University of Houston

Homework 2 Solutions

COSC 3320
Algorithms and Data Structures

Due: Sunday, March 3, 2024
11:59 PM

1 Exercises

Exercise 1: Sweepy the Robot (20 Points)

`Sweepy` is a robot that cleans a carpet modeled as an $m \times n$ grid, `carpet`. If $\texttt{carpet}[i, j] = 1$, then the carpet at position (i, j) is dirty. Otherwise, we have $\texttt{carpet}[i, j] = 0$. `Sweepy` starts at cell (0, 0) and can travel right or down.

Give a dynamic programming algorithm that maximizes the number of dirty cells visited by `Sweepy`.

1. Define the subproblems for your solution.
2. Give a recursive formulation to solve the subproblems. Include the base cases.
3. Give pseudocode for a memoized, top-down algorithm that outputs this maximum.
4. Give pseudocode for an efficient, bottom-up algorithm that outputs this maximum.
5. What is the runtime of your solution? Justify your answer.
6. Explain how to modify your algorithm to output the path taken by `Sweepy`.

Solution.

1. Let $\textsc{max-dirt}(i, j)$ denote the maximum number of dirty cells cleaned by `Sweepy` starting at cell $(0, 0)$ and *ending* at cell (i, j).

2. The only way `Sweepy` can reach cell (i, j) is from cell $(i-1, j)$ or from cell $(i, j-1)$. Thus, we have

$$\textsc{max-dirt}(i, j) = \texttt{carpet}[i, j] + \max(\textsc{max-dirt}(i-1, j), \textsc{max-dirt}(i, j-1))$$

Our base cases are $\textsc{max-dirt}(0, 0) = \texttt{carpet}[0, 0]$ and

$$\begin{aligned} \textsc{max-dirt}(0, j) &= \textsc{max-dirt}(0, j-1) + \texttt{carpet}[0, j] \\ \textsc{max-dirt}(i, 0) &= \textsc{max-dirt}(i-1, 0) + \texttt{carpet}[i, 0] \end{aligned}$$

Note that these last two lines can be considered part of the recursive formulation as well.

3.

```
def MAX-DIRT(carpet):
    m, n = DIMENSIONS(carpet)
    cache is a 2D lookup table with cache[0, 0] == carpet[0, 0]
    ▷ Fill first row of table
    for j in 1...n:
        cache[0, j] = cache[0, j − 1] + carpet[0, j]
    ▷ Fill first column of table
    for i in 1...m:
        cache[i, 0] = cache[i − 1, 0] + carpet[i, 0]

    def MAX-DIRT-HELPER(i, j):
        if (i, j) ∉ cache:
            above = MAX-DIRT-HELPER(i − 1, j)
            left = MAX-DIRT-HELPER(i, j − 1)
            cache[i, j] = max(above, left) + carpet[i, j]
        return cache[i, j]

    return MAX-DIRT-HELPER(m − 1, n − 1)
```

4.

```
def MAX-DIRT(carpet):
    m, n = DIMENSIONS(carpet)
    cache is a 2D array with cache[0, 0] == carpet[0, 0]
```

```
    ▷ Fill first row of table
    for j in 1 ... n:
        cache[0, j] = cache[0, j − 1] + carpet[0, j]

    ▷ Fill first column of table
    for i in 1 ... m:
        cache[i, 0] = cache[i − 1, 0] + carpet[i, 0]

    ▷ Fill table
    for i in 1 ... m:
        for j in 1 ... n:
            above = cache[i − 1, j]
            left = cache[i, j − 1]
            cache[i, j] = max(above, left) + carpet[i, j]

    return cache[m − 1, n − 1]
```

5. There are $m \times n$ subproblems, each requiring $\mathcal{O}(1)$ time to solve, for $\mathcal{O}(mn)$ in total.

6. Set $i, j = m-1, n-1$. Until $i, j == 0, 0$, look at the maximum dirt that can be cleaned by reaching $(i-1, j)$ and $(i, j-1)$ — visit the location corresponding to the maximum of the two. In pseudocode:

```
i, j = m − 1, n − 1
path = []
while i, j ≠ 0, 0:
    append (i, j) to path
    above = cache[i − 1, j]
    left = cache[i, j − 1]
    if above > left:
        i = i − 1    ▷ Go up
    else:
        j = j − 1    ▷ Go left
```

□

Exercise 2: Hotel (20 Points)

There are n hotels, $\texttt{hotel}_0$, $\texttt{hotel}_1$, ..., $\texttt{hotel}_{n-1}$. You start on a trip at $\texttt{hotel}_0$ and wish to reach $\texttt{hotel}_{n-1}$. From any $\texttt{hotel}_i$, you can travel to any $\texttt{hotel}_j$ (with $j > i$) at a cost of $\textsc{penalty}(i, j)$. Your *total penalty* for a sequence of stops is the sum of the penalty for each trip. For example, if $n = 4$, one possible sequence of stops is

$$\texttt{hotel}_0 \to \texttt{hotel}_2 \to \texttt{hotel}_3$$

which incurs total penalty

$$\textsc{penalty}(0, 2) + \textsc{penalty}(2, 3)$$

Design a dynamic programming algorithm to minimize the total penalty to travel from $\texttt{hotel}_0$ to $\texttt{hotel}_{n-1}$.

1. Define the subproblems for your solution.
2. Give a recursive formulation to solve the subproblems. Include the base cases.
3. Give pseudocode for a memoized, top-down algorithm that outputs this minimum.
4. Give pseudocode for an efficient, bottom-up algorithm that outputs this minimum.
5. What is the runtime of your solution? Justify your answer.
6. Explain how to modify your algorithm to also output the optimal sequence of hotels.

Solution.

1. Let $\textsc{min-cost}(i)$ denote the minimum total penalty to travel from $\texttt{hotel}_0$ to $\texttt{hotel}_i$.

2. Consider any path ending at $\texttt{hotel}_{n-1}$ — the path must involve a direct trip from some hotel $\texttt{hotel}_k$ to $\texttt{hotel}_{n-1}$

$$\underbrace{\texttt{hotel}_0 \to \cdots \to \texttt{hotel}_k}_{\texttt{hotel}_0 \text{ to } \texttt{hotel}_k} \underbrace{\to \texttt{hotel}_{n-1}}_{\texttt{hotel}_k \text{ to } \texttt{hotel}_{n-1}}$$

   By definition, the optimal path to reach $\texttt{hotel}_k$ from $\texttt{hotel}_0$ is $\textsc{min-cost}(k)$, hence we must have

$$\textsc{min-cost}(i) = \min_{0 \le k < i} \{\textsc{min-cost}(k) + \textsc{penalty}(k, i)\}$$

   with base case $\textsc{min-penalty}(0) = 0$.

3. **def** $\textsc{Min-Cost}(\texttt{hotels})$:
   $\quad n = \textsc{length}(\texttt{hotels})$
   $\quad \texttt{cache} = \{0 : 0\}$

   $\quad$ **def** $\textsc{Min-Cost-Helper}(i)$:
   $\quad\quad$ **if** $i \notin \texttt{cache}$:
   $\quad\quad\quad \texttt{cache}[i] = \min_{0 \le k < i} \{\textsc{Min-Cost-Helper}(k) + \textsc{penalty}(k, i)\}$

   $\quad\quad$ **return** $\texttt{cache}[i]$

   $\quad$ **return** $\textsc{Min-Cost-Helper}(n - 1)$

4. **def** $\textsc{Min-Cost}(\texttt{hotels})$:
   $\quad n = \textsc{length}(\texttt{hotels})$
   $\quad \texttt{cache} = [0,]$
   $\quad$ **for** i **in** $1 \ldots n$:
   $\quad\quad \texttt{cache}[i] = \min_{0 \le k < i} \{\textsc{Min-Cost-Helper}(k) + \textsc{penalty}(k, i)\}$

   $\quad$ **return** $\texttt{cache}[n - 1]$

5. There are n subproblems, each of which requires $\mathcal{O}(n)$ time, for a total of $\mathcal{O}(n^2)$.

6. Set $i = n - 1$. While $i \neq 0$, find the value of k which minimizes

$$\textsc{min-cost}(k) + \textsc{penalty}(k, i)$$

and set $i = k$. The sequence of i values is our sequence of stops. In pseudocode:

$i = n - 1$
`path` $= []$
while $i \neq 0$:
    append i to `path`
    set i to the value that minimizes $\textsc{min-cost}(k) + \textsc{penalty}(k,\, i)$

$\square$

Note: an alternative formulation is to let $\textsc{min-cost}(i,\, j)$ denote the minimum penalty to travel from $\texttt{hotel}_i$ to $\texttt{hotel}_j$. This has almost the same solution as the Matrix Chain Multiplication problem, with

$$\textsc{min-cost}(i,\, j) = \min\left(\min_{i<k<j}(\textsc{min-cost}(i,\, k) + \textsc{min-cost}(k,\, j)),\, \textsc{penalty}(i,\, j)\right)$$

and base case

$$\textsc{min-cost}(i,\, i+1) = \textsc{penalty}(i,\, i+1)$$

Exercise 3: Bin Packing (20 Points)

The *Bin Packing Problem* is as follows:

Given a set of n objects of sizes $s_0, s_1, \ldots, s_{n-1}$, each satisfying $0 < s_i < 1$, and an unlimited supply of bins of capacity 1, pack all n objects into bins such that the total size of the objects in each bin *does not exceed 1* and the number of bins is **minimized**.

Consider the following greedy algorithm: Assume you have already placed objects $0, 1, \ldots, k-1$. Place object k in a bin that has the *maximum* amount of *available* space. If no such bin exists, place the object in a new bin.

1. Describe how to efficiently implement this algorithm using a binary heap.
2. Analyze the runtime of this algorithm.
3. Show that this greedy algorithm does not always return the optimal solution.

Solution.

1. Initialize a max-heap with an empty bucket. For each item, inspect the top element — if it has enough space, append the item to the bucket and heapify. Otherwise, push a new bucket containing only the new item to the heap.
2. In the worst case, each item is placed it its own bucket, hence the worst case runtime is $\mathcal{O}(n \log n)$.
3. Take weights $\{0.9, 0.05, 0.95, 0.1\}$. Our algorithm outputs

$$\underbrace{\{0.9, 0.05\}, \{0.95\}, \{0.1\}}_{\text{3 bins}}$$

but the optimal output is

$$\underbrace{\{0.9, 0.1\}, \{0.95, 0.05\}}_{\text{2 bins}}$$

□

Exercise 4: Shopping (20 Points)

A shop sells n collectibles at prices $c_0, c_1, \ldots, c_{n-1}$. You want to purchase all n collectibles, but are only able to purchase one each month. Unfortunately, the price of each item *doubles* each month.

1. Give a greedy algorithm to minimize the total cost to purchase all n items.
2. Prove that your greedy algorithm always outputs the optimal solution.

Solution.

1. Simply buy the items in decreasing order of cost.
2. In order to reach a contradiction, assume there exists an optimal ordering, $\mathcal{O}$, that is strictly better than that output by the greedy algorithm, G. Say

$$G = g_0, g_1, \ldots, g_{n-1}$$
$$\mathcal{O} = f_0, f_1, \ldots, f_{n-1}$$

Let i be the first index where $g_i \neq f_i$. Since the sets $\{g_0, g_1, \ldots, g_{n-1}\}$ and $\{f_0, f_1, \ldots, f_{n-1}\}$ are the same, there must be an index $j > i$ such that, $f_j = g_i$. Since the values $g_0, g_1, \ldots, g_{n-1}$ are in decreasing order, we must also have that $f_i < g_i$. Thus, $f_i < f_j$.

Now, construct a new ordering $\mathcal{O}'$ by swapping f_i and f_j:

$$\mathcal{O} = f_0, f_1, \ldots, f_i \ldots, f_j \ldots, f_{n-1}$$
$$\mathcal{O}' = f_0, f_1, \ldots, f_j \ldots, f_i \ldots, f_{n-1}$$

The cost for each is given by:

$$\textsc{Cost}(\mathcal{O}) = f_0 + 2f_1 + \cdots + 2^i f_i + \cdots + 2^j f_j + \cdots + 2^{n-1} f_{n-1}$$
$$\textsc{Cost}(\mathcal{O}') = f_0 + 2f_1 + \cdots + 2^i f_j + \cdots + 2^j f_i + \cdots + 2^{n-1} f_{n-1}$$

Now consider their difference, $\textsc{Cost}(\mathcal{O}) - \textsc{Cost}(\mathcal{O}')$:

$$\begin{aligned}
\textsc{Cost}(\mathcal{O}) - \textsc{Cost}(\mathcal{O}') &= \left(2^i f_i + 2^j f_j\right) - \left(2^i f_j + 2^j f_i\right) \\
&= 2^i f_i + 2^j f_j - 2^i f_j - 2^j f_i \\
&= f_i\left(2^i - 2^j\right) + f_j\left(2^j - 2^i\right) \\
&= f_i\left(2^i - 2^j\right) - f_j\left(2^i - 2^j\right) \\
&= (f_i - f_j)\left(2^i - 2^j\right)
\end{aligned}$$

Now, since $i < j$, $2^i - 2^j$ is negative. Similarly, since $f_i < f_j$, $f_i - f_j$ is negative. Thus

$$\textsc{Cost}(\mathcal{O}) - \textsc{Cost}(\mathcal{O}') = (f_i - f_j)\left(2^i - 2^j\right) > 0$$

hence

$$\textsc{Cost}(\mathcal{O}) > \textsc{Cost}(\mathcal{O}')$$

This contradicts the optimality of $\mathcal{O}$. Thus, there can be no ordering strictly better than G, hence G must be optimal.

□

Exercise 5: Knapsack (20 Points)

This following exercise will compare the tradeoffs between top-down and bottom-up dynamic programming. Consider the 0/1 Knapsack problem.

1. Implement a memoized, top-down solution to the 0/1 Knapsack problem.
2. Implement an efficient, bottom-up solution to the 0/1 Knapsack problem.
3. Run both implementations on varying inputs of your choice. Describe the performance differences between the two implementations.
4. Describe the differences in correctly implementing both versions. For example, what sort of errors did you encounter in each implementation? Which version took longer to implement correctly?
5. What are the overall tradeoffs between the two versions?