Board Cutting

Given a board that is n units long, we wish to cut it in the most profitable way. For each possible length, we know the selling price - these can be any positive values. For example, the table of lengths and selling prices might be (this is not the same example as I used in class):

length	1	2	3	4	5	6	7
selling price	2	6	7	10	13	15	17

We can assume that the selling price for any length greater than the largest listed length is 0 - so in the example shown here, the selling price for pieces of length 8 or more is 0.

We will assume that there are **k** lengths with non-zero selling prices. For simplicity we will assume these lengths are 1, 2, ..., k, but they can actually be any lengths.

We only make "whole unit" cuts, so there are n-1 cut-points in the board – each cut is an element of the set $\{1, 2, \ldots n-1\}$. Thus there are 2^{n-1} different ways to cut the board. Many of these result in identical collections of pieces - for example, cutting 1 unit off the left end has exactly the same result as cutting 1 unit off the right end - but even with duplicates removed the number of possible ways to cut the board is exponentially large.

Why not use a Greedy approach?

Suppose we always cut off the most valuable piece we can (ie. cut off pieces of length 7 until we can't do that any more, then stop). If we start with n = 10 this results in a piece of length 7 and a piece of length 3, with total value of 24. But we can easily beat that: a piece of length 6 and a piece of length 4 has a total value of 25.

Suppose we compute the value/length ratio, and cut as many pieces as possible of the length with the highest ratio? This approach will fail for this example:

length	1	2	3	4	5
selling price	2	3	8	12	14.9

You should verify this! (Try a board of length 10) If you feel like I'm cheating by having 14.9 as a selling price, just multiply all the prices by 10.

As with the other problems, we need to find a recurrence relation that describes the optimal solution in terms of solutions to smaller problems.

Let Max_Val(n) be the maximum profit we can get from a board of length n.

We have two ways of approaching this.

The left-most cut approach

If we consider the set of cuts that comprise the optimal solution, regardless of the order in which we make them, one of the cuts must be closest to the left end of the board. This piece does not get divided any more, so its value in the solution is just its selling price. The other side of course may have many more cuts, but it is a shorter board than the original and we can use a recurrence to refer to its optimal solution.

The base case is simple: Max_Value(0) = 0 but we also need to include this:
Max_Value(x) = $-\infty \quad \forall\ x < 0$ - why?

Recall that we are assuming we know the selling price for each length in {1, 2, ..., k}

For any $n \geqslant 1$, we have

Max_Value(n) = max {
Selling_Price(1) + Max_Value($n-1$),
Selling_Price(2) + Max_Value($n-2$),
Selling_Price(3) + Max_Value($n-3$),
...
Selling_Price(k) + Max_Value($n-k$)
}

Now we can see why we needed to specify the Max_Value for negative board lengths: $n-k$ might be negative.

Or we can look at it this way ...

The first cut approach

The first cut we make (if any) divides the board into two parts. We need the optimum solution for each of the two parts - adding them together gives us our total profit. We don't know where to make the first cut but we can list the possibilities.

The base case for our recurrence is the same: Max_Val(0) = 0

For any n > 1, we have

Max_Val(n) = max { Selling_Price(n), # value of the board without any cuts
Max_Value(1) + Max_Value($n-1$),
Max_Value(2) + Max_Value($n-2$),
Max_Value(3) + Max_Value($n-3$),
...
Max_Value($\lfloor \frac{n}{2} \rfloor$) + Max_Value($\lceil \frac{n}{2} \rceil$)
}

(The last line uses floor and ceiling because n might be odd.)

As with the path problem and the coin problem, each result depends only on smaller results – and once we compute Max_Value(i) for the first time, we can simply look it up when we need it again. We can store the values in a 1-dimensional array and access each one in constant time.

As with the path problem, in practice we solve all the subproblems from smallest up to largest so that we always have all the information we need when we need it.

There is a significant difference between this problem and the "shortest path" problem : in the path problem, each partial result depended on either 1 or 2 previous results. In this problem, each partial result depends on a different number of previous results. However, the number of required previous results never exceeds n, so the complexity remains polynomial.

These two recurrences are both correct, but they have different orders of complexity! Make sure you know which is faster. You may wonder why I bothered showing you both of them. It's worth it because it provides us with a nice way of relating Dynamic Programming to the other paradigms we have studied. The Leftmost Cut approach is very much like a Greedy Algorithm. We make a cut and then move on rightward – the difference is that we don't know where the leftmost cut should be so we try out all the possibilities. The First Cut approach is very much like a Divide and Conquer Algorithm. We cut the board "in the middle" and then work on the left side and the right side independently – of course in this problem we don't know exactly where "the middle" is so we try out all the possibilities.

BONUS MATERIAL! This is something I just thought about while writing up these notes on March 2, 2019 – no other CISC-365 class in the history of the universe has seen this.

Even though we don't know exactly where "the middle" is, we know that our optimal solution will not include any board pieces of length > k. So if we consider a (k+1)-wide segment of the board around the $\lfloor \frac{n}{2} \rfloor$ point, there must be at least one cut within this segment. This means we can limit our recurrence relation for the first cut to just the k-1 possible locations for this "first cut in the middle" - and then the same argument applies on the left side and the right side. This actually reduces the complexity of the First Cut approach to O(n) – the same as the Leftmost Cut approach. I will leave the exact formulation of the recurrence relation as an exercise.

Dynamic Programming Paradigm

Constructing a dynamic programming solution to an optimization problem always involves the same steps. I'll describe them in a sequence, but in practice they usually proceed in parallel.

1. Find a recurrence relation that defines an optimal solution in terms of optimal solutions to subproblems, and establish the base case(s).

2. Determine one or more parameters that define each subproblem. In the path problem, these were the row and column of the vertex. In the other problems, they were the size of the board, and the target money value.

3. Define a table to hold the values of optimal solutions for the subproblems. If each subproblem is defined by 2 parameters, this will typically be a two-dimensional table. If the subproblems are defined by a single parameter, the table is typically one-dimensional.

4. Determine the order in which the elements of the table will be filled in. There may be alternatives - the essential requirement is that when we want to fill in a particular element of the table, the subproblems on which it depends have already been solved and their table elements have been filled in.

5. Determine how we will use the completed table to extract the details of the optimal solution. There are two popular methods:

- store information in the table that indicates which particular subproblems provide the optimal solutions to larger problems, so that when we get to the optimal solution for the original problem, we can easily trace back the steps that get us there
- work backwards from the final entry in the table, re-examining the different subproblems that might have contributed to it, and determine which subproblem(s) were actually chosen ... and then work backwards from there in the same way.

Principle of Optimality

It is time to talk about identifying which problems can be solved using dynamic programming.

The problems just discussed have something in common. In each case, we can look at the optimal solution to the problem as a sequence of decisions. After the first decision (whatever it is) the problem reduces to a smaller problem (or problems), **and the solution to the smaller problems must be the optimal solution to those problems**. That is, the solutions to subproblems that are embedded in the optimal solution must themselves be optimal.

Consider the path problem. We don't know which vertices are part of the optimal solution, but if (a,b) and (c,d) are both part of the optimal solution, then the part of the optimal solution that connects these two vertices must be the best possible way of getting from one of these vertices to the other. We can be sure of this because of the following reasoning:

- Consider the optimal solution P. By definition of the term "optimal", there is no solution better than P.
- Let (a,b) and (c,d) be any two vertices that lie on the optimal solution, and let P* be the part of P that joins (a,b) to (c,d)
- Note that every path joining (a,b) to (c,d) must lie completely within the part of the grid with (a,b) at its upper left corner and (c,d) at its lower right corner.
- Suppose there is a better path joining (a,b) to (c,d) - call it P**. This means that P** has a lower total weight than P*
- Note that because P** is completely contained in the rectangle we just defined, it cannot overlap with any part of P that is before (a,b) or after (c,d)
- Now construct a path by removing P* from P, and replacing it with P** - call this new path P'. Clearly P' is a solution to the problem because it is a path that joins (0,0) to (n,m), and equally clearly P' has a lower total cost than P because we have replaced part of P with a cheaper alternative.
- But that is a contradiction, because there can be no solution better than P.
- Therefore our supposition that P** is better than P* must be false.
- Therefore P* is the optimal path from (a,b) to (c,d)

You should work through the equivalent arguments for the other problems as well.

This property is called the **Principle of Optimality**. **A problem satisfies the P of O if every optimal solution contains only optimal solutions to embedded subproblems.**

Dynamic Programming will not be effective for a problem that fails to satisfy the P of O

... because dynamic programming is based on the idea of combining optimal solutions to subproblems to solve the original problem.

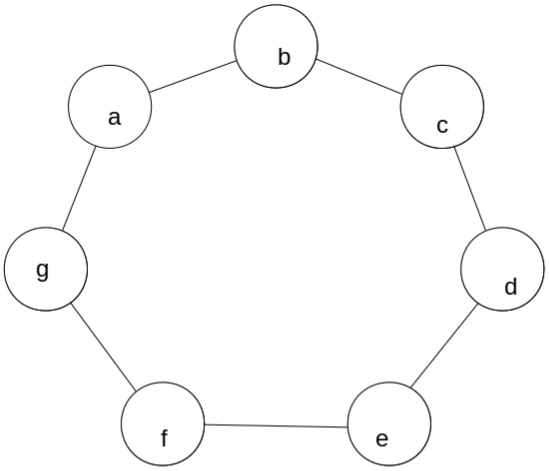
Are there problems that don't satisfy the P of O? Indeed there are. Here are two examples:

1. Graph colouring

Consider the problem of properly colouring the vertices of a 5-cycle. This requires 3 colours, but any proper subgraph of the graph requires only 2 colours - thus the optimal solution contains at least one embedded subproblem solution which is not an optimal solution for that subproblem.

2. Longest path between two vertices in a graph

Consider this graph:



The optimal solution to the problem "find the longest path from vertex a to vertex b" is a-g-f-e-d-c-b . This contains the path f-e-d ... which is NOT the optimal solution to the problem "find the longest path from vertex f to vertex d" (In fact, none of the subpaths of a-g-f-e-d-c-b is an optimal solution to the corresponding subproblem.)

Thus this problem does not satisfy the principle of optimality. However, there are millions of problems that ***do*** satisfy the P of O, and we will look at a few more.