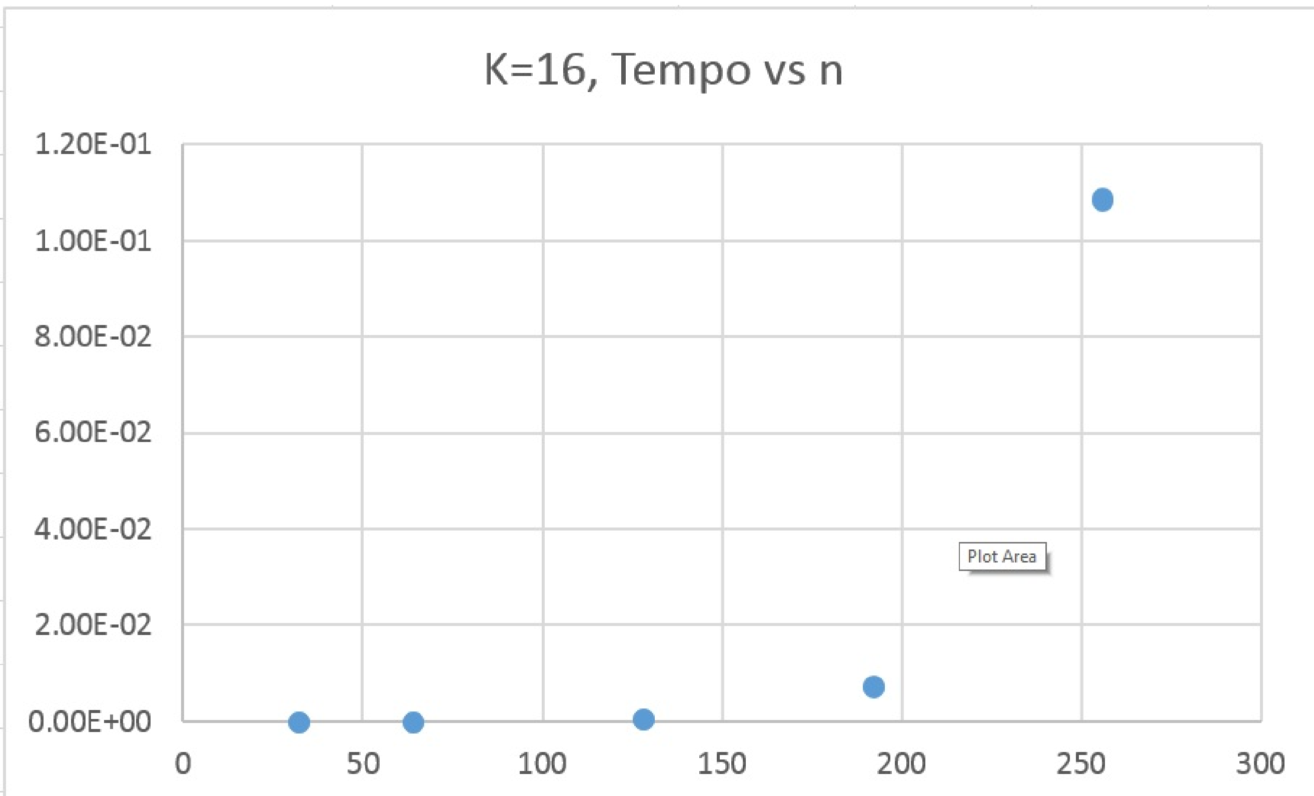

Figure 1: Grafico de Tempo por tamanho da entrada.

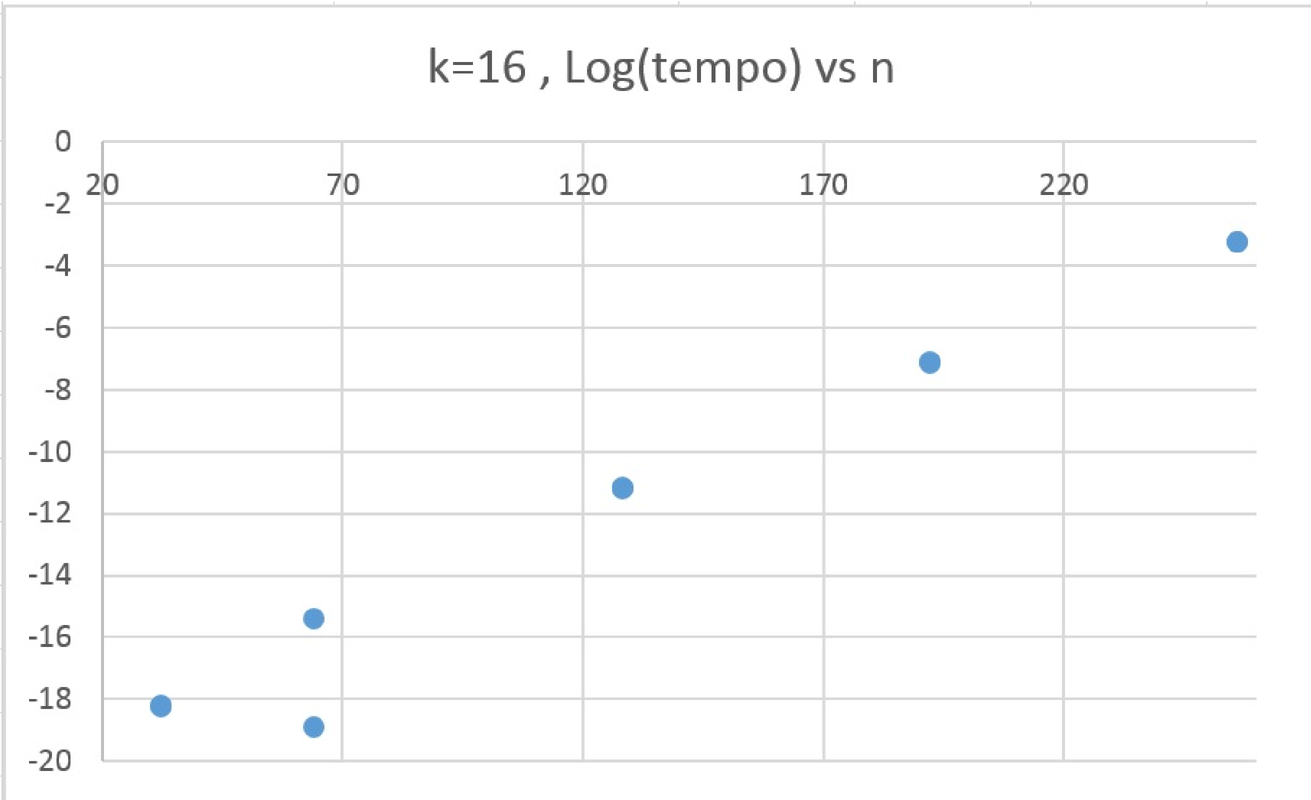

Figure 2: Grafico de Log(tempo) por tamanho da entrada.

Problema da Mochila Fracionária

Descrição do Problema: O problema consiste em determinar quais objetos pode ser carregados, integral ou fracionalmente, em uma mochila com capacidade máxima W, a partir de um conjunto de n objetos divisíveis. Cada objeto possui um peso positivo w_i e um valor positivo v_i, que contribui $x_i * w_i$ para o peso total e $v_i \times x_i$ para o valor total transportados na mochila, em que x_i é uma fração do objeto. O objetivo deste problema é maximizar o valor total transportado, respeitando-se o peso máximo da mochila.

Algoritmo $O(n \log n)$

Descrição do algoritmo: Para maximizar o valor transportado precisamos que cada objeto contribua o máximo possível para o valor total da mochila. Para este algoritmo, primeiramente, calculam-se as densidades de cada objeto (v_i/w_i), os quais são então ordenados de forma crescente com base neste valor. Em seguida, adicionam-se os objetos em ordem descrescente na mochila, até que se atinja o peso máximo. Caso o último objeto a ser inserido tenha peso inferior ao peso disponível na mochila, insere-se uma fração de tal objeto para que se preencha completamente a mochila. Este algoritmo pode ser implementado pelo seguinte código em C++:

```
void kpfrac(Object * obj)
{
    int i, weight = 0;

    std::sort(obj, obj + num_elem);

    i = num_elem - 1;
    while (i >= 0 && weight < W)
    {
        if (weight + obj[i].weight <= W)
        {
```

```
            obj[i].frequency = 1.0;
            weight += obj[i].weight;
        } else {
            obj[i].frequency = (W - weight) * 1.0 / obj[i].weight;
            weight = W;
        }
        i--;
    }
}
```

Análise de complexidade do algoritmo: O algoritmo pode ser separado em duas partes: ordenação e iteração sobre o vetor ordenado. A primeira parte é realizada a partir do *sort* da biblioteca padrão de C++11, que garantidamente tem complexidade $O(n \log n)$. E a segunda parte é uma iteração em n objetos que gulosamente adiciona a mochila os objetos em ordem descrecente; esta etapa tem complexidade $O(n)$. Desta maneira, a complexidade total será $O(n \log n + n) = O(n \log n)$.

Algoritmo $O(n)$

Descrição do algoritmo: Primeiramente, como no algoritmo anterior, definimos a densidade (v_i/w_i) para os n elementos, uma tarefa de pré-processamento com custo de $O(n)$. Dado este conjunto de objetos com suas densidades calculadas, encontramos a mediana de todo o conjunto usando a estratégia das medianas da medianas, que tem complexidade $O(n)$.
Em posse da mediana, divide-se o conjunto de objectos em 3 grupos:

1. O primeiro com os elementos menores que a mediana;
2. O segundo com os elementos iguais a mediana;
3. O terceiro com os elementos maiores que a mediana.

Se a soma dos pesos dos elementos do terceiro conjunto for maior que o peso total da mochila, chama-se a função recursivamente sobre este conjunto, descartando-se dessa maneira os elementos menores ou iguais a mediana são ($\approx n/2$ elementos). Se a soma do terceiro conjunto for menor que o peso máximo W, pode-se adicionar todos estes elementos na mochila, etapa em que também eliminamos $\approx n/2$ elementos.
Em seguida, adicionam-se todos os elementos do segundo conjunto quanto possíveis na mochila. Por último, caso o peso máximo ainda não tenha sido atingido, chama-se a função recursivamente sobre o primeiro conjunto, sendo o novo peso igual a subtração da soma dos pesos dos conjuntos 2 e 3 do peso anterior. Este algoritmo pode ser implementado pelo seguinte código em C++:

```
void kpfrac_linear (Object * objects, int length, int weight) {
    // Base case: We have an empty array or a full knapsack, return;
    if (length <= 0 || weight == 0) return;

    Object * R_1, * R_2, * R_3;
    int idx_R_1, idx_R_2, idx_R_3;

    R_1 = new Object[length];
    R_2 = new Object[length];
    R_3 = new Object[length];
```

```
idx_R_1 = idx_R_2 = idx_R_3 = 0;

// Step 1. Taking advantage of `find_kth` side-effects.
Object median = find_median(objects, length); // O(length)

// Step 2
int R_3_weight = 0;
for (int i = 0; i < length; i++) { // O(length)
    // Step 2.1
    if (objects[i] < median) {
        R_1[idx_R_1++] = objects[i];
    }
    // Step 2.2
    else if (objects[i] == median) {
        R_2[idx_R_2++] = objects[i];
    }
    // Step 2.3
    else {
        R_3[idx_R_3++] = objects[i];
        R_3_weight += objects[i].weight;
    }
}

// Step 3
if (R_3_weight > weight) {
    kpfrac_linear(R_3, idx_R_3, weight); // T(length/2)
} else {
    // Step 4
    for (int i = 0; i < idx_R_3; i++) { // O(idx_R_3) < O(length)
```

```
            R_3[i].frequency = 1.0;
            weight -= R_3[i].weight;
            inserted.push_back(R_3[i]);
        }

        // Step 5.
        // Either takes all from `R_2` completely with leftover space or ex
        // the knapsack.
        for (int i = 0; i < idx_R_2 && weight > 0; i++) {
            if (R_2[i].weight < weight) {
                R_2[i].frequency = 1.0;
                weight -= R_2[i].weight;
            } else {
                R_2[i].frequency = weight * 1.0 / R_2[i].weight;
                weight = 0;
            }

            inserted.push_back(R_2[i]);
        }

        // Step 6.
        kpfrac_linear(R_1, idx_R_1, weight); // T(length/2)
    }

    delete [] R_1;
    delete [] R_2;
    delete [] R_3;
}
```

Análise de complexidade do algoritmo: Na execução deste algoritmo realizam-se operações sobre o conjunto de elementos na ordem de $O(n)$, tanto para a etapa de seleção de pivô através do método de mediana das medianas quanto para a etapa de segmentação em grupos segundo o pivô. Para cada chamada recursiva, jogam-se fora $n/2$ elementos. Dessa maneira:

$$T(n) <= T(\frac{n}{2}) + O(n)$$

Aplicando-se o Teorema Mestre para solucionar a recorrência, o pior caso será $O(n)$. Contudo, a criação de tais conjuntos em cada execução do algoritmo contituem operações que aumentam o seu fator constante, prejudicando sua eficiência para conjuntos pequenos.

Variação do algoritmo: Utilizando-se o mesmo algoritmo, iremos adicionar um pivot definido da seguinte da maneira:

$$pivot = \frac{1}{|K|} \sum_{j \in K} \frac{v_j}{w_j}$$

O algoritmo irá mudar somente na etapa em que encontra-se a mediana. Ao invés de se adotar a mediana como pivô, utilizar-se-á a média das densidades dos objetos. Esta heurística é ótima para distribuições aproximadamente normais, nas quais a média e mediana são próximas.

Para esse novo algoritmo o pior caso terá complexidade $O(n^2)$. Isso irá ocorrer quanto este algoritmo receber uma entrada patológica, cuja distribuição de elementos minimiza a quantidade de elementos descartados em cada chamada recursiva. No caso, quando as densidades dos objetos diferirem em ordens de magnitude (e.g. a primeira sendo 10, a segunda 100, a terceira 1000 e assim em diante) ocorrerá que apenas um elemento será eliminado: o de maior densidade, já que a média será maior que o segundo maior elemento. Posteriormente, será necessário aplicar a recorrência sobre os $n-1$ elementos restantes e assim em diante. Portanto, a cada chamada da função o n seria apenas decrescido de 1 elemento. Por esse motivo a complexidade seria:

$$n + (n-1) + (n-2) + ... + 1 = n\frac{(n+1)}{2} = O(n^2)$$

Este problema afeta outros algoritmos que utilizam alguma heurística, pois podem elaborar-se entradas patológicas para atacar a heurística empregada. Afinal, está serve como uma boa aproximação, mas não uma solução perfeita.

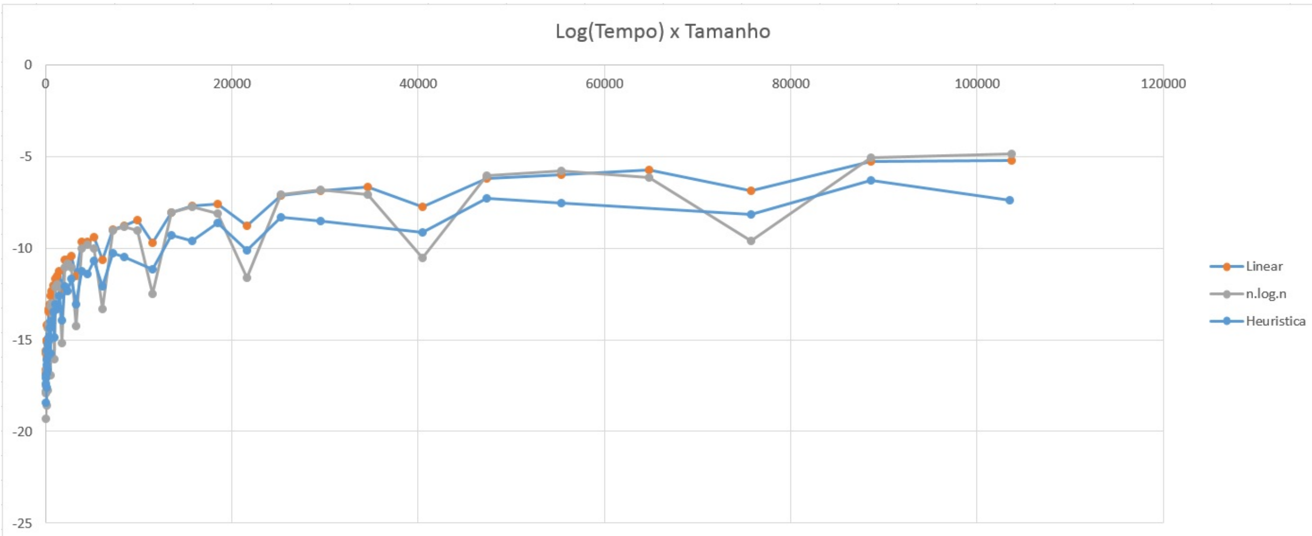

Figure 3: Grafico de Log(tempo) por tamanho da entrada.

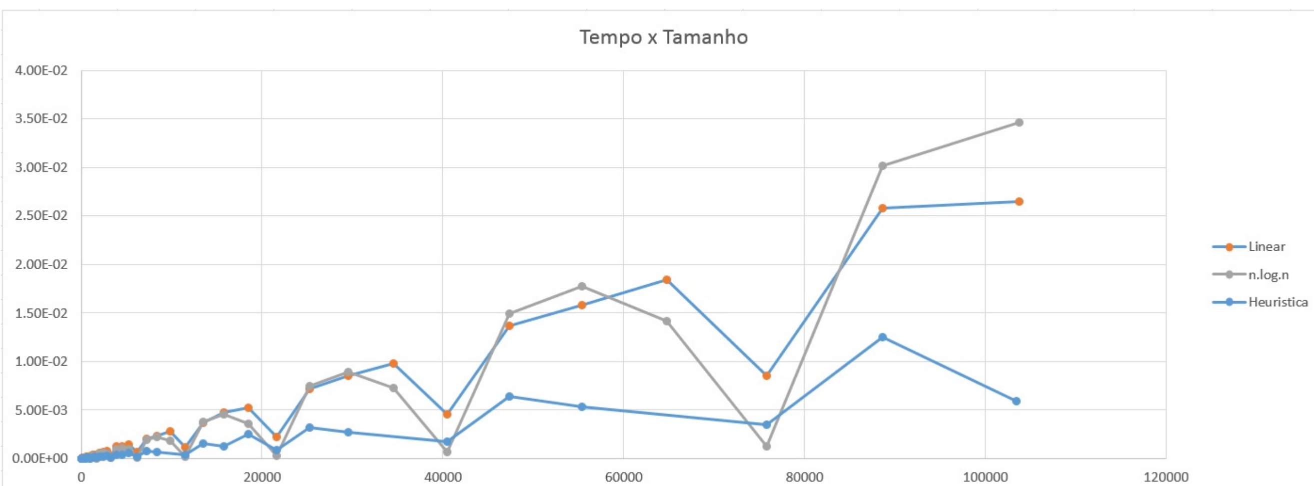

Figure 4: Grafico de tempo por tamanho da entrada.

Observe-se nos gráficos acima que os resultados experimentais dos algoritmos linear por mediana das mediana e o de $n \times log(n)$ foram similares. Isto deve-se ao fato deve-se ao fato do fator constante ser um pouco elevado para o algoritmo linear. Isto se deve aos vetores que são criados para segmentação do vetor de objetos. Note que, para a entrada mais elevada, em que $n \approx 10000$, teríamos uma complexidade na faixa de $100000 \times c \times log100000$. Para

log na base 10, isto resultaria em $500000c$. Portanto, um fator constante de pelo menos 5, produziria resultados comparáveis.

Além disso, é nítido no segundo gráfico como o algoritmo heurístico teve um desempenho significativamente superior aos demais. Isto certamente deve-se ao fato da heurística utilizada: apesar do cálculo realizado ser uma operação de complexidade $O(n)$, seu fator constante é inferior ao do algoritmo de mediana das medianas. Afinal, ambos baseiam-se no mesmo arcabouço lógico, variando-se apenas a forma de seleção de um pivô.