Dynamic Programming

Henry Z. Lo

June 17, 2014

1 Motivating Example

1.1 Fibonacci numbers

Problem 1. *Given a number n, produce the n^{th} Fibonacci number.*

Recall that a Fibonacci number is the sum of the previous two Fibonacci numbers. The first two Fibonacci numbers are 1:

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ f(n-2) + f(n-1) & \text{otherwise} \end{cases}$$

1.1.1 Brute force solution

The obvious algorithm would be to follow this recursive formula, yielding the computation tree seen in Figure 1. For each number from 1 to n, the algorithm spawns off two recursive calls - thus, time complexity is $O(2^n)$.

1.1.2 Memoization

Note in the computation of this algorithm, there are *overlapping subproblems* - for example, $f(2)$ is computed 3 times, and $f(3)$ is computed twice. We can save entire subtrees if we simply store (memoize) the result of $f(2)$ and $f(3)$, and then simply try a lookup next time $f(2)$ or $f(3)$ is needed.

See Figure 2 for a visual.

How fast is this new algorithm? Well, notice that each Fibonacci number is only calculated once. Thus, this algorithm requires roughly $O(n)$ computations.

1.1.3 Dynamic programming approach

Another method to avoid computing Fibonacci numbers repeatedly is to compute them from the bottom up. For this problem, we can simply compute the Fibonacci numbers up from 2 to n (see Figure 4).

$f(2)$ only involves one computation, and once we have $f(2)$, $f(3)$ can be calculated in one step. Thus, continuing up to n requires $O(n)$ computations.

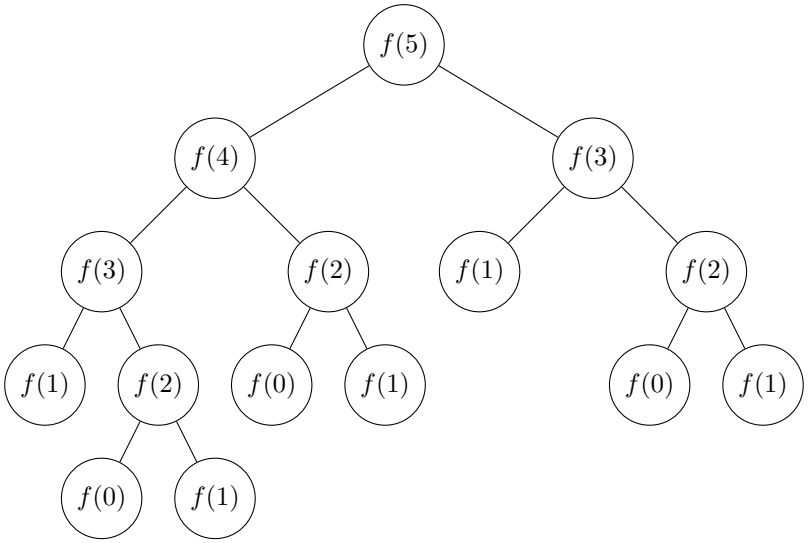

Figure 1: Computation tree for calculating the 5th Fibonacci number.

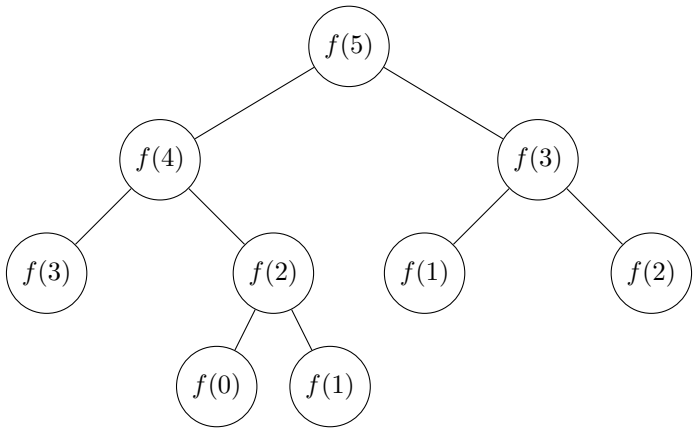

Figure 2: Fibonacci computation tree, with memoization.

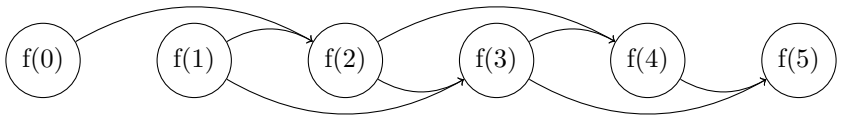

Figure 3: Fibonacci computations, with dynamic programming.

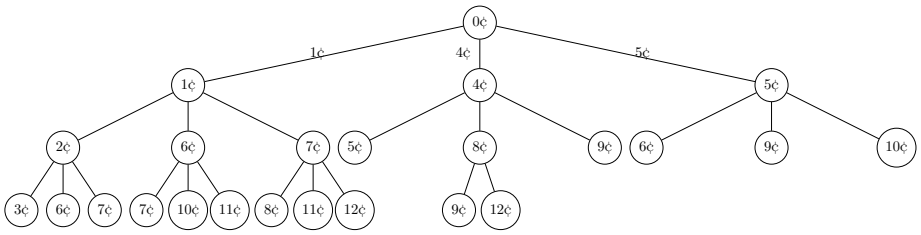

Figure 4: Bottom-up computation tree for the change-making problem.

2 Dynamic Programming

2.1 Change making revisited

Problem 2. *Given a value n, and a set of coins $c_1, ..., c_m$, find the minimal set of coins summing up to n.*

Though the greedy method worked for the U.S. and probably most modern currencies, it does not work in general for all coinages. Let's consider a currency with $1¢, 4¢, 5¢$, and a value of $12¢$.

We use this example to demonstrate dynamic programming, which can get the correct answer.

2.2 Brute force search

Dynamic programming works on programs where you need to calculate every possible option sequentially. This leads to computation paths which can be modeled in a tree. Finding a solution is equivalent to performing a breadth-first search in this tree.

For example, the computation tree in Figure 1 shows every computation path that can be taken. For the change making program, Figure 4 shows a partial computation tree – it would be complete if we expand all $7¢$ and $6¢$ nodes.

Where the same value appears in multiple nodes in the tree, this means that multiple computation paths lead to the same computation state, i.e. there are *overlapping subproblems*. In Figure 4, there are three ways to get to $7¢$: using $1¢$ and $4¢$, $4¢$ then $1¢$, or $5¢$. But how we got to $7¢$ has no bearing on how we select coins in the future, so we should only regard the optimal path to $7¢$, which is $5¢$.

2.3 Problems which can be solved

In other words, when we have overlapping subproblems, we can disregard those paths which are not optimal.

Thus, dynamic programming requires two properties of problems:

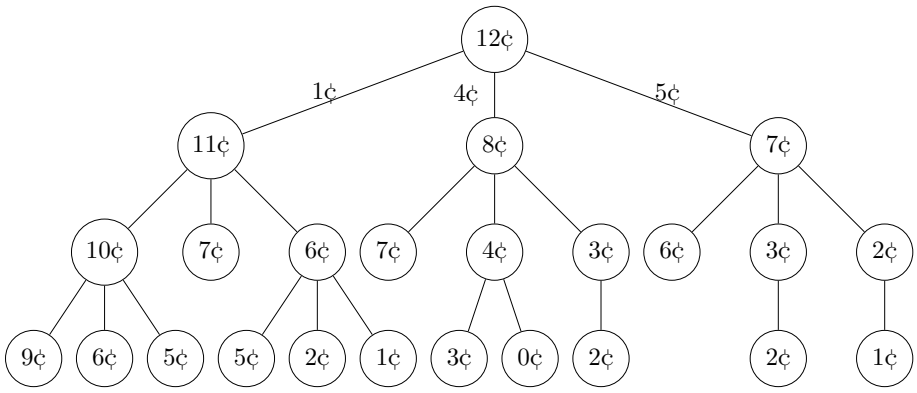

Figure 5: Top-down computation tree for the change-making problem.

- **Optimal substructure**: An optimal solution can be computed from optimal solutions of its subproblems. This allows us to keep paths that are optimal, because they will be useful for calculating the final solution.
- **Overlapping subproblems**: This property states that the problem can be broken down into smaller problems, and many of these smaller problems are the same. This allows us to save computations by storing results.

We have already seen optimal substructure. Overlapping subproblems is needed because otherwise, there would be no point in remembering the subproblem answers (as in our change-making example). The optimal substructure property guarantees that the solutions to our subproblems are still useful.

Intuitively, a problem can be improved by dynamic programming if its computation tree actually forms a graph (same nodes appear at multiple places in the tree).

2.4 Memoization

One simple way to memorize previous results is to perform the computation top-down. As we solve subproblems, we *memoize* their solutions, so that they do not need to be computed when we encounter them again.

Figure 2 shows the bottom-up memoization approach for the Fibonacci problem, and Figure 5 for the coin problem.

In this approach, we only solve subproblems as needed. When we do solve it, we cache the solution (store it in a hashmap or something). In practice, this is easier to code, since we solve the problems exactly as before.

2.5 Dynamic programming

Technically, *memoization* is a type of dynamic programming, but dynamic programming usually refers to the bottom-up approach. This is exemplified in

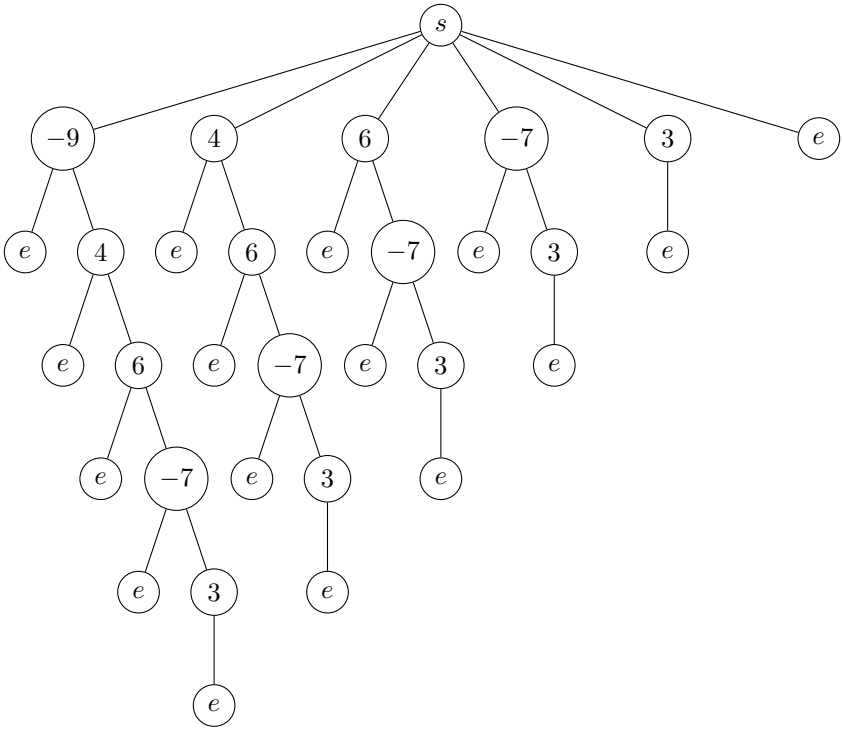

Figure 6: Computation tree for the maximum subarray problem. s stands for start, and e stands for end.

Figures 3 and 4.

This approach computes smaller solutions in the hopes that they will be useful for reaching the larger solutions. This is typically harder to program, because the memorization is implicit in the algorithm.

3 Examples

3.1 Maximum contiguous subarray

Problem 3. *Given an integer array, find the subarray with the largest sum. The subarray must be contiguous, i.e. it must contain every element between its start and end.*

For example, for the array `[-9,4,6,-7,3]`, the solution is `[4,6]`. The computation tree for this array is:

In this problem, we must search the whole tree, because we do not have a fixed target to reach. We just want to maximize the sum along the path.

Note that there are many many overlapping problems here. Thus, we can reduce the problem to the graph in Figure 7.

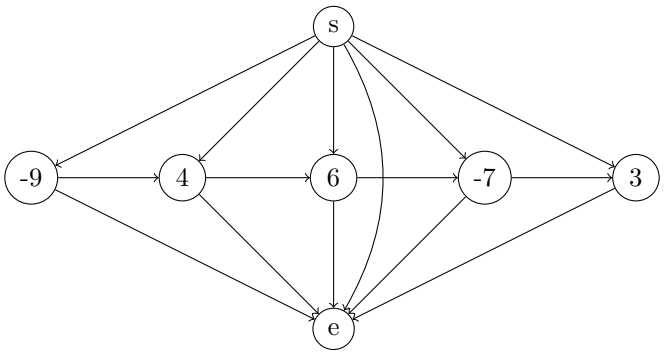

Figure 7: Reduced graph for the maximum subarray problem.

Along with this reduced computation graph, we can take advantage of the following facts:

- If our current path sum is 0, we can discard the entire path, since the empty path is better.
- If our current path is positive, and
  - if the node to the right is positive, we move to it, since we can increase the current path sum.
  - otherwise, we end the path, and record it.
- If we have already visited a node from the left, we can disregard it; this is because it has already been considered in a larger sum path.

Putting these facts together, we can come up with an $O(n)$ algorithm which traverses the list only once.

3.1.1 Algorithm

For simplicity, we return the sum over the maximum sequence. By simple extension we can return the start and the end of the maximum sequence instead.

```
max_subarray(a) {
  sum = 0
  max_sum = 0
  // i here is the number, not the index
  for (i in a)  {
    if (i > 0) {
      sum += i
    } else {
      max_sum = sum
      // if negative is larger than sum, then not worth it
```

```
      if ((sum + i) < 0) {
        sum = 0
      }
    }
  }
  return(max(sum, max_sum))
}
```

3.2 Edit distance

Problem 4. *There are two strings, x and y. We want to calculate the edit distance between them. Edit distance is the minimal amount of insertions, deletions, and edits to morph one string into the other.*

This problem is often encountered in comparing DNA sequences.
As an example, the edit distance between `at` and `sit` is 2.

$$\texttt{at} \rightarrow \texttt{sat} \rightarrow \texttt{sit}$$

We have to be careful about how we define the computation tree, otherwise it can be enormous. Note that to properly calculate edit distance, we need to:

- Compare characters in x with characters in y.
- Start this comparison from the beginning of both x and y.
- Finish at the end of x, y, or both.

Thus, we define each node to be a comparison of a character in x and a character in y. Thus, we start at s-s,, and move up either one character in x (addition), one character in y (deletion), or both (matching), until we get to the end of one string.

The partial computation tree (with repeated subtrees not computed) is shown in Figure 8. The goal is to maximize matches, and minimize every other type of node.

The Wagner-Fischer algorithm notes that this type of tree can actually be represented using a matrix, as seen in Figure 1:

- Top left represents root.
- Edges represent leaves.
- From each square, we can move below (insert), to the right (delete), or bottom right (match).
- The values in the square represent the cumulative distance of the path.
- The goal is to find the path from the top left to an edge with the minimum distance.

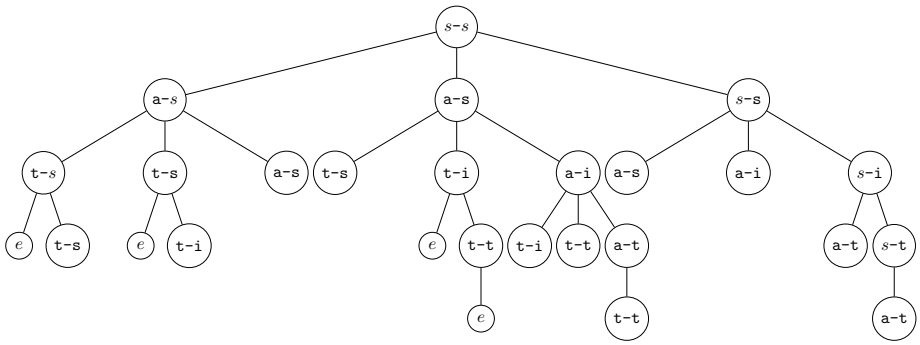

Figure 8: Partial computation tree for the edit distance problem, for strings `at` and `sit`. s stands for start, e represents the end of matching.

		a	t
	0	1	2
s	1	1	2
i	2	2	2
t	3	3	2

Table 1: Matrix generated by the Wagner-Fischer algorithm for computing edit distance

- Each node's value is the minimum of its predecessors' values, plus one if it's not a match.

The last fact takes advantage of the optimal substructure of the problem. By condensing the overlapping problems in the computation tree into a matrix form, we reduce computation time to $O(mn)$, where m and n are the lengths of the two strings.