Estruturas de Dados e Algoritmos II

Licenciatura em Engenharia Informática

Trabalho Prático 1

The Dream Factory

UNIVERSIDADE DE ÉVORA

Grupo: g110
Miguel Pombeiro, 57829 | Miguel Rocha, 58501

Departamento de Informática
Universidade de Évora
Março 2025

Índice

1 Introdução

O problema proposto em "The Dream Factory" descreve uma situação em que se pretende minimizar a quantidade de espaço desperdiçado ao embalar um conjunto de sonhos num dado conjunto de números para serem entregues aos clientes. De forma a manter os clientes satisfeitos, os sonhos devem ser entregues pela ordem em que os pedidos foram feitos.

Para implementar um algoritmo que resolva este problema, as principais restrições a ter em conta consistem no facto da ordem dos sonhos ter de ser mantida aquando do empacotamento e que estes têm que ser embalados de forma a minimizar o desperdício total de espaço.

De forma a garantir que estas restrições são cumpridas, será necessário analisar as várias formas em que os sonhos podem ser embalados na ordem correta. Desta forma, seria, ainda possível tomar uma abordagem mais *naive*, com recurso a *brute force*, que iria testar todas as maneiras possíveis de fazer este empacotamento. Contudo, esta abordagem seria computacionalmente cara e, com certeza, não iria cumprir as restrições espaciais e temporais impostas. Assim sendo, e visto tratar-se de um problema de otimização, a solução implementada para a resolução do problema foi feita com recurso à técnica de programação dinâmica.

2 Algoritmo

2.1 *Input*

Os *inputs* disponíveis para cada teste são:

- Número de números disponíveis (**N**) e o número de sonhos (**D**) a ordenar.
- Números disponíveis para embalar os sonhos (n_i).
- Tamanhos dos sonhos a embalar (d_i).

2.2 Descrição do Algoritmo

O algoritmo implementado para resolver este problema utiliza a técnica de programação dinâmica em conjunto com a pesquisa dicotómica, de forma a obedecer às restrições espaciais e temporais impostas no enunciado.
O programa começa por ler os dados do *input* de acordo com a formatação descrita no enunciado. Este dados são guardados em variáveis de inteiros, D e N, que contêm o número de sonhos e o número de números, respetivamente, ou em vetores de inteiros, d e n, que irão conter os tamanhos dos sonhos e os números.
Após esta leitura, e sabendo que serão feitas pesquisas dicotómicas no vetor de números, é necessário que este seja ordenado de forma crescente. Assim, é possível encontrar mais rapidamente o melhor número disponível para acomodar o(s) sonho(s) que se estão a considerar.
Com os passos anteriores já concluídos, é possível é possível dar inicio ao processamento dos dados para encontrar o espaço mínimo desperdiçado. Para tal, e uma vez que se trata de uma versão iterativa de programação dinâmica, é necessário criar uma tabela, onde serão guardados o resultados dos pequenos subproblemas. Neste caso, foi definido um vetor de inteiros, $w[D+1]$, onde em cada índice será armazenado o menor desperdício total possível ao embalar os primeiros n, sonhos. Este vetor é inicializado com o caso base, $w[0] = 0$, ou seja, sem sonhos para embalar, não há espaço desperdiçado. Os restantes índices do *array* devem ser inicializados a $+\infty$, uma vez que se pretende encontrar um resultado mínimo.
De seguida, e de forma a preencher a tabela, é necessário iterar por cada um dos sonhos tentando embalar cada um deles da melhor forma possível. Assim, para cada sonho na lista, são verificadas todas as formas possíveis de os agrupar com os sonhos que os antecedem. Para tal, é calculada a soma do tamanho do sonho atual com os anteriores, que se estão considerar, e é procurado no vetor de números, o menor número possível que possa acomodar essa soma (o menor número igual ou superior à soma dos sonhos do intervalo considerado). Para fazer esta procura, como já foi referido, é utilizada a pesquisa dicotómica.

Encontrado um número que consiga comportar a soma dos tamanhos dos sonhos, o espaço desperdiçado é calculado da seguinte forma:

$$desperdício = número - soma\ dos\ sonhos$$

A este valor ainda deverá ser somado o valor do espaço desperdiçado com o melhor empacotamento dos sonhos que não foram considerados para a soma dos sonhos. Este valor é então comparado com o valor que já está guardado no índice atual do vetor w, que contém o desperdício mínimo encontrado até ao momento. Se o valor for inferior, então foi encontrada uma nova forma de embalar que gera menos desperdício de espaço e o valor no vetor é atualizado.
Por fim, quando todos os sonhos já foram empacotados, deverá ser retornado o valor do último índice do vetor w, que irá conter o espaço mínimo desperdiçado.

2.3 Árvore do Problema

De forma a melhor compreender o problema, foram feitas várias tentativas para resolver os exemplos dados no enunciado, que permitiram observar quais seriam os passos necessários para encontrar uma solução adequada.
A Figura 1 ilustra uma tentativa de resolução do exemplo 1 do enunciado, que permitiu averiguar a melhor forma de desenhar o algoritmo para resolver o problema. Na figura, a azul está representado o vetor de sonhos que ainda falta embalar; a laranja a soma de sonhos que se estão a considerar; a verde o desperdício gerado ao embalar os sonhos no menor número possível; a roxo o número de sonhos a empacotar.

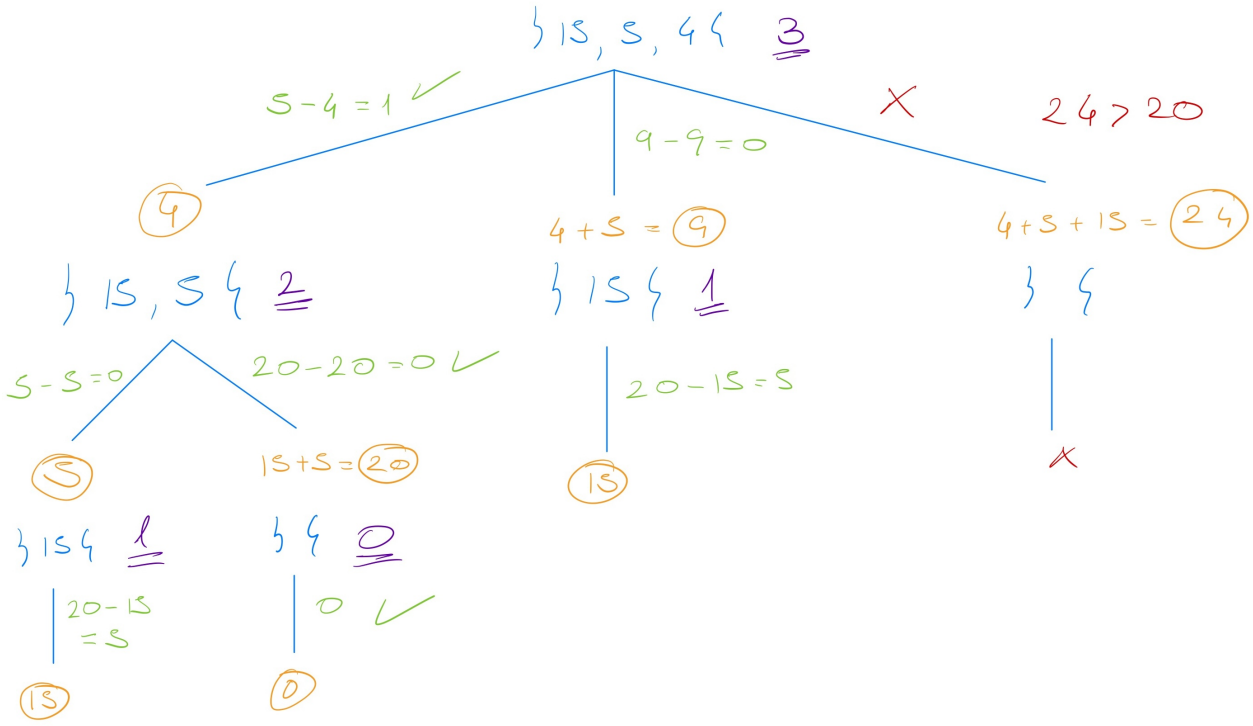

Figura 1: Árvore ilustrativa da resolução do Teste 1 do enunciado.

2.4 Função Recursiva

A base do programa é composta pela seguinte função recursiva, que foi implementada a partir da árvore do problema, Figura 1. O argumento desta, j, representa o número de sonhos que se pretende empacotar. É de notar que, para esta implementação recursiva, o *array* dos sonhos (d) está invertido.

$$w_{nd}(j) = \begin{cases} 0 & \text{se } j = 0 \\ \min\limits_{\substack{0<k\leq j \\ \sum_{l=k}^{j} d_l \leq max(n_i),\ 0<i\leq N}} \{w_{nd}(k-1) + \text{minWaste}(k,\ j)\} & \text{se } j > 0 \end{cases}$$

A função *minWaste(k, j)* é responsável por calcular o menor desperdício possível dos sonhos k a j.

$$minWaste(k,\ j) = \min_{0<i\leq N} \{n_i - \sum_{l=k}^{j} d_l\}$$

2.5 Pseudocódigo

Algorithm 1 Algoritmo de minimização de capacidade desperdiçada para *Dream Factory*

Require: *Array* de Números ordenado, $Numbers$; *Array* of Sonhos, $Dreams$
Ensure: Desperdício de capacidade mínimo

```
1: N ← |Numbers| and D ← |Dreams|
2: let w[0...D] be a new array
3: for i ← 1 to D do
4:     w[i] ← +∞
5: end for
6: w[0] ← 0
7: for j ← 1 to D do
8:     let sum ← 0
9:     for k ← (j − 1) downto 0 do
10:        sum ← sum + Dreams[k]
11:        if sum > max(Numbers) then
12:            break
13:        end if
14:        idx ← PESQUISA-DICOTÓMICA(Numbers, sum)
15:        if idx < N then
16:            w[j] ← min(w[j], w[k] + numbers[idx] − sum)
17:        end if
18:    end for
19: end for
20: Return w[D]
```

3 Complexidades

3.1 Temporal

No algoritmo apresentado, apenas as operações seguintes fazem variar a complexidade temporal:

- Ciclo **3 - 5** é executado D vezes. Em Java, foi utilizado o método fill(*array, value*) do *package* java.util.Arrays, que tem complexidade $\mathcal{O}(D)$.
- Ciclo **7 - 19** é executado D vezes (variável j).
  - Ciclo **9 - 18** é executado, no máximo, j vezes (variável k).
    - **Pesquisa Dicotómica** no *array* de números, que tem uma complexidade temporal de $\mathcal{O}(\log(N))$.

$$\begin{aligned}
\sum_{j=1}^{D}\sum_{k=0}^{j-1} 1 \cdot \log(N) &= \log(N) \cdot \sum_{j=1}^{D}\sum_{k=0}^{j-1} 1 \\
&= \log(N) \cdot \sum_{j=1}^{D} j \\
&= \log(N) \cdot \left(\frac{D \cdot (D+1)}{2} - 0\right) \\
&= \log(N) \cdot \left(\frac{D^2 + D}{2}\right) \\
&= \mathcal{O}(D^2 \cdot \log(N)).
\end{aligned}$$

Na implementação em Java, foi também realizada a ordenação do *array* de números (n) que pode, também, afetar a complexidade temporal. Para isso, foi utilizada a implementação de *quicksort* do *package* java.util.Arrays (método sort(*array*)), que neste caso tem complexidade $\mathcal{O}(N \log N)$.
Para além das operações já descritas, todas as restantes têm custo constante, $\mathcal{O}(1)$.

Assim, o cálculo final de complexidade é o seguinte:

$$\mathcal{O}(D^2 \cdot \log(N) + N \cdot \log(N) + D) = \mathcal{O}(D^2 \cdot \log(N))$$

Uma vez que o ciclo das linhas 7-19 do algoritmo pode ser interrompido quando o maior número não consegue conter os sonhos que se estão a considerar, não é possível determinar um limite inferior preciso para o número de iterações. Assim sendo apenas pode ser utilizada a notação $\mathcal{O}$, que representa o limite superior para a complexidade temporal.

3.2 Espacial

No algoritmo é feito um cálculo iterativo do espaço mínimo desperdiçado, com recurso a tabulação. Para armazenar esta tabela, é utilizado um vetor com dimensão que depende do número de sonhos (D), neste caso, $D+1$. As restantes variáveis utilizadas são números inteiros e não variam com *input*, tendo complexidade espacial $\mathcal{O}(1)$. Assim sendo, a complexidade espacial do algoritmo será $\Theta(D)$.

4 Comentários

Durante a realização do trabalho foram enfrentadas várias dificuldades em desenhar uma árvore do problema, mais concretamente uma árvore que conseguisse ser traduzida para a função recursiva deste problema. Depois de várias tentativas, a solução final foi inspirada na implementação apresentada nas aulas práticas.
Em geral, este programa cumpriu os requisitos propostos, garantindo que os sonhos fossem empacotados na ordem correta e de maneira a minimizar o desperdício de capacidade.
Foi consultado o livro *Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. Introduction to Algorithms, MIT Press* , para mais informações acerca do cálculo das complexidades temporal e espacial, e, de outros exemplos de programação dinâmica.