Classical Music Festival

Rishubh Thaper

November 15, 2020

1 Problem Description

You are attending an international classical music festival in Vienna! Of course, as an aficionado, you are beyond excited to be in one of the most famous cities in the world and the centuries-old nucleus of classical music.

The festival takes place over two months (!), and each day features a multitude of events located throughout the city, including piano recitals, symphonic concerts, and chamber performances; each one carries a separate cost of admission. You have access to all of the programs for the events and can estimate the amount of time each event will last.

With this information, you have created a list of events you are interested in throughout the course of the festival along with the duration and cost of admission of each one (fortunately, each event takes place on a different day so you can attend each one if you want to!). Below is an example:

1. ***A Celebration of Frederic Chopin***: 60 minutes, $15
2. ***Contemporary Symphonic Music***: 120 minutes, $25
3. ***Beethoven's Fifth Symphony***: 40 minutes, $8
4. ***Selected 18th Century String Quartets***: 75 minutes, $15
5. ***Tchaikovsky's Sixth***: 65 minutes, $20

Since you have a limited budget, you want to maximize the amount of time spent listening to beautiful music but remain under budget. Continuing on with the example above, suppose you can spend a maximum of $50. Then you can listen to at most 235 minutes of music by attending the second event, Contemporary Symphonic Music, the third event, Beethoven's Fifth Symphony, and the fourth event, Selected 18th Century String Quartets. You still have $2 left over at the end, but you do not have enough money to attend another event.

Write a function

```
int maxMinutes(Vector<pair<int, int>> events, int budget)
```

that accepts as input a `Vector<pair<int, int>>` containing ordered pairs of the form (minutes, cost) holding the duration and cost of each event, along with your budget for the entire festival, and then returns the maximum number of minutes of music you can listen to by staying under budget. For the example above `events` will contain the `pairs`

$$\{60, 15\}, \{120, 25\}, \{40, 8\}, \{75, 15\}, \{65, 20\}.$$

When writing your function, you should note the following:

- It may be that all events cost more than your budget, in which case your function should return 0.
- Although unlikely, you may not be interested in any events at all, so the input vector will be empty and your function should also return 0.
- You can assume that the duration and cost of each event will be integers.

In addition, since you are such an avid fan of the genre, you will most likely be interested in a large number of events (no worries, you have determined that all events you have selected are reachable!), and because the festival so your function will need to run in under a second on the following test sets, where N is the size of `events` and M is the amount of `money` .

- **Test Set 1**: $N \leq 10$, $M \leq 20$,
- **Test Set 2**: $N \leq 50$, $M \leq 100$,
- **Test Set 3**: $N \leq 500$, $M \leq 1000$

```
int maxMinutes(Vector<pair<int, int>> events, int budget) {
  /* TODO: Fill me in! */
  return 0;
}
```

2 Solution I: Recursive Backtracking

Analysis

For each event $1 \leq i \leq N$, where N is the total number of events you are interested in, let d_i be its duration and c_i be its cost, and suppose your budget is $M \geq 0$. Then the vector

`events` contains the pairs

$$\{d_1, c_1\}, \{d_2, c_2\}, \ldots, \{d_N, c_N\}.$$

Let $b_i \in \{0, 1\}$ be the bit that represents whether we include event i in our itinerary; that is, $b_i = 0$ if we choose not to attend event i, and $b_i = 1$ if we do choose to attend event i. Then, we want to maximize the value of $\sum_{i=1}^{N} b_i d_i$ given that $\sum_{1 \leq i \leq n, b_i = 1} c_i \leq M$. This is simply a variant of the famous Knapsack Problem, so we can implement a recursive backtracking algorithm to explore all possible subsets of events and compute the total cost across all events in each subset. Disregarding any subset that has a total cost greater than M, we pick the subset for which the total duration of its events is maximal, and return that value.

Assuming all necessary `includes` have been executed, the code is as follows:

```
int maxMinutesNaive(Vector<pair<int, int>> events, int money,
int minutes, int index) {
  if (money < 0) {
    return -1;
  }
  if (index == events.size()) {
    return minutes;
  }
  return max(maxMinutesNaive(events, money - events[index].second,
  minutes + events[index].first, index + 1),
  maxMinutesNaive(events, money, minutes, index + 1));
}

/* Wrapper function that calls the helper function at index 0. */

int maxMinutesNaive(Vector<pair<int, int>> events, int money)
{
  return maxMinutesNaive(events, money, 0, 0);
}
```

The helper function keeps tracking of the running total duration in the parameter `minutes`. The Big-O of this function is `O(2^n)`.

3 Solution II: Memoization

Analysis

The naive backtracking solution is adequate for small inputs, but for $N \geq 10$, the function quickly becomes time-consuming. If you are interested in a large number of events throughout

the course of the entire two months that the festival lasts, it will be painstakingly slow to calculate your optimal itinerary.

For this reason, a more efficient approach is required. We note that the problem can be divided into NM distinct states of the form `(index, money)`, where `money` is the amount of money remaining after you have considered all events up to position `index`.

Moreover, since this problem exhibits *optimal substructures* (an optimal solution includes the optimal choices at each step) and *overlapping subproblems* (meaning the same state can be explored more than once), we can simply keep store the values at each state in a *memoization table* and return them when encountered to prune for states that have already been visited. This is known as top-down **dynamic programming**.

```
int maxMinutesMemo(Vector<pair<int, int>> events, int money,
int index, Grid<int>& memo) {
  if (money < 0) {
    return -1;
  }
  if (index == events.size()) {
    return 0;
  }
  if (memo[index][money] != -1) {
    return memo[index][money];
  }
  if (events[index].second > money) {
    return memo[index][money] = maxMinutesMemo(events, money,
    index + 1, memo);
  }
  else {
    return memo[index][money] = max(events[index].first +
    maxMinutesMemo(events, money - events[index].second, i
    ndex + 1, memo),
    maxMinutesMemo(events, money, index + 1, memo));
  }
}

/* Main wrapper function that initializes the memo
 * table, then calls the helper function.
*/

int maxMinutesMemo(Vector<pair<int, int>> events, int money) {
  Grid<int> memoTable(events.size(), money + 1);
  for (int i = 0; i < events.size(); i++) {
```

```
    for (int j = 0; j <= money; j++) {
      memoTable[i][j] = -1;
    }
  }
  return maxMinutesMemo(events, money, 0, memoTable);
}
```

4 Solution 3: Dynamic Programming

Analysis

Bottom-up dynamic programming avoids the overhead of recursive calls by preparing a table ahead of time that contains the values of a certain mathematical function.

In this problem, we define the function $f(i, m)$ as the maximum possible amount of minutes spent attending the festival if events at indices $0, 1, 2, \ldots, i-1$ have all been considered and at most m dollars have been spent. Then the answer that is returned is $f(N, M)$.

It is easy to see the recurrence relation that such a function satisfies: $f(i, m)$ is equal to the maximum of $f(i-1, m)$ and $f(i-1, m-c_{i-1}) + d_{i-1}$. For the base cases, we tabulate $f(0, m) = 0$ for all values of $m \leq M$.

In fact, since the way a row is filled depends only on the row directly above, we can actually create a $2 \times (M+1)$ table, simply replacing row 0 with the values in row 1 at the end of each inner loop. This is a clever space-saving trick that greatly reduces the memory usage.

The code is as follows:

```
int maxMinutesDP(Vector<pair<int, int>> events, int money) {
  int dp[2][money + 1];
  memset(dp, 0, sizeof(dp));
  for (int i = 0; i <= events.size(); i++) {
    for (int j = 0; j <= money; j++) {
      if (i == 0 || j == 0) {
        continue;
      }
      else if (events[i - 1].second <= j) {
        dp[1][j] = max(dp[0][j - events[i - 1].second] +
                       events[i - 1].first, dp[0][j]);
      }
      else {
        dp[1][j] = dp[0][j];
      }
    }
    for (int j = 0; j <= money; j++) {
```

```
        dp[0][j] = dp[1][j];
      }
    }
    return dp[1][money];
}
```

The time complexity of this solution is `O(N*M)` and the space complexity is `O(2*M)` .

However the dynamic programming function does sacrifice some stylistic clarity, as it is much easier to see the recursive nature of the problem through the backtracking function, with its lucid division of base cases and recursive case and obvious path of exploration. Nevertheless, the time and space efficiency of dynamic programming firmly outweighs its stylistic trade-off.

5 Test Cases

I tested my function on the following cases to ensure that each version works as intended. See source code for more exhaustive test cases.

```
STUDENT_TEST("Test all three functions on small example input") {
  musicEvents = {{60, 15}, {120, 25}, {40, 8}, {75, 15}, {65, 20}};
  EXPECT_EQUAL(maxMinutesNaive(musicEvents, 50), 235);
  EXPECT_EQUAL(maxMinutesMemo(musicEvents, 50), 235);
  EXPECT_EQUAL(maxMinutesDP(musicEvents, 50), 235);
}

STUDENT_TEST("Test all three versions on input with overlapping
paths") {
  // Multiple paths to 15 dollars spent
  musicEvents = {{52, 13}, {40, 10}, {28, 7}, {35, 6}, {30, 5},
        {15, 3}, {11, 2}, {5, 1}};

  EXPECT_EQUAL(maxMinutesNaive(musicEvents, 15), 85);
  EXPECT_EQUAL(maxMinutesMemo(musicEvents, 15), 85);
  EXPECT_EQUAL(maxMinutesDP(musicEvents, 15), 85);
}

STUDENT_TEST("Test all three functions in input in which all events
    cost more than budget") {
  Vector<pair<int, int>> musicEvents = {{40, 25}, {30, 30},
    {60, 22}, {50, 24}, {70, 35}};
  EXPECT_EQUAL(maxMinutesNaive(musicEvents, 20), 0);
  EXPECT_EQUAL(maxMinutesMemo(musicEvents, 20), 0);
  EXPECT_EQUAL(maxMinutesDP(musicEvents, 20), 0);
```

```
}

STUDENT_TEST("Test all three functions on edge cases") {
  // When there are no events, the functions should return 0
  Vector<pair<int, int>> musicEvents = {};
  EXPECT_EQUAL(maxMinutesNaive(musicEvents, 20), 0);
  EXPECT_EQUAL(maxMinutesMemo(musicEvents, 20), 0);
  EXPECT_EQUAL(maxMinutesDP(musicEvents, 20), 0);

  musicEvents = {{75, 15}, {9, 3}, {60, 10}, {24, 8}, {10, 1}};

  // When the budget is 0, the functions should return 0
  EXPECT_EQUAL(maxMinutesNaive(musicEvents, 0), 0);
  EXPECT_EQUAL(maxMinutesMemo(musicEvents, 0), 0);
  EXPECT_EQUAL(maxMinutesDP(musicEvents, 0), 0);
}
```

6 Problem Motivation

Concept Coverage

This question is designed to encourage students to recognize classic recursive problems out of their traditional context. Behind the fun story that the problem tells lies a very familiar Knapsack problem. Specifically, the problem requires proficiency in each of the following:

- Ability to recognize and implement a solution to a famous computer science problem.
- Knowledge of recursive backtracking techniques and implementing helper functions to complete the task.
- Optimizing algorithms for time and space efficiency.

In particular, the traditional recursive backtracking approach fails with input sizes large enough to be reasonable for the context of the problem, so a more efficient solution is absolutely necessary.

Personal Significance

I have always been intrigued by the idea of solving complex optimization problems with only a few lines of code. This is a class of problems I could not even begin to analyze before, and CS106B propelled me to the next level of computational problem-solving with only a couple lectures.

I also was fascinated by the idea of tweaking the basic idea of backtracking to make it time-efficient, and although I had heard of the terms *memoization* and *dynamic programming*, I had never known what they meant. The extension ideas on Assignment 4 encouraged me to research these advanced recursive techniques, and I decided to incorporate all of them into a single problem to demonstrate step-by-step how recursive backtracking leads naturally to the efficiency powerhouse of dynamic programming.

The Knapsack problem, being a classic example that nicely showcases all three recursive techniques, was the best choice for me to illustrate my growth in this area. Thus, I devised a variation of the problem by incorporating my love for classical music and the great composers. Maybe one day, I will use the function myself when planning for a trip to a music festival!

7 Concept Mastery and Common Misconceptions

Recursion and recursive techniques are the primary goals assessed by this problem, although Big-O and time efficiency figures prominently as the input sizes become larger. However, misconceptions and bugs could arise in any one of the following areas of the problem:

- Deciding which parameters to include in the helper function
- Determining the appropriate base cases
- Storing return values in the memoization table
- Exploiting optimal substructures to derive the correct recursive dynamic programming function

The recursive backtracking function works well by keeping track of the money remaining, the total minutes spent, and the index of the current event being considered. However, the memoization function only works as intended when the number of minutes is removed as a parameter and instead, the current event's duration is added to the result of the recursive call instead of into the call itself. This is because the memoization table is used as a pruning mechanism, and keeping the total duration of events chosen inside the function as a parameter results in arms-length recursion where the memoization table may not be updated.

Also, students may incorrectly terminate the function at a base case in which the current index is equal to the index of the last event instead of one more than that (the size of the vector), which does not update the total minutes to include the duration of the last event. Students will realize that they need to continue the recursion past the last event to ensure that it is explored. Additionally, when pruning for subsets that go over budget, a negative value should be returned in the base case instead of 0, which could be a possible answer to

the problem.

Moreover, students may forget to update the memoization table at each recursive call, which never changes the initial values of -1. This is the most important step and that distinguishes top-down dynamic programming from backtracking.

Finally, although it looks deceptively simple, the recursive case is simply the maximum of two recursive calls – one including the event, and the other excluding it. This is guaranteed by optimal substructures, and students may attempt to develop a dynamic programming function that fails to update properly by neglecting to compute the *maximum* total duration for each possible total cost.

As a concluding note, recursion can commonly lead to off-by-one errors with incorrect base cases and arms-length updating.