ICS 621: Analysis of Algorithms Fall 2021

Lecture 5

Prof. Nodari Sitchinava *Scribe: In Woo Park, Darlene Agbayani, Michael Rogers*

1 Overview

Previously, we covered the Lazy Binomial Heap operations, including an in depth analysis of EXTRACT-MIN(Q). We also discussed Fibonacci heaps, and how they can be used to reduce run-times even further, and a well-known application of the Fibonacci Heap data structure, Dijkstras Algorithm.

In this lecture, we will be introducing Dynamic Programming (DP), often performed in a step by step process starting with thinking about the brute force solution, defining a recursive solution to identify the optimal subproblems, and finally by adding the solved problems to a memo table to lookup, thus reducing wasted effort resolving problems over and over again. We will look at DP in the context of specific examples including Subset Sum, Knapsack, and Longest Increasing Subsequence. In addition, we introduce two approaches in DP: top-down and bottom-up and elaborate how they differ from each other in terms of implementation while their time complexity is usually of the same order. The red text scattered throughout the notes depict changes or additions from one algorithm to the next, similar algorithm, in order to highlight the transitioning process.

2 Dynamic Programming

> Dynamic programming is not about filling in tables. Its about smart recursion!
>
> — *Jeff Erickson*

Dynamic Programming (DP) solves problems by reusing the solutions to subproblems. In other words, it is an optimization method over plain recursion. Often times, it is more intuitive to solve a problem recursively. Therefore, we can use recursive backtracking to first identify the overlapping subproblems, and then, we solve each subproblem just once and record the solution in a table for later usage. Rather than re-computing the subproblems, we can prune the search space by accessing the solutions in the table.

We will look at DP in the context of specific examples including Subset Sum, Knapsack, and Longest Increasing Subsequence. In addition, we introduce two approaches in DP: top-down and bottom-up and elaborate how they differ from each other in terms of implementation while their time complexity is usually of the same order.

2.1 SubsetSum

In SUBSETSUM, we are given a set of numbers and a positive integer x. We need to check if there is a subset or subsequence of numbers that adds up to the given number.

Problem: Given a set of S of n integers, and a positive integer x, is there a subset of S that sums to x?

E.g.

$$S = \{ \quad 17, \quad 5, \quad 7, \quad 15, \quad 3, \quad 8\} \quad x = 16$$

Solution: Generate all possible subsets, add up the elements of each subset and check if the sum equals x. In the above example, the elements of the set that add up to $x = 16$ are identified with '1':

$$\begin{array}{ccccccccc} S = \{ & 17, & 5, & 7, & 15, & 3, & 8\} & x = 16 \\ & 0 & 1 & 0 & 0 & 1 & 1 & \end{array}$$

2.1.1 First Attempt

In the following algorithm, we check each of 2^n possible subsets. We can enumerate the possible subsets by viewing the bits, which identify which items of the set are selected or not selected, as a binary number. For example, in the above example, the subset of $\{5, 3, 8\}$ is the 19-th subset, because 010011 viewed as a binary number is equal to 19 in decimal, i.e., $010011_2 = 19_{10}$. Thus, we can use bitwise operators (AND (&) and SHIFT-LEFT (<<)) to determine whether or not the j-th bit is set (equivalent to j-th item being selected) and then check if the sum of the selected items of the set add up to x. This method clearly runs in exponential time as there are 2^n subsets to check (the first **for** loop) and the second **for** loop iterates n times, giving a total of $n \cdot 2^n$.

Algorithm 1

SUBSETSUM(S, x)

```
1  n = length(S)
2  for i = 0 to 2^n − 1                         ▷ i-th subset out of 2^n
3      sum = 0
4      for j = 0 to n − 1
5          if (i & (1 << j))                    ▷ Is the j-th bit set in i?
6              sum += S[j]                      ▷ j-th item is part of the subset
7      if (sum == x)
8          return TRUE                          ▷ If the actual subset is desired, can return i here
9  return FALSE
```

2.1.2 Recursive Solution

Lets look again at the unordered sequence S provided earlier along with Figure 1, which shows the various paths we can take to search for a solution. For each element, if we fix the first bit (most significant bit) to be 0 or 1, and allow the remaining $n - 1$ bits to vary, we have two paths from the root of the tree, either left or right. Using this method, we can then recurse.

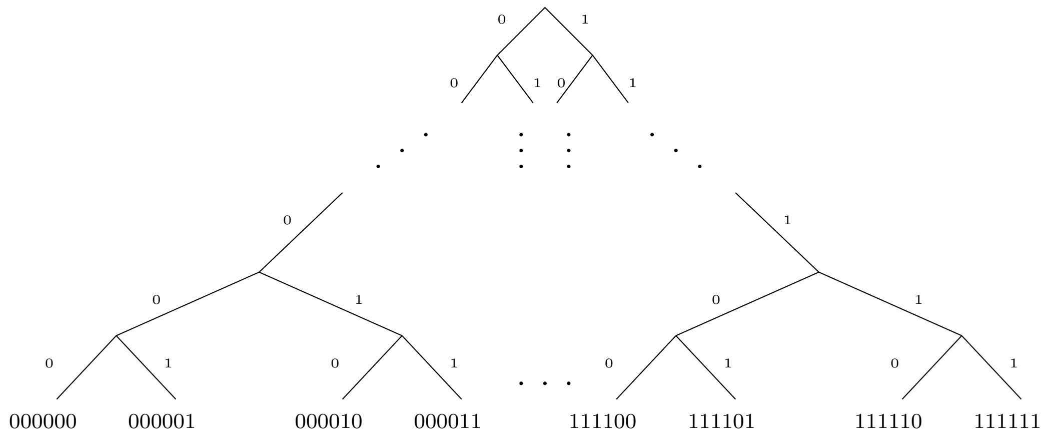

Figure 1: The search space of SubsetSum. Each leaf identifies one of the 2^n possible subsets of the input.

Let SubsetSum$(S[i:n], x)$ be the function to find whether there is a subset of $S[i:n]$ with sum equal to x. The problem can be divided into two subproblems:

- Include the element $S[i]$, and determine (recursively) if the items of $S[i+1:n]$ add up to $x - S[i]$.
- Exclude the element $S[i]$, and determine (recursively) if the items of $S[i+1:n]$ add up to x.

If either subproblem returns TRUE then the whole problem should return TRUE.

In the base case, whenever $x = 0$, the answer is TRUE, because an empty set is the subset of any set, or whenever $x < 0$, the answer is FALSE, because the input set contains only non-negative items and no subset will add up to a negative x. Thus, the following recursive formula defines the solution for this problem:

$$\textsc{SubsetSum}(S[i:n], x) = \begin{cases} \text{TRUE} & \text{if } x = 0 \\ \text{FALSE} & \text{if } x < 0 \text{ (or } i > n) \\ \textsc{SubsetSum}(S[i+1:n], x) & \\ \quad\quad \textbf{or } \textsc{SubsetSum}(S[i+1:n], x - S[i]) & \text{otherwise} \end{cases}$$

Observe that some of the parameters to the SubsetSum function do not change in the recursive calls, e.g., array S and integer n. If we define these as global variables, we do not need to pass them on during the recusive calls (see Algorithm 2). Therefore, we can simplify our understanding of what the function depends on by changing what SubsetSum returns during the recursive call, which will be useful for determining the memoization table dimensions of the dynamic programming version of the solution in the next section.

2.1.3 Dynamic Programming

Recall that Dynamic Programming is an optimization over plain recursion. In the previous recursive solution, there might be multiple calls to any subproblem SubsetSum(i, x). Therefore, if we can store (memoize) the answer to SubsetSum(i, x) during the first call to it, we can look up the answer instead recomputing it.

Algorithm 2

Initialize S and x

MAIN()

1 **return** SUBSETSUM$(1, x)$

SUBSETSUM(i, x) ▷ Returns TRUE iff there exists a subset within $S[i:n]$ that adds up to x

1 **if** $(x < 0$ **or** $i > n)$ **return** FALSE

2 **else if** $(x == 0)$ **return** TRUE

3 **else**

4 **return** SUBSETSUM$(i + 1, x)$ **or** ▷ $S[i]$ not selected

5 SUBSETSUM$(i + 1, x - S[i])$ ▷ $S[i]$ selected

The number of variables (i, x) that are passed to the recursive call will define the memoization table. Since there are two variables being passed in, the memoization table will be a 2-dimensional array. It will have an index for i and an index for x.

Therefore, we create a 2D table $SS[0..n][0..x]$ and use it to memoize the return values. The entry $SS[i][x] =$ TRUE will represent that there exists a subset of $S[i : n]$ whose sum equals to x, and FALSE otherwise.

The following is a recursive algorithm that follows the recursive structure mentioned above.

Initialize S and x

Initialize $SS[20][MAX_X]$ to UNDEFINED values

MAIN()

1 **return** SUBSETSUM$(1, x)$

SUBSETSUM(i, x) ▷ Returns TRUE iff there exists a subset within $S[i:n]$ that adds up to x

1 **if** $(x < 0$ **or** $i > n)$ **return** $SS[i][x] =$ FALSE

2 **else if** $(x == 0)$ **return** $SS[i][x] =$ TRUE

3 **else if** $(SS[i][x] \neq$ UNDEFINED$)$ **return** $SS[i][x]$

4 **else**

5 **return** $SS[i][x] =$ SUBSETSUM$(i + 1, x)$ **or** SUBSETSUM$(i + 1, x - S[i])$

In this algorithm, in the worst case, we need calculate the $SS[i][x]$ for every i and x at most once, so the worst cost is $O(xn)$.

The table stores the answer to the recursive call with parameters i and x, so if we ever need to call it again, we don't have to go through the process of re-computing. We can return whatever is stored in that entry. In order for this action to be useful, such as to save time, we should be running multiple calls on the same values i and x.

Student Question

Q: Doesn't it take time to look things up in the table? Since this problem is NP complete, does

that mean the table is guaranteed to be useful?
A: This process is a lot faster than solving an exponential time algorithm, and it is guaranteed to be useful for small values of x.

2.2 0-1 Knapsack

0-1 KNAPSACK. *Given a set S of n items, each with its own value V_i and weight W_i for all $1 \leq i \leq n$ and a maximum knapsack capacity C, compute the maximum value of the items that you can carry, without exceeding capacity C. You cannot take fractions of items.*

Example:

$$\{(V_i, W_i)\} = \{(10, 17), (5, 7), (3, 8), (9, 15)\} \qquad C = 16$$

Then, the best value is 9 in the case of (V_4, W_4).

2.2.1 First attempt

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. This problem is similar to the subset-sum problem. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the subset with the maximum value.

2.2.2 Recursive

Like the subset-sum problem, MAXV(i, C) returns the maximum value among items $S[i : n]$ with remaining knapsack capacity of C. When $i = 1$, that is the initial problem, if the first item is not included in the knapsack, then this problem becomes the problem of finding the maximum value among items $S[2 : n]$ with remaining knapsack capacity of C. When the first item is included in the knapsack, then this problem becomes the problem of finding the maximum value among items $S[2 : n]$ with remaining knapsack capacity of $C - W_1$.

To consider all subsets of items, the Knapsack problem can be divided into two subproblems:

- The item is not included in the optimal subset, maximum value obtained by $S[i + 1, n]$ and the maximum capacity C.
- The item is included in the optimal set, maximum value obtained by the value of ith item plus the maximum value obtained by $S[i + 1, n]$ and $C - W_i$.

And when the $C \leq 0$, return FALSE, because there is no capacity to store any item. When $i \geq n$, return FALSE, because that is out of the range.

If $W_i > C$, then V_i cannot be included in the optimal subset and case 1 is the only possibility. Else, the V_i can be either included in the optimal subset or not included in the optimal subset.

The following is the recursive formula for this problem:

$$\text{MAXV}(i, C) = \begin{cases} 0, & \text{if } i > n \\ 0, & \text{if } C \leq 0 \\ \text{MAXV}(i+1, C), & \text{if } W_i > C \\ \max\{\text{MAXV}(i+1, C), V_i + \text{MAXV}(i+1, C - W_i)\}, & \text{if } W_i \leq C \end{cases}$$

The following is a recursive algorithm that follows the recursive structure mentioned above.

Algorithm 3

Initialize $V[1 \mathinner{..} n], W[1 \mathinner{..} n]$ and C

```
MAIN()
1  return MAXV(1, C)

MAXV(i, C)
1  if (i > n or C ≤ 0) return 0
2  if (W[i] > C)
3        return MAXV(i + 1, C)
4  return MAX(MAXV(1 + 1, C), V[i] + MAXV(i + 1, C − W[i]))
```

2.2.3 Dynamic Programming

This is similar to the SUBSETSUM problem. Given that we have two parameters, we know we must create a 2D table. Initialize $M[1..n][1..C]$, and use it to memoize the returned values. Line 2 checks to see if the table is filled and returns the value, if defined.

The following code shows recursive algorithm that follows the recursive structure mentioned above.

Algorithm 4

Initialize $V[1..n], W[1..n]$ and C and set $M[1..n][1..C]$ to UNDEFINED.

```
MAIN()
1  return MAXV(1, C)

MAXV(i, C)
1  if (i > n) or (C ≤ 0) return 0
2  if (M[i][C] ≠ UNDEFINED) return M[i, C]
3  if (W[i] > C)
4        return M[i][C] = MAXV(i + 1, C)
5  return M[i][C] = MAX(MAXV(1 + 1, C), V[i] + MAXV(i + 1, C − W[i]))
```

2.3 Longest Increasing Subsequence

The Longest Increasing Subsequence (LIS) problem asks to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.

Problem. Given a sequence $A[1...n]$ determine the length of its longest subsequence (not necessarily contiguous) that is in increasing order.

$$\text{Sequence: } A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$$

Hint 1: LIS(i) returns the size of the longest increasing subsequence that terminates on $A[i]$.

Hint 1: When deciding whether to add $A[i]$, find the longest subsequence in $A[1 : i-1]$ that allows adding $A[i]$.

2.3.1 Recursive

The brute force solution is to compare all elements with the next element. In our case, -7 is our first element so it is added as part of our sequence. Once element 10 is called, it is checked against all next elements if 10 is the best next number in our sequence. The next element in the sequence is 9 which is less than 10, therefore, it is chosen as the next sequence. This process is repeated until we arrive at element 2. No other element except 1 is less than 2, but 1 would end our sequence so we keep 2 in this attempt. We repeat the process until we arrive at a sequence $\{-7, 2, 3, 8\}$.

Algorithm 5

```
LIS(i)
1  if (i == 1) return 1
2  best = 1                                   ▷ LIS = {A[i]}
3  for j = 1 to i − 1
4        if (A[i] > A[j])
5        current = LIS(j) + 1
6        if (best < current)
7              best = current;
8  return best

MAIN()
1  best = 0
2  for i = 1 to n
3        current = LIS(i)
4        if (best < current)
5              best = current
```

2.3.2 Dynamic Programming

The length of LIS for $A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$ is 4 and LIS $= \{-7, 2, 3, 8\}$. We employ DP to solve this problem in polynomial time.

Let's consider that LIS(i) returns the size of the longest increasing subsequence of the array $A[1:i]$. Then, we can write LIS(i) recursively by deciding whether to add $A[i]$ to the solution of LIS($i-1$) or not. Since we don't know if including/excluding $A[i]$ will block LIS to have longer subsequence later or not, we need to modify this recursion in a little smarter way. To deal with that, we always compare $A[i]$ with all previous $A[j]$ where $j < i$ since LIS(i) must be the longest increasing subsequence that ends at i, i.e.:

$$\text{LIS}(i) = \begin{cases} 1 & \text{if} \quad i = 1 \\ \max \begin{cases} 1 & \\ \text{LIS}(1) + 1 & \text{if} \quad A[i] > A[1] \\ \text{LIS}(2) + 1 & \text{if} \quad A[i] > A[2] \\ \text{LIS}(3) + 1 & \text{if} \quad A[i] > A[3] \\ \dots & \quad \dots \\ \text{LIS}(i-1) + 1 & \text{if} \quad A[i] > A[i-1] \end{cases} & \text{if} \quad i > 1 \end{cases}$$

The difference between the brute force and the dynamic programming solution is that in dynamic programming, we will use a 1 dimensional array to store the elements, and **before** returning, we will store it in the array. This process is important because if it's in the array, it has been solved before, therefore we can simply return the element. If not, we would have to recalculate every time.

Algorithm 6

```
LIS(i)
1  if (L[i] != UNDEFINED return L[i]
2  if (i == 1) return L[i] = 1
3  best = 1                                   ▷ LIS = {A[i]}
4  for j = 1 to i − 1
5       if (A[i] > A[j])
6       current = LIS(j) + 1
7       if (best < current)
8            best = current;
9  return L[i] = best

MAIN()
1  best = 0
2  for i = 1 to n
3       current = LIS(i)
4       if (best < current)
5            best = current
```

The complexity to find the Longest Increasing Subsequence depends on nested loop and is of the

order $O(n^2)$. In other words, the entry LIS(i) takes $O(i)$ time to compute which gives $O(1) + \ldots + O(n) = O(n^2)$ overall complexity.

3 Bottom-up Dynamic Programming

The bottom-up approach of DP consists of first looking at the smaller subproblems, and then solving the larger subproblems using the solution to the smaller subproblems. We can think of that as a table filling algorithm where we start by filling out lower cells and then going up. For example, if we need to calculate fifth Fibonacci number, we calculate first, then second then third all the way up to the fifth number. This approach does not use recursion but it consumes more space, since we will iteratively compute all subproblems until we reach the main problem.

There are some pros and cons to this approach. For example, this approach is not always as intuitive and requires extra work if the recursion is already defined. However, sometimes top-down recursion is even harder to define. As mentioned in the previous paragraph, another con of the bottom-up approach is we must fill up the entire memo table for it to work where as the recursive approach may exit early. Finally, the bottom-down approach is typically a bit faster due to no recursion and this also makes it easier to analyze the running times.

3.1 SubsetSum

For example in subset sum problem, we start by using only the first element. We can either use it or not, so the possible sums are $\{0, A[1]\}$. Now we keep both of these, and also try adding $A[2]$ to each one. We get $\{0, A[1], A[2], A[1] + A[2]\}$. Then we repeat until all of the elements are included. Figure 2 illustrates how bottom-up approach is like filling out a table where at each step we need to calculate the applicable underneath cells.

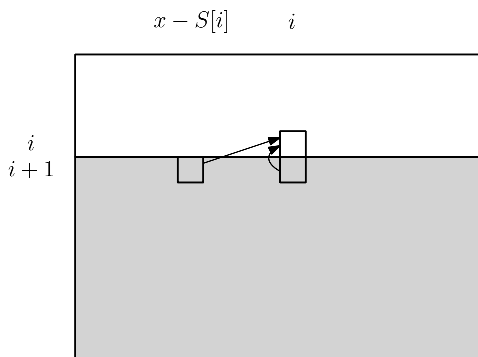

Figure 2: Tabular representation of bottom-up approach for subset sum problem.

Unlike bottom-up approach, in top-down approach we try to include the first element in the subset, and recursively see if any combination of the rest of elements add up to the rest of the sum. If we don't find a solution, leave out the first element and continue. In other words, we first break the problem into subproblems and then calculate and store values which is the opposite of bottom-up approach where we solve smaller subproblems first, then solve larger subproblems from them.

The pseudocode for implementation is as follows:

Algorithm 7

initialize S and x
int $S[20] = \{\dots\}$; int $x = \dots$
int $SS[20][MAX_X]$; memset(SS, UNDEFINED, sizeof(SS));

```
SubsetSum(i, x)        ▷ Returns TRUE iff there exists a subset within S[i : n] that adds up to x
1  if (x < 0 or i > n) return SS[i][x] = FALSE;
2  else if (x == 0) return SS[i][x] = TRUE;
3  else if (SS[i][x] != UNDEFINED) return SS[i][x];
4  else
5      return SS[i][x] = SubsetSum(i + 1, x) or
6                        SubsetSum(i + 1, x − S[i])
MAIN()
1  for (i = 19; i ≥ 0; i + +)
2      for (j = 0; j < MAX_X; j + +)
3          SS[i, j] = SS[i + 1][j] or SS[i + 1][j − S[i]]
4  return SS[[0][x]
```

The time complexity of the above pseudocode is dependent on nested loop which is a function of $|S|$ as the size of the set and x as the desired summation. Therefore, we have $O(|S| \cdot x)$ as the running time for the worst-case which is much better than exponential in the case without DP.

3.2 0-1 Knapsack

Recall the 0-1 Knapsack problem from earlier. *Given a set S of n items, each with its own value V_i and weight W_i for all $1 \leq i \leq n$ and a maximum knapsack capacity C, compute the maximum value of the items that you can carry. You cannot take fractions of items.*

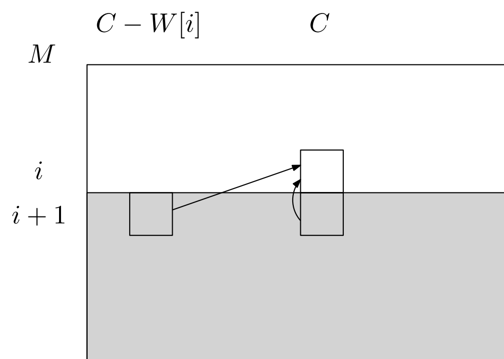

Figure 3: Tabular representation of bottom-up approach for the knapsack.

Algorithm 8 ⟨ This is not a bottom-up solution! ⟩

Initialize $V[1..n], W[1..n]$ and C and set $M[1..n][1..C]$ to UNDEFINED.

```
MAIN()
1  return MAXV(1, C)

MAXV(i, C)
1  if (i > n) or (C ≤ 0) return 0
2  if (M[i][C] ≠ UNDEFINED) return M[i, C]
3  if (W[i] > C)
4       return M[i][C] = MAXV(i + 1, C)
5  return M[i][C] = MAX(MAXV(1 + 1, C), V[i] + MAXV(i + 1, C − W[i]))
```

3.3 Football Game

Problem:

In a football game, after every scoring play, the cheerleaders do as many jumps as the total number of points on the scoreboard. For example, if the team first scored a touchdown (7 pts), then a field goal (3 pts), then a safety (2 pts), the cheerleaders did $7 + 10 + 12 = 29$ total jumps.
Given the number n of total jumps, compute the largest possible number of points scored in the game?

You may assume the possible points are given as a set of S of m positive integers, with the largest value at most 20. For example, in regular football rules, $m = 5$ and $S = \{2, 3, 6, 7, 8\}$.

Solution:

Let $P(n)$ define the maximum number of points for n total jumps. What is the recursive definition of $P(n)$?

Hint: Try all possible values for the last score and take the maximum.

$$P(n) = \max \begin{cases} P(n - S[0]) + S[0] \\ P(n - S[1]) + S[1] \\ P(n - S[2]) + S[2] \\ \dots \\ P(n - S[m-1]) + S[m-1] \end{cases}$$

3.3.1 First attempt

$P(n)$ is the number of points we scored for n jumps and the possibility for the last score is $S[0], S[1], S[2], ..., S[m-1]$. Therefore, if we subtract $S[i]$ values from n, that will be the total number of jumps plus the score that we just scored.

However, this analysis is incomplete. We're not only adding what was scored in the current position, but we also need to add what we added previously. Take for example the previous example:

$$\begin{aligned}&\text{Points scored} : 7, 3, 2\\&\text{Total Jumps} = (7 + 10) + 12 = 29\\&P(29) = P(29 - 2) + 2? \quad \leftarrow P(29) = P(17) + 2\end{aligned}$$

If we scored, 7, 3, and 2, the total jumps will be 29, but in the last play only 2 points were scored. We're not looking for number of points for $P(29 - 2)$ because we want the the number of jumps before the previous play, $P(29 - 12) = P(17)$. $P(17)$ is P(29 - # of jumps of the last play) where the number of jumps **does not** equal the number of points being scored. This process is defined as:

$$\begin{aligned}&\text{If given the current score } x:\\&P(29) = P(29 - x) + 2\\&P(n) = P(n - x) + S[i]\end{aligned}$$

3.3.2 Recursive

Now that our process is defined, we have to find what the value of x is. We know that x is the number of points, given the number of jumps. Therefore we can define our analysis slightly differently:

$$P(n, x) = \max \begin{cases} P(n - x, x - S[0]) + S[0] \\ P(n - x, x - S[1]) + S[1] \\ P(n - x, x - S[2]) + S[2] \\ \dots \\ P(n - x, x - S[m - 1]) + S[m - 1] \end{cases}$$

In $P(n, x)$, n is the total number of jumps, and x is current score, which is the total score plus before adding the new points. We subtract n minus the current score, (i.e. 12, from the previous example) and the previous score will be x minus the previous score ($S[0]$). With this definition, how do we determine what x is? Well, x turns out to be what we're trying to compute, which is P, the maximum value of the score the team scored. It seems that it has become a circular definition:

$$x = P(n, x) = \max \begin{cases} P(n - x, x - S[0]) + S[0] \\ P(n - x, x - S[1]) + S[1] \\ P(n - x, x - S[2]) + S[2] \\ \dots \\ P(n - x, x - S[m - 1]) + S[m - 1] \end{cases}$$

P represents the maximum score that the team could have scored at the end of the game when the cheerleaders have jumped n times. For that, we need to know what the current score is, which is x, which is P. That way we can subtract the current score from n for the recursive definition, and

so we can subtract it from the current play score. This recursive definition is a circular definition because neither of n or x get smaller and we need to find x, which is $P(n, x)$, which is $P(n, x)$, which $P(n, x)$, etc. Instead, in the next section we solve this problem using the **Bottom-up** solution.

3.3.3 Bottom-up

In order to build the bottom-up solution, we need only use the recursive backtracking solution from earlier and rewrite it to be iterative so that we are solving the problems and looking up the solutions as we go. Let M be the memoization table, the row index j represents the number of jumps, and the column index x represents the total number of points. Then the entry $M[j, x]$ indicates if x points are possible given j jumps: TRUE, if x points is possible, and FALSE, if x points is not possible. The result depends on the row $j - x$, and the columns $x - S[0], x - S[1], \ldots, x - S[m-1]$.

$$M[j, x] = OR \begin{cases} M[j - x, x - S[0] \\ M[j - x, x - S[1] \\ M[j - x, x - S[2] \\ \ldots \\ M[j - x, x - S[m-1]] \end{cases}$$

To answer the question of maximum number of points given n jumps, we return the largest x for which the entry $M[n, x]$ is set to TRUE (the rightmost entry in the last row that is set to TRUE.

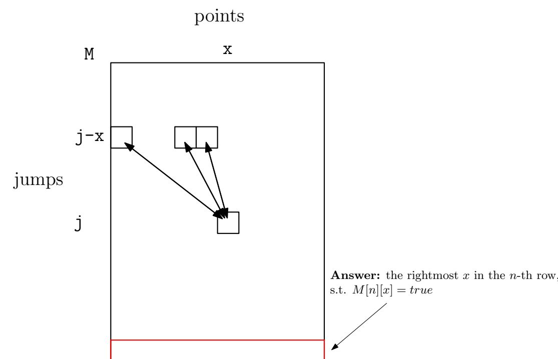

Figure 4: Tabular representation of bottom-up approach for the football problem.

For example, to build the table, we start at $j = 0, x - 0$, which is row $j - x$, or row 0. In fact, it is not possible to have a scoreboard with only 1 point in football so we can even skip to start from 2 points, so the number of jumps will also be 2. We can initialize the table with TRUE values up to n where the $M[S_i][S_i]$ is provided, meaning we know it's possible $n = 2$ can result from a 2 point play, initially. Likewise, any of the given scores in S are possible up to $S_i = n$. This is the starting process to fill out the table. But, we do not need to think about much else because we let the recursive process do the work instead.

Then to find the answer, we just iterate backwards from the n-th position, and the first element marked TRUE will be the answer.

Algorithm 9

```
FOOTBALL(S[1..m], n)                    ▷ Returns maximum possible points, for n total jumps
 1  Initialize every entry of a 2D array M[n][n] to FALSE
 2  for i = 1 to m                                                          ▷ Base cases
 3      M[S[i]][S[i]] = TRUE
 4  for j = 2 to n
 5      for x = 1 to j − 1
 6          for i = 1 to m
 7              if (x − S[i] > 0 and M[j − x, x − S[i]] = TRUE)
 8                  M[j][x] = TRUE
 9  for x = n downto 1
10      if M[n][x] = TRUE
11          return x
12  return 0
```