Dynamic Programming Example: Winning a board game

The problem: A game board is represented by two arrays $cost[1..n]$ and $jump[1..n]$. The value $cost[i]$ represents the cost of visiting position i. The value $jump[i]$ represents the maximal number of positions one is allowed to jump to the right. For example, for $n = 6$:

17	2	100	87	33	14
1	2	3	1	1	1

The object of the game is to jump from the first to the last position in the row. The cost of a game is the sum of the costs of the visited fields. The goal is to compute the **cheapest** game.

Step 0: We focus on the optimal value The optimal solution has two parts to it: its cost, and the set of jumps. We will first focus on how to find the cheapest cost. Then, once we have an algorithm for that, we will think how we can extend it so that it computes not only the optimal cost, but also the jumps corresponding to the optimal cost. This will be our general strategy for all DP problems.

1 Step 1: Recursive formulation

1.1 Optimal substructure

Does the problem have optimal substructure? Let's see. Consider someone tells us what is the cheapest game from 1 to the end; let's call this optimal game O; let's say for e.g. that the cost of O is 133. Let's imagine that the first move in O is to jump to location 2. So now we are at position 2 on the board, and we incurred the cost 17 of being in position 1. Claim: It must be that the remaining part of O (in our case, $133 - 17 = 116$) must be the optimal game from 2.

Why is this true? We'll use what's called a *contradiction argument*: Let's assume that the claim was not true, namely let's assume that: (A) the remaining part of O is not the optimal game from position 2. This means that the optimal game from position 2 is cheaper than 116. Then we could take the optimal game from 2 (which is cheaper than 116), add to it $cost[1]$, and we get overall a solution to jump from 1 to the end that wiill be cheaper than $116 + 17$. So what, you might ask? Well, that's not possible, because we said that O is the optimal game. So it;s not possible to get a game better than O. Therefore our assumption (A) lead to a logical imopssibiliy. Therefore (A) cannot be right.

The argument above can be extended to show that wherever the first jump is, the remaining part of the optimal solution must be the optimal game from that position. This shows that the optimal solution contains within it optimal solutions to subproblems.

1.2   Recursive solution

How do we actually solve the problem?   So now we know the problem has optimal substructure, so we know we can approach it recursively. But...how?

In fact, optimal substructure gives us a constructive way to approach the problem, because it amounts to the following:

> *If someone told us the first jump, all we need to do is (jump and) recurse from there.*

We'll need to come up with a notation that will allow us to solve the problem recursively.

Our notation.   We need to decide how to parametrize the function we want to compute, that is, what parameters it should have and what they represent. We know we want to compute a function that returns the cheapest game, but what is the parameter in the game? As we jump we end up at arbitrary positions on the board and want to find the optimal game from there. So the parameter is the position we are at on the board.

> Notation: Let $Cheapest(i)$ represent the cost of the cheapest game starting at position i.

Put differently, when called with argument i, $Cheapest(i)$ computes and returns the cost of the cheapest game starting at position i.

To find best game from 1, we will call it with $i = 1$, $Cheapest(1)$.

A recursive solution for $Cheapest(i)$**:**   Clearly, if we know the first move from i, we could just take it and recurse from there. But, we don't. So, what do we do? We explore all moves, and pick the one that gives cheapest game.

- if $Jump[i] == 1$: we don't have any choice. We jump to the next position $i + 1$ and recurse from $i + 1$.

- if $Jump[i] == 2$: we have two choices: we either jump to $i + 1$ and find the optimal solution from there, or jump to $i + 2$ and find the optimal solution from there. Since we don't know which one is the best one, we try both, and choose the cheapest.

- if $Jump[i] == 3$: we have three choices: we either jump to $i+1$ and find the optimal solution from there, or jump to $i + 2$ and find the optimal solution from there, or jump to $i + 3$ and find the optimal solution from there. Since we don't know which one is the best one, we try all three, and choose the cheapest.

The following procedure implements this.

```
//compute and return the value of the cheapest game from i
Cheapest(i)

    IF i>n:  return 0

    ELSE

        IF Jump[i]==1 THEN
            x=Cost[i]+Cheapest(i+1)
            answer = x
        IF Jump[i]==2 THEN
            x=Cost[i]+Cheapest(i+1)
            y=Cost[i]+Cheapest(i+2)
            answer = min(x,y)
        IF Jump[i]==3 THEN
            x=Cost[i]+Cheapest(i+1)
            y=Cost[i]+Cheapest(i+2)
            z=Cost[i]+Cheapest(i+3)
            answer = min(x,y,z)
        return answer

END Cheapest
```

Correct: Why is this correct? At every step, the algorithm is exploring *all* possibilities and chooses the best one.

Analysis: What is the asymptotic running time of the procedure? Since this is a recursive algorithm, we express the running time with a recurrence. Let $T(n)$ be the worst-case time to find the cheapest game (starting at position 1) on a board of size n. Then,

$$T(n) = T(n-1) + T(n-2) + T(n-3) + \Theta(1)$$

Solving this recurrence may be tricky, but this is one of the situations where we don't care. All we want a lower bound for $T(n)$:

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + T(n-3) + \Theta(1) \\
&\geq 3\,T(n-3) \\
&= 3^2\,T(n-6) \\
&= \ldots \\
&= 3^k\,T(n-3k) \\
&\geq 3^{n/3} = \Omega(3^{n/3}).
\end{aligned}
$$

Note that we don't compute carefully the $\Theta()$ of the running time, but we know it's at least exponential. When running times are exponential, we usually don't care to find a tight bounds.

Exercise: To get some intuition, draw the recurrence tree for a board of size 3 with the jumps all equal to 3. You can also try to draw the recursion tree for the board in this section.

Comments   Let's try and understand why the running time of Cheapest(1) is exponential:

Question: How many different sub-problems Cheapest(i) can there be?
Answer: At most n, one for each value of i.

Question: Aside from recursion, how much work does Cheapest(i) do?
Answer: $O(1)$

Question: Then why is the algorithm exponential?
Answer: It's because of **overlapping calls** to same sub-problems, causing them to be solved several times.

2   Step 2: Recursive (top-down) DP with memo-ization

To avoid redundant calls to the same problem, we create a table $T[0..n]$ and will use $T[i]$ to store the result of $Cheapest(i)$.

Note that a subproblem $Cheapest(i)$ is parametrized by an index i, so we can map between a subproblem and its location in the table.

We initialize the table with a value that can be distinguished from the value returned by $Cheapest(i)$; in this case we can use -1.

The recursive algorithm can be modified to use the table.

```
//compute and return the value of the cheapest game from i
Cheapest(i)

     IF i>n:  return 0

     //if it's been already calculated, retrieve it

     ELSE IF T[i] != -1:  return T[i]

     ELSE

          IF Jump[i]==1 THEN
              x=Cost[i]+Cheapest(i+1)
              answer = x
          IF Jump[i]==2 THEN
              x=Cost[i]+Cheapest(i+1)
              y=Cost[i]+Cheapest(i+2)
              answer = min(x,y)
          IF Jump[i]==3 THEN
              x=Cost[i]+Cheapest(i+1)
              y=Cost[i]+Cheapest(i+2)
              z=Cost[i]+Cheapest(i+3)
              answer = min(x,y,z)
          T[i] = answer // store the answer
          return answer

END Cheapest
```

Analysis: Let's visualize the recursion: To find the solution to our problem we call Cheapest(1). This will recursively call Cheapest(2), Cheapest(3), which in turn will call Cheapest(i) for $i > 1$, until i will reach the end of the table. At that point recursion will hit base case and will return a value for Cheapest(n), which will be stored in $T[n]$. The first value stored in the table will be $T[n]$, followed by $T[n-1]$, and so on. We can see that the recursion moves left to right, and the table is filled right to left.

Calling Cheapest(1) triggers a cascade of recursive calls that will fill the entire table. The key of the analysis is to separate between recursion and the actual computation.

Cost of recursion: Each Cheapest(i) is solved exactly once, because afterwards it is stored in $T[i]$ and is retrieved when needed. There are n different subproblems, and each is called recursively exactly once. Thefore the cost of recursion is $O(n)$.

Cost of computation: We know that calling Cheapest(1) will fill the entire table. Let's ignore recursion, and imagine that when calling Cheapest(i), $T[i + 1]$, $T[i + 2]$ and $T[i + 3]$ have been computed and are stored in the table. Cheapest(i) will retrieve them in $O(1)$ time and compute the miminum of x, y, z in $O(1)$ time. Ignoring the cost of recursion, Cheapest(i) can be computed

in $O(1)$ time.

Therefore the cost of Cheapest(1) is: $O(n) + O(n) \cdot O(1) = O(n)$.

3   Step 3: Iterative (bottom-up) DP

It is actually possible to eliminate recursion completely and compute the cheapest game iteratively, but iterating through the table in the order in which recursion fills it (in this case: right to left). From an asymptotic point of view the recursiove and non-recursive solutions are equivalent; in practice recursion adds an overheadd so an iterative solution is preferred.

```
//returns the cheapest game from 1
  cheapest_iterative()

    create table T[1..n+3] and set T[n+1]=T[n+2]=T[n+3]= 0

    FOR i = n down to 1:
        IF Jump[i]==1 THEN T[i] = cost[i]+ T[i+1]
        IF Jump[i]==2 THEN T[i] = cost[i]+ min(T[i+1], T[i+2])
        IF Jump[i]==3 THEN T[i] = cost[i]+ min(T[i+1],T[i+2], T[i+3])

    //the value in T[1] is the cost of the cheapest game

    return T[1]
```

Analysis: $O(n)$
The iterative DP solution is harder to come up with, but a lot simpler to analyze.

Step 4: From finding optimal cost to finding the optimal game

Now we have an algorithm to compute the cost of the cheapest game. How can we extend it to compute the set of jumps that correspond to this optimal game?

There are two ways to go about this.

- One way is to traverse the table of optimal sub-solutions T[] (computed with either top-down or bottom-up DP) and piece together the optimal jumps one at a time reverting the same logic that we used to compute T[i].

- The other way is to extend the Cheapest(i) to keep a move[] table in additional to the table T[]; where move[i] will store the optimal jump at index i. When cheapest(i) decides what the optimal jump is at step i is, it stores that jump in move[i] (for e.g. if the optimal jump at i was 3, then it would store that move[i]=3). We start at index 1 and traverse the move[] and T[] tables and piece the optimal solution one jump at a time.