CPSC 365 / ECON 365: Algorithms March 1, 2022

Lecture 11

Lecturer: Andre Wibisono *Scribe: Alden Tan Ming Yang*

1 Overview

This lecture introduces dynamic programming through the example of Fibonacci numbers, and discusses Directed Acyclic Graphs and topological sort. Specifically, these lecture notes cover:

- Introduction to Dynamic Programming
- Fibonacci Numbers
- Directed Acyclic Graphs

2 Introduction to Dynamic Programming

2.1 Overview

To solve problems of size n:

- Decompose the problem into subproblems of smaller size.
- Solve subproblems in some order: eg. from smallest to largest.

2.2 Dynamic Programming Recipe

More specifically:

- Figure out what the subproblems are. Usually, we maintain the solutions to subproblems in an array or table with entries $T(i)$ or $T(i, j)$, etc. Then, identify what your solution is in terms of the table entries (e.g. $T(n)$).
- Find a recurrence relation between subproblems.
- Compute solutions from smallest to largest (to avoid repetition).

3 Fibonacci Numbers

Let $F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)\ \forall n \geq 2$.

This counts the number of ways to tile a $1 \times n$ board by 1×1 and 1×2 tiles. To see this, notice that when deciding on the first tile, you can either use a 1×1 or 1×2 tile. If you use a 1×1 tile, the remaining board length is $n - 1$. If you use a 1×2 tile, the remaining board length is $n - 2$. Therefore, the number of possible tilings of a $1 \times n$ board $= T(n) = T(n-1) + T(n-2)$.

The exact solution to the Fibonacci recurrence is

$$F(n) = \frac{\phi^n - (-\phi)^{-n}}{\sqrt{5}}$$

where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio, and $-\frac{1}{\phi} = 1 - \phi \approx -0.618$. Thus, $F(n) \approx \frac{\phi^n}{\sqrt{5}} \approx 1.618^n$ grows exponentially.

3.1 Recursive Implementation

A naive recursive implementation of the Fibonacci function takes exponential time.

Algorithm 1 Recursive Fibonacci

```
1: function FIB(n)
2:     if n = 0 or 1 then
3:         return n
4:     end if
5:     FIB(n − 1) + FIB(n − 2)
6: end function
```

The running time $T(n)$ of this algorithm satisfies the recursion

$$T(n) = T(n-1) + T(n-2)$$

This algorithm is wasteful as it repeats many computations.

3.2 Recursive with Memoization

As an optimization, let's remember answers to previous computations, which we call memoization. Here, the term "Memoize" refers to "writing a memo."

Algorithm 2 Recursive Fibonacci with Memoization

```
1:  function FibMemo(n)
2:      if n = 0 or 1 then
3:          return n
4:      end if
5:      if n is in Memo then
6:          return Memo[n]
7:      end if
8:      Memo[n] = FIBMEMO(n − 1) + FIBMEMO(n − 2)
9:      return Memo[n]
10: end function
```

Here, we store answers in a dictionary `Memo`, so that subsequent queries take $O(1)$. The number of recursive calls made is $O(n)$, as each subproblem $\texttt{Fib}(n-1), ..., \texttt{Fib}(0)$ is only computed once. Therefore, the running time is linear: $T(n) = O(n)$.

3.3 Fibonacci Dependencies Graph

To draw a directed graph of dependencies of subproblems:

- Nodes are subproblems: $v_0 = \texttt{Fib}(0)$, $v_1 = \texttt{Fib}(1)$, ..., $v_n = \texttt{Fib}(n)$.
- Put a directed edge $e_{i \to j} = (v_i, v_j)$ if problem v_i is required to solve problem v_j.
- The graph should be acyclic: does not have a (directed) cycle. If it has a cycle, then the problem cannot be solved.
- For the Fibonacci example, edges are $(v_i, v_{i+1}), (v_i, v_{i+2})\ \forall\ 0 \le i \le n$ because we have the recursion $\texttt{Fib}(i+2) = \texttt{Fib}(i+1) + \texttt{Fib}(i)$.
- Reorder computation from smallest to largest (bottom-up).

3.4 Fibonacci: Bottom-up approach

Algorithm 3 Bottom-up Fibonacci

```
1: function Fib2(n)
2:     F[0] = 0
3:     F[1] = 1
4:     for i from 2 to n do
5:         F[i] = F[i − 1] + F[i − 2]
6:     end for
7:     return F[n]
8: end function
```

This computes the same answers as the recursive with memoization approach, but the computation now proceeds from the smallest to largest. Since we are filling in an array `F` of size $n+1 = O(n)$, and each entry takes $O(1)$ time to compute, the running time $T(n) = O(n) \times O(1) = O(n)$ is linear.

3.5 Dependencies Graph

We want the dependencies graph to be a Directed Acyclic Graph (DAG).

- We can always linearize (topologically sort) a DAG. I.e., put an ordering on the verticies $1, ..., n$ such that all edges $e = (i, j)$ go from smaller to larger vertices: $i < j$.
- Then, we solve problems from smallest to largest.

4 Directed Acyclic Graphs (DAG)

4.1 Directed Graph

- Let $G = (V, E)$ be a directed graph on vertex set V with edge set E.
- Each edge $e = (i, j)$ is directed from $i \in V$ to $j \in V$.
  - Not necessary symmetric: $(i, j) \in E$ does not imply $(j, i) \in E$.
  - Undirected graph is equivalent to a symmetric directed graph: $(i, j) \in E \Leftrightarrow (j, i) \in E$.

4.2 Directed Acyclic Graph

Let $G = (V, E)$ be a directed graph on vertex set V with edge set E.

- G is acyclic if it has no (directed) cycle.

- In this case, we say G is a DAG.
- How to check if a directed graph G is a DAG? Ans: Use DFS and check if there is no back edge.
- Lemma: G is a DAG $\Leftrightarrow$ there is no back edge in DFS search tree.

4.3 Topological Sort

Let $G = (V, E)$ be a DAG.

- A topological ordering is a labelling $V = \{1, ..., n\}$ such that each edge $e = (i, j) \in E$ goes from smaller to larger indices: $i < j$.
- Theorem: G is a DAG if and only if it has a topological ordering.
- Note: There can be multiple topological orderings.

Figure 6.1 A dag and its linearization (topological ordering).

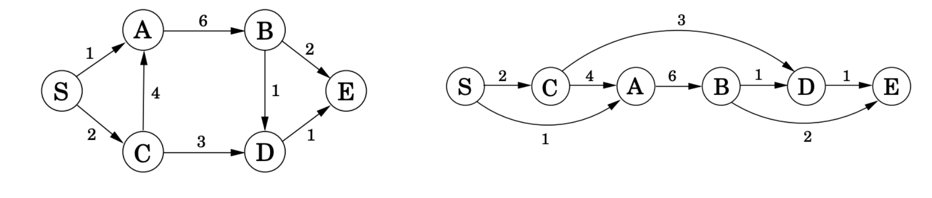

Figure 1: Topological ordering example 1 from lecture

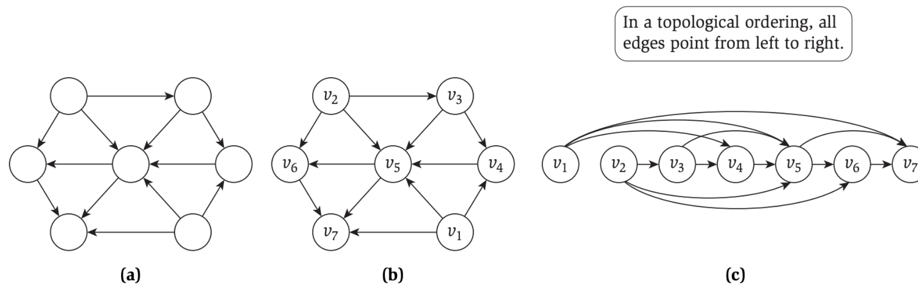

Figure 2: Topological ordering example 2 from lecture

4.4 Finding topological orderings

4.4.1 Approach 1: Depth-First Search

- Explore G using DFS.
- Keep track of time $t(v)$ when you have finished exploring v and its descendants.
- Order the vertices from the largest $t(v)$ to smallest $t(v)$.
- Lemma: If G is a DAG, then the above produces a topological ordering. That is, each edge $e = (v_i, v_j) \in E$ has $t(v_i) > t(v_j)$.
- Thus, we can find topological ordering in linear time.

4.4.2 Approach 2: Greedy method

- Lemma: If G is a DAG, then there is a vertex $s \in V$ with no incoming edge.
- To find a topological ordering on G:
  - Find a node v with no incoming edges, and order it first.
  - Delete v from G.
  - Recursively compute a topological ordering of $G - \{v\}$ and append this order after v.

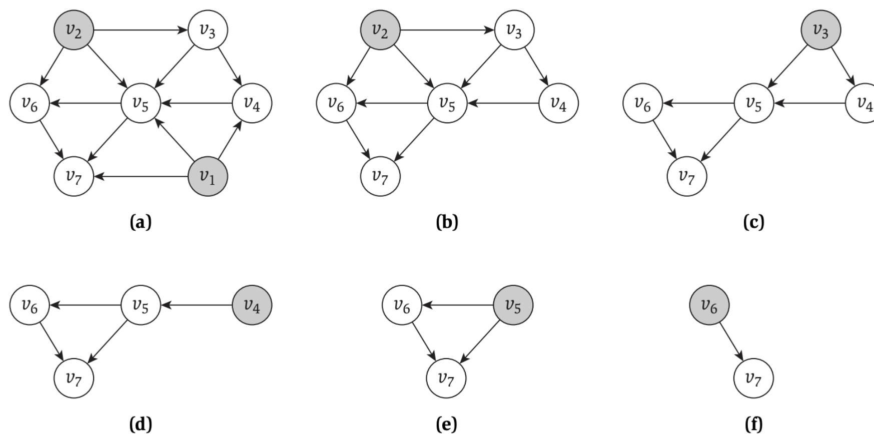

Figure 3.8 Starting from the graph in Figure 3.7, nodes are deleted one by one so as to be added to a topological ordering. The shaded nodes are those with no incoming edges; note that there is always at least one such edge at every stage of the algorithm's execution.

Figure 3: Greedy method of finding topological ordering