CS 3510 — Algorithms, Spring 2022

Lecture 7: Dynamic Programming

February 1, 2022

Lecturer: Frederic Faulkner *Scribe: Aditya Diwakar*

The exam was moved to Feb 10. You can bring a single sheet of normal printer paper (single side, handwritten). Topics cover Lectures 1-6.

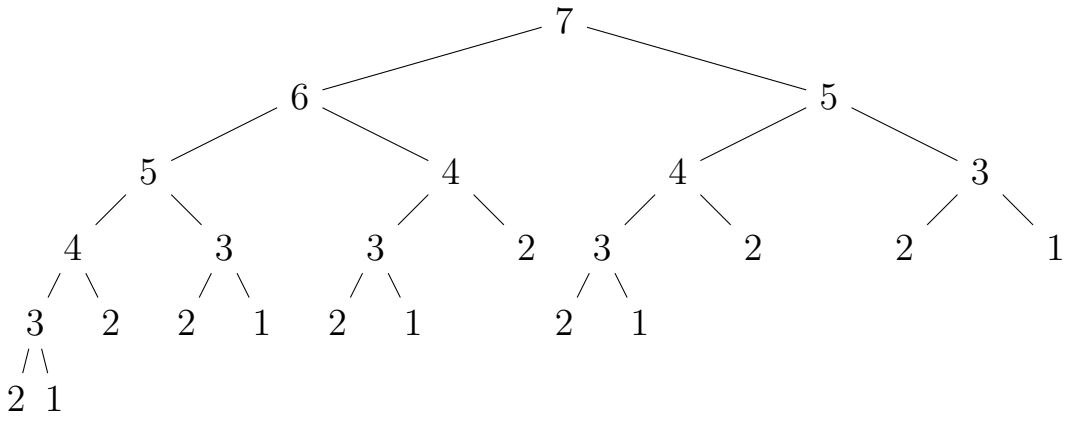
Dynamic Programming

Using a normal approach, how long does it take to compute the Fibonacci sequence? The sequence goes like $1, 1, 2, 3, 5, 8, 13, \ldots$ where the nth term is the sum of the $(n-1)$ and $(n-2)th$ term.

A naive algorithm for this could be given by the following algorithm:

```
1 Function Fib(n):
2     if n = 0 ∨ n = 1 then
3         return 1
4     return Fib(n − 1) + Fib(n − 2)
```

For example, let us compute the recursive tree for $Fib(7)$:



This tree contains an exponential amount of work, but also a lot of wasted work as we are recomputing many duplicate values (that were already computed before).

For example, we are completely recomputing the entire recursive tree for $Fib(5)$ on the left most node on level 3 when it is already being computed as the right most node on level 2.

A better approach is to use *dynamic programming* where we start with an empty list and build up the answer from there (being able to remember the computation for smaller subproblems).

In the following algorithm, we initialize a list and include the two base cases in the first two enties (first index is 1). There are no recursive calls, but we are still exploiting the recursive structure of the problem.

```
1 Function Fib(n):
2     T ← []
3     T[1] ← 1
4     T[2] ← 1
5     for i ∈ 2 → n do
6         T[i] ← T[i − 1] + T[i − 2]
7     return T[n]
```

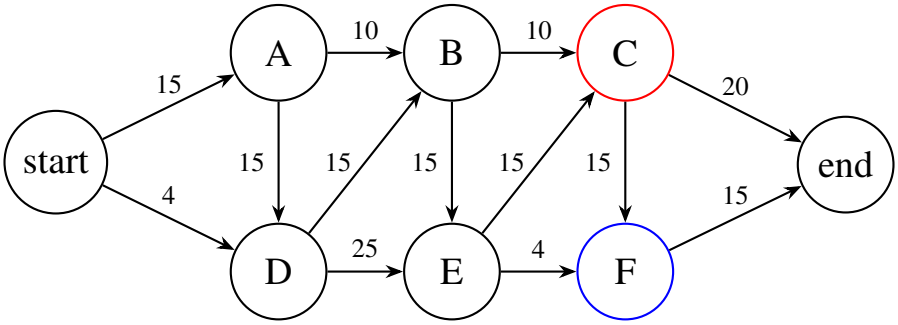
In general, dynamic programming is great because

- Exploiting recursive structure without recursion
- Solve the smaller problems first and store the solutions to avoid recalculation

Let's see another example on the next page!

Route Planning Example

Let's say you have a start and end point in a grid, and you are curious about how long it takes to get from the start to the end.



If we pick the two nodes (colored red and blue), then the time that it takes to get to the end can be defined recurively (T is the time it takes to get to a certain node):

$$T(\texttt{end}) = \min(T(C) + 20, T(F) + 15)$$

A recursive approach this would have a lot of wasted computation (recomputing the distance to nodes we already have a stored value for). Rather, we opt for a dynamic programming approach.

Idea: Since we need all the subproblems, calculate the subproblems first and then use those to propogate up the problem (larger and larger problems).

First, we know that $T(\texttt{start}) = 0$, then $T(A) = 15$ and $T(D) = min(T(A) + 15, 4) = 4$. Repeating this logic gives us:

$$\begin{aligned}
T(\texttt{start}) &= 0 \\
T(A) &= T(\texttt{start}) + 15 = 15 \\
T(D) &= \min(T(\texttt{start}) + 4, T(A) + 15) = \min(4, 19) = 4 \\
T(B) &= \min(T(A) + 10, T(D) + 15) = \min(25, 19) = 19 \\
T(E) &= \min(T(B) + 15, T(D) + 25) = \min(34, 29) = 29 \\
T(C) &= \min(T(B) + 10, T(E) + 15) = \min(29, 44) = 29 \\
T(F) &= \min(T(E) + 4, T(C) + 15) = \min(33, 44) = 33 \\
T(\texttt{end}) &= \min(T(C) + 20, T(F) + 15) = \min(49, 48) = \boxed{48}
\end{aligned}$$

To formalize this example into an algorithm: the shortest path in a DAG (Directed Acyclic Graph) is:

```
1 Function ShortestPath(G, v):
      /* topologically sort the DAG, covered in Unit 3 of this course  */
2     G_T ← TopoSort(G) ;                          // graph is now "ordered"
3     T ← {} ;                          // dynamic programming map <edge: time>
4     T[s] ← 0 ;                                  // takes no time to get to start
      /* iterating over edges making use of topological sort           */
5     for edge(x, y) ∈ G_T do
          /* checking if there is a faster way get to y                */
6         T[y] ← min(T[y], T[x] + weight(x, y))
7     return T[v]
```

The running time for the `TopoSort` is $O(n^2)$ where n is the number of vertices, and there can be as many as $|V|^2 = n^2$ edges, so the loop iterates a total of n^2 times gives $O(n^2)$ total runtime.

Dynamically Robbing Houses Example

You have houses $h_1, \ldots, h_n$ where h_i is the amount of the i^{th} house. You are a robber and cannot rob consecutive houses. What is the maximum wealth you can accure?

For every dynamic programming problem, there are 4 parts for a total answer:

1. Table entry definition
2. Recursive relationship between entries of the table
3. Pseudocode (for the algorithm, this unit only)
4. Running time analysis

Let $T[i]$ be the maximum wealth from houses $m_1, \ldots, m_i$ (this is the table entry definition). Now, we want to know how to define $T[i]$ in terms of smaller i.

We could choose to rob the i^{th} house, but then we could not rob the previous house, so our sum would be $T[i] = h_i + T[i-2]$. However, we could also choose not

to rob this house, so $T[i] = T[i-1]$ and since we want to maximize how much money to make, the recursive relationship between entries would be defined as:

$$T[i] = \max(h_i + T[i-2], T[i-1])$$

For pseudocode, we can implement this as:

```
1 Function Robbery(h_1, ..., h_n):
      /* let T be an empty array that is n in length */
2     T ← []
3     T[1] ← h_1 T[2] ← max(h_1, h_2)
      /* this loop is inclusive on n */
4     for i ∈ 3 → n do
5         T[i] ← max(h_1 + T[i-2], T[i-1])
6     return T[n]
```

This algorithm has a total runtime given by $O(n)$ because all operations are $O(1)$ while the loop is iterating $O(n)$ times meaning we are making $O(n)$ $O(1)$ operations, giving $O(n)$ final runtime.

Challenge: Hopscotch Example

Let's play a game of hopscotch! There is a straight line of squares with integers values on each square. We start at the start and eventually reach the end.

The score is the sum of the squares that you land on. *However*, you can only jump foward by 1 or 3 steps. What is the maximum score?

Try this question. Answer is on the next page.

Answer

We can represent $T[i]$ has the maximum score for the squares $s_1, \ldots s_i$, then we can give a recurrence relation as:

$$T[i] = \max(T[i-1], T[i-3]) + s_i$$

In pseudocode, it could be given as:

```
1 Function Hopscotch(s_1, ..., s_n):
      /* let T be an empty array that is n in length                    */
2     T ← []
3     T[1] ← s_1
4     T[2] ← s_1 + s_2
5     T[3] ← s_1 + s_2 + s_3
6     for i ∈ 4 → n do
7         T[i] = max(T[i − 1] + s_i, T[i − 3] + s_i)
8     return T[n]
```

This algorithm has a runtime of $O(n)$ since the initialization of the array takes $O(n)$ time (for n-long array) and the base case updates are all $O(1)$. The for loop iterates a total of $O(n)$ times giving a final runtime of $O(n)$.