Estruturas de Dados e Algoritmos II

Licenciatura em Engenharia Informática

1º Trabalho

The Dream Factory

André Gonçalves, 58392 | André Zhan, 58762 | Mooshak: g112

Departamento de Informática
Universidade de Évora
Março 2025

Conteúdo

1 Algoritmo

O objetivo do programa é minimizar o desperdício ao embalar os sonhos nos números disponíveis, garantindo que a ordem dos pedidos seja respeitada. Para isso, o algoritmo segue os seguintes passos principais:

1.1 Leitura dos Dados

- O programa começa por ler os valores de N (número de transportadores disponíveis) e D (número de sonhos encomendados).
- Em seguida, lê a lista de N números disponíveis para embalar os sonhos.
- Depois, lê a sequência de D sonhos, respeitando a ordem em que foram pedidos.

1.2 Solução para o Problema

O problema da Dream Factory, que consiste em empacotar sonhos de diferentes tamanhos em números disponíveis, respeitando a ordem dos sonhos e minimizando o desperdício total, pode ser resolvido eficientemente utilizando uma combinação de programação dinâmica e busca binária.

- **Programação Dinâmica para Desperdício Mínimo:** A programação dinâmica é utilizada para determinar o desperdício mínimo possível ao empacotar os sonhos. O algoritmo constrói uma solução de forma iterativa, considerando subconjuntos de sonhos e escolhendo a melhor maneira de os empacotar para minimizar o desperdício. A ideia-chave é construir uma tabela ‘dp‘, onde ‘dp[i]‘ representa o desperdício mínimo ao empacotar os primeiros ‘i‘ sonhos, considerando que os sonhos são indexados de 0 a ‘i-1‘.
- **Busca Binária para Escolher o Número Ideal:** Para cada subconjunto de sonhos, o algoritmo precisa de encontrar o menor número disponível que pode acomodar todos os sonhos desse subconjunto. A busca binária é empregue para encontrar eficientemente este número ideal dentro do conjunto de números disponíveis. Ordenar os números disponíveis inicialmente permite que a busca binária seja realizada em tempo logarítmico, otimizando o desempenho geral.
- **O Processo Detalhado:**

    1. *Ordenação Inicial:* Os números disponíveis são ordenados para otimizar a busca binária.

2. *Iteração sobre os Sonhos:* O algoritmo itera sobre os sonhos, considerando todos os possíveis subconjuntos de sonhos consecutivos. Os sonhos são indexados de 0 a D-1.
3. *Cálculo do Tamanho Total:* Para cada subconjunto, o tamanho total dos sonhos é calculado.
4. *Busca Binária:* Uma busca binária é realizada para encontrar o menor número disponível que pode acomodar o subconjunto de sonhos.
5. *Cálculo do Desperdício:* O desperdício é calculado como a diferença entre o número escolhido e o tamanho total dos sonhos.
6. *Atualização da Tabela DP:* A tabela 'dp' é atualizada para refletir o menor desperdício encontrado ao empacotar aquele subconjunto de sonhos.

- **Benefícios da Abordagem:** A combinação de programação dinâmica e busca binária garante que o algoritmo encontre a solução ótima para o problema da Dream Factory, minimizando o desperdício total e respeitando a ordem dos sonhos. A programação dinâmica garante que todas as combinações possíveis sejam consideradas, enquanto a busca binária otimiza a escolha do número ideal para cada subconjunto de sonhos.

1.3 Função Recursiva

A função recursiva que define o desperdício mínimo pode ser expressa da seguinte forma:

$$dp[i] = \begin{cases} 0, & \text{se } i = 0 \\ \min\limits_{i \leq k < D} \left\{ dp[k] + \left(num[\text{indice}] - \sum\limits_{j=i}^{k} sonhos[j] \right\} \right\}, & \text{se } i > 0 \end{cases}$$

Onde:

- $dp(i)$ representa o desperdício mínimo ao empacotar os primeiros i sonhos.
- O caso base ocorre quando não há sonhos a serem empacotados ($i = 0$), resultando em desperdício zero.
- Para cada sonho i, tentamos encontrar o ponto k onde um número disponível pode embalar o subconjunto de sonhos $\{i, ..., k\}$, minimizando o desperdício.

1.4 Cálculo Iterativo

Algorithm 1: encontrarDesperdicioMinimo($N, D, numeros, sonhos$)

```
Input: N, D, numeros, sonhos
Output: desperdicioMinimo
Ordenar numeros ;
INF ← Integer.MAX_VALUE ;
dp[0..D] ← INF ;
dp[0] ← 0 ;
for i ← 0 to D − 1 do
    tamanhoTotal ← 0 ;
    for j ← i to D − 1 do
        tamanhoTotal ← tamanhoTotal + sonhos[j] ;
        indice ← buscaBinaria(numeros, tamanhoTotal) ;
        if indice ≠ −1 then
            desperdicio ← numeros[indice] - tamanhoTotal ;
            dp[j+1] ← min(dp[j+1], dp[i] + desperdicio) ;
        end
    end
end
return dp[D] ;
```

Algorithm 2: buscaBinaria(numeros, alvo)

```
Input: numeros, alvo
Output: indice
esquerda ← 0 ;
direita ← numeros.length - 1 ;
while esquerda ≤ direita do
    meio ← ⌊(esquerda + direita)/2⌋ if numeros[meio] ≥ alvo then
        direita ← meio - 1 ;
    end
    else
        esquerda ← meio + 1 ;
    end
end
if esquerda < numeros.length then
    return esquerda ;
end
else
    return-1 ;
end
```

2 Análise de Complexidade

Para avaliar a eficiência do programa, analisamos a sua complexidade temporal e espacial, tendo em conta os métodos e estruturas de dados utilizados em Java. A análise será expressa em função dos parâmetros N (número de transportadores disponíveis) e D (número de sonhos encomendados).

2.1 Complexidade Temporal

A complexidade do programa pode ser analisada separando as principais operações executadas:

1. **Ordenação dos transportadores:** Antes de processarmos os sonhos, ordenamos o array de transportadores utilizando o método `Arrays.sort()`, que implementa o algoritmo *Dual-Pivot Quicksort* para arrays de inteiros. Este algoritmo tem complexidade temporal de:
$$O(N \log N)$$

2. **Preenchimento do array dp com programação dinâmica e busca binária:** O programa percorre cada sonho e tenta combiná-lo com sonhos subsequentes, acumulando os seus tamanhos. Para cada combinação possível, realiza uma *busca binária* para encontrar o menor transportador que os possa acomodar.

   O pior caso ocorre quando cada sonho é combinado com todos os sonhos subsequentes, o que resulta num total de aproximadamente $O(D^2)$ iterações. Como cada iteração contém uma busca binária ($O(\log N)$), a complexidade total deste processo é:
$$O(D^2 \log N)$$

3. **Cálculo final e saída:** A obtenção do valor final de $dp[D]$ e a sua impressão têm complexidade constante $O(1)$, pelo que podem ser ignoradas na análise assintótica.

Assim, a complexidade temporal global do programa é dominada pela fase de programação dinâmica com busca binária:

$$O(N \log N + D^2 \log N)$$

No pior caso, quando D é significativamente maior que N, o termo $O(D^2 \log N)$ domina a complexidade.

2.2 Complexidade Espacial

A complexidade espacial do programa depende das estruturas de dados utilizadas:

- O array de transportadores ocupa $O(N)$.
- O array de sonhos ocupa $O(D)$.
- O array *dp*, utilizado para armazenar o desperdício mínimo para cada subconjunto de sonhos, ocupa $O(D)$.

Como não são utilizadas estruturas auxiliares significativas, a complexidade espacial total do programa é:

$$O(N + D)$$

Isto significa que o programa utiliza memória proporcional ao número de transportadores e ao número de sonhos encomendados.

2.3 Conclusão

O programa apresenta uma complexidade temporal de $O(N \log N + D^2 \log N)$, sendo adequado para os limites estabelecidos no enunciado ($N \leq 100$, $D \leq 100000$). A complexidade espacial é $O(N + D)$, garantindo um uso eficiente da memória.

3 Referências

Referências

[1] Slides das aulas teóricas.

[2] Oracle. *Java Platform, Standard Edition Documentation*. Disponível em: `https://docs.oracle.com/javase/`

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

[4] Complexidade de Algoritmos em 3 simples passos utilizando a Notação Big O. Disponível em: `https://www.youtube.com/watch?v=zXBaLEGv0iM`