Ian Piper
CSCI203

Algorithms and Data Structures

Week 10 – Lecture A

Dynamic Programming

- Dynamic Programming (DP) is a problem solving technique that is:
  - General;
  - Efficient;
  - Easy to understand.
- It is applicable to a wide range of different problems.
- It usually finds a solution in polynomial time…
  - … this is a GOOD THING™.
- It is often the only efficient technique we know for a problem.

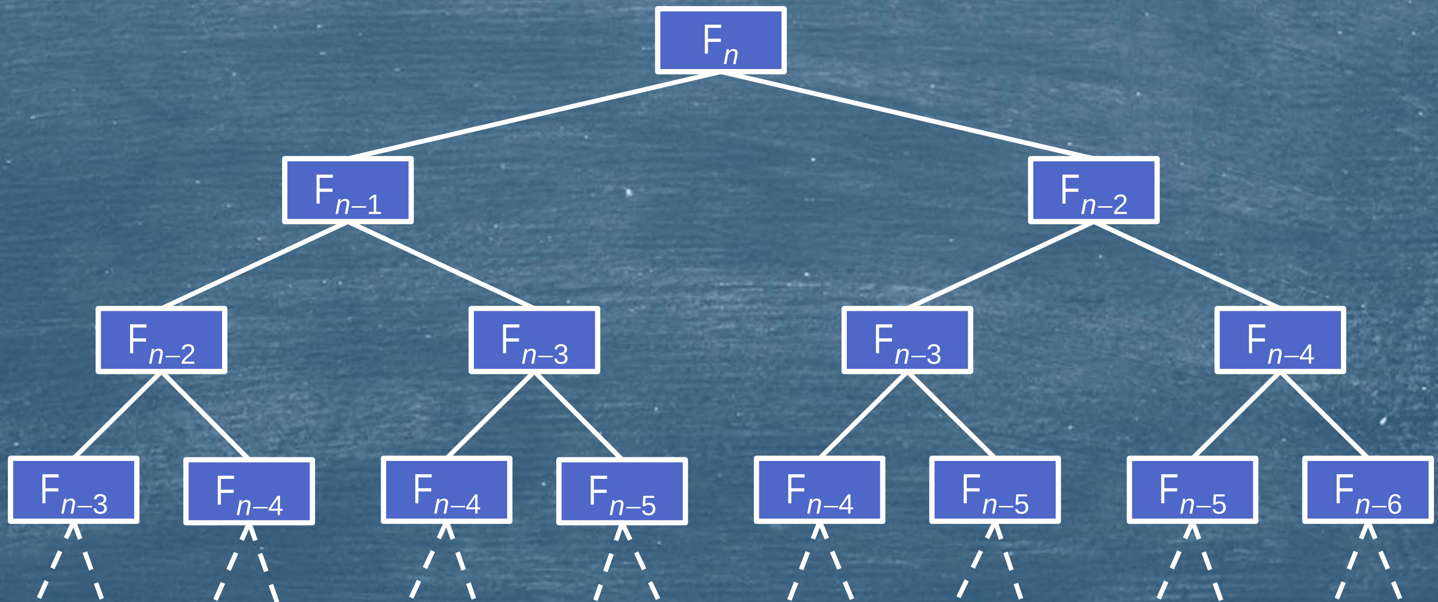
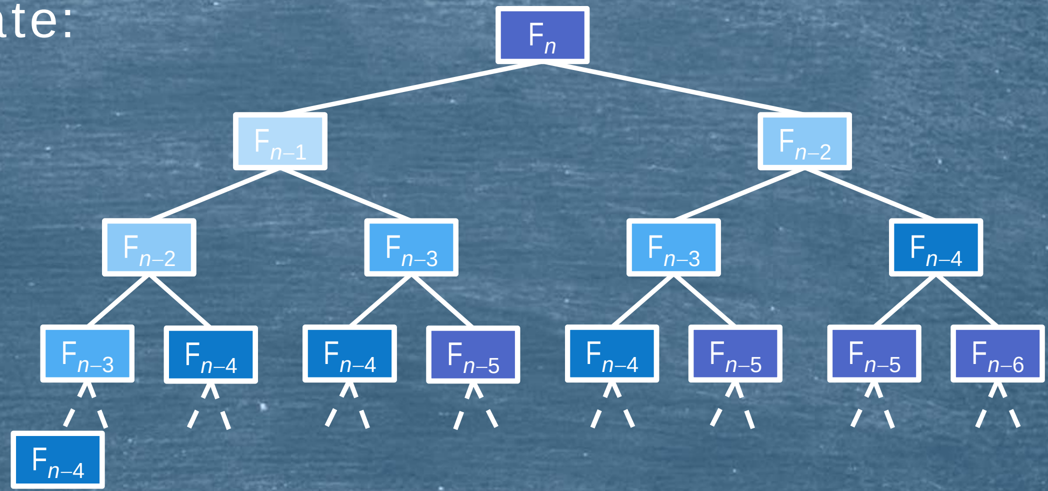
Dynamic Programming

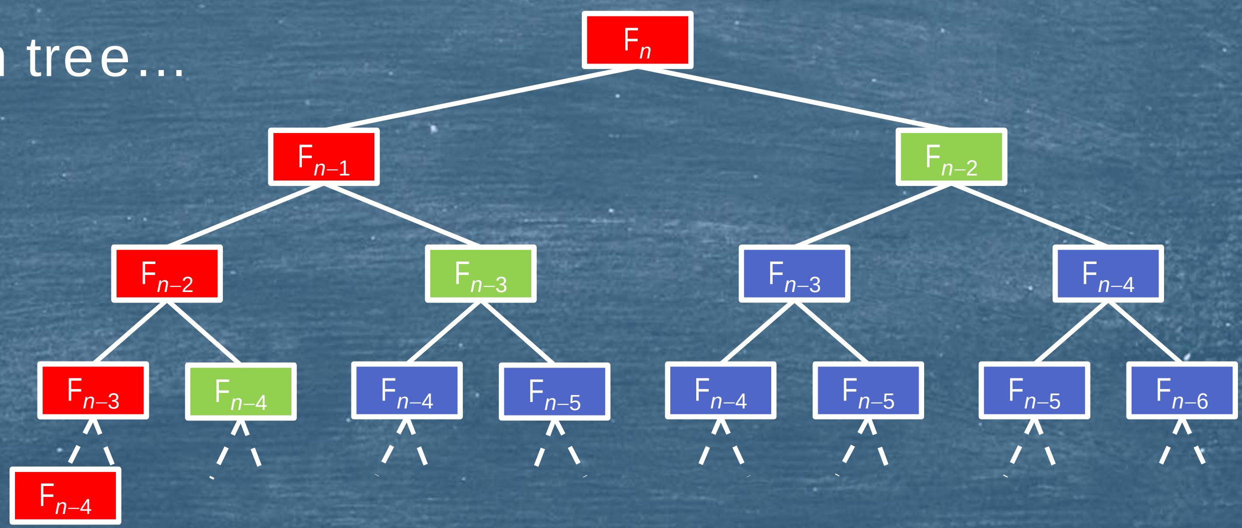
- The simplest way to think about dynamic programming is to look at it as “clever brute force”.
- That seems to be a contradiction:
  - Brute force is just looking at every possible solution;
    - Traversing the entire problem graph/tree;
  - There is nothing clever about that!
- Another way to look at it is that we:
  - Break the problem into sub-problems;
  - Re-use the solutions to the sub-problems.
- We can best see how DP works by looking at some examples.

Dynamic Programming I: Fibonacci Numbers

- We are all familiar with the Fibonnaci numbers:
  - 1, 1, 2, 3, 5, 8, 13…
- Each number is defined as the sum of its two immediate predecessors:
  - $Fib_1 = Fib_2 = 1$;
  - $Fib_n = Fib_{n-1} + Fib_{n-2}$, otherwise.
- We can compute Fibonacci numbers directly from this definition.

Recursive Fibonacci

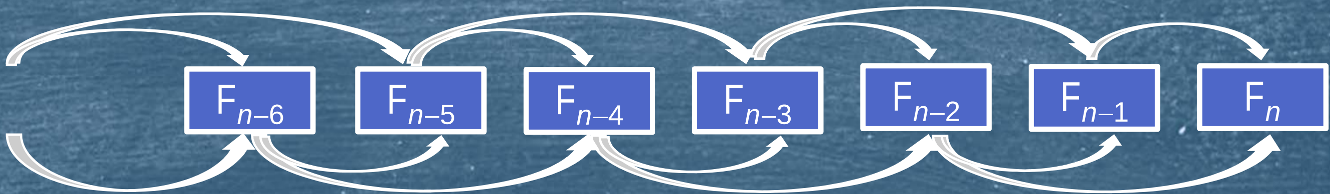
- ```
  Procedure fib(n: integer): integer
      f: integer

      if (n≤2) then
              f = 1
      else
              f = fib(n-1) + fib(n-2)
      fi
      return f
  End procedure fib
  ```
- This procedure is correct but it is not efficient.
- It is, in fact, an exponential time algorithm.
  ```

Recursive Fibonacci a BAD THING™

- From the code we can see that the time required to compute the nth Fibonacci number:
  - $T(n) = T(n-1) + T(n-2) + O(1)$
  - $T(n) > T(n-2) + T(n-2)$
  - $T(n) \in \Theta(2^{n/2})$
- Interestingly, the time taken to compute the n^{th} Fibonacci number is proportional to the n^{th} Fibonacci number.
- This is a bit like having a 1:1 scale map:
  - Accurate but hard to fold up.

Further Analysis

- Let us look at this another way
- Evaluating F_n requires that we evaluate the following tree:



- So, to get F_n we evaluate:
  - F_n once;
  - F_{n-1} once;
  - F_{n-2} twice;
  - F_{n-3} three times;
  - F_{n-4} five times;
  - Etc.



- The cost is in the repeated evaluations of the same thing.
- What if we only evaluated each of them once?
- This is the key insight in Dynamic Programming!

Memoization: the Heart of DP

- The recognition that we only need to perform a given calculation once is central to Dynamic Programming.
- How do we remember the previous evaluations?
  - We use a dictionary;
    - A hash table.
- Let us look at the DP version of our fib procedure…

Recursive Fibonacci with Memoization

```
memo: dictionary = {}
Procedure fibDP(n: integer): integer
         f: integer

    if (n in memo) return memo[n]
    if (n≤2) then
         f = 1
    else
         f = fibDP(n-1) + fibDP(n-2)
    fi
    memo[n]=f
    return f
End procedure fibDP
```

Analysis

- Now:
  - We only recurse the first time we evaluate a given Fibonacci number.
  - In all other cases we just look up the dictionary.
- Our evaluation tree...
- ...becomes:
  - Evaluate;
  - Memoize;
  - Ignore.



Analysis

- With this, Dynamic Programming, approach:
  - We compute F_k once for each value $1 \leq k \leq n$;
    - n calls;
    - O(1) per call;
  - We look up F_k once for each value $1 \leq k \leq n-1$;
    - $n-1$ calls;
    - O(1) per call.
- So, **`fibDP`** takes $O(n)$ time to compute F_n.

In General

- We can state the general technique for dynamic programming as follows:
  - Solve any sub-problem once and memoize (remember) these solutions for later re-use.
- In essence: DP is recursion + memoization.
- The critical problem in using DP is the identification of the sub-problems.
- The solution time for dynamic programming is derived as follows:
  - Multiply the number of distinct sub-problems by the solution time per sub-problem;
  - Note: we only solve a sub-problem once.

Turning Dp on its Head

- Another way to think about dynamic programming is to look at it as *bottom up* solution.
- In contrast, recursion is *top down* solution.
- We can write a bottom up Fibonacci algorithm as follows:

Bottom up Fibonacci Numbers

- ```
  Procedure fibUp(n: integer): integer
      fib: dictionary = {}

      k=1
      repeat
          if k ≤ 2 then
              f = 1
          else
              f = fib[k-1]+fib[k-2]
          fi
          fib[k] = f
          k++
      until k==n
      return fib[n]
  End procedure fibUp
  ```
- Note that this solution completely eliminates the need for recursion in calculating the n^{th} Fibonacci number.
- All dynamic programming algorithms can be transformed in this way.
  ```

Bottom Up in General

- The bottom up approach to DP still involves solving the same set of sub-problems as in the top down approach.
- What changes is the order in which we solve them.
- The bottom up order can be considered as a topological sort of the problem's dependency graph.
- For the Fibonacci numbers...



- ... so the sort order is F_1, F_2, F_3,... F_{n-3}, F_{n-2}, F_{n-1}, F_n.

Saving Space with DP

- Often, the bottom up version of dynamic programming allows us to save space (memory) as well as time.
- As we presented the algorithm, it used a dictionary containing *n* entries.
- In fact, we only ever need the last two values; we can forget the earlier ones.
- This allows us to re-write the algorithm without explicit memoization.

Memo-free Fibonacci Numbers

- ```
  Procedure fibSmall(n: integer):integer
      prev:integer = 0
      f: integer = 1

      k=2
      repeat
          f = f+prev
          prev = f-prev
          k++
      until k == n
      return f
  end procedure fibSmall
  ```
  ```

Dynamic Programming II: Shortest Paths

- Let us apply the insights we have gained on dynamic programming to a second problem.:
  - Single source, all destinations shortest path.
- We will proceed as follows:
  1. Create a top down, recursive, naïve algorithm;
  2. Memoize it;
  3. Reconstruct it as a bottom up algorithm.
- This is a useful general approach to algorithm design in dynamic programming.

Step 1: the Naïve, Recursive Algorithm.

- In deriving the naïve algorithm we need to introduce another key component of dynamic programming…
  - …guessing!
- Don't know the answer?
  - Guess!
- Don't just try any guess…
  - …try them all!
- So, DP = recursion + memoization + guessing.
- The best guess is the answer we are looking for.

Some Notation for Shortest Paths

- Remember from last week:
  - Given a graph, $G=(V, E, W)$, find the shortest path from a starting vertex, $s \in V$, to all other vertices, $v \in V$;
  - $w(u, v)$ is the weight of the edge (u, v);
  - $D(s, v)$ is the length of the shortest path between s and v.
- If some vertex, u, is on the shortest path from s to v then:
  - $D(s, v) = D(s, u) + D(u, v)$.
- Specifically, if vertex u immediately precedes vertex v in the shortest path from s to v, then:
  - $D(s, v) = D(s, u) + w(u, v)$.
- Our problem is that we don’t know which vertex, u, to try…
  - …so we guess—try them all and pick the best.

The Naïve Algorithm

- ```
  Procedure short(V{}: vertex, E{}: edge, W(): weight, s: vertex, v: vertex)
       if v==s then
              d=0
       else
              d = ∞
              for each u where (u,v) ∈ E
                     d = min(d, short(V, E, W, s, u) + w(u,v))
              rof
       fi
       return d
  End procedure short
  ```
- This is a really bad algorithm:
  - We compute the shortest path between s and every other vertex repeatedly.
- It is really easy to improve, however;
  - Memoize the computation.
  ```

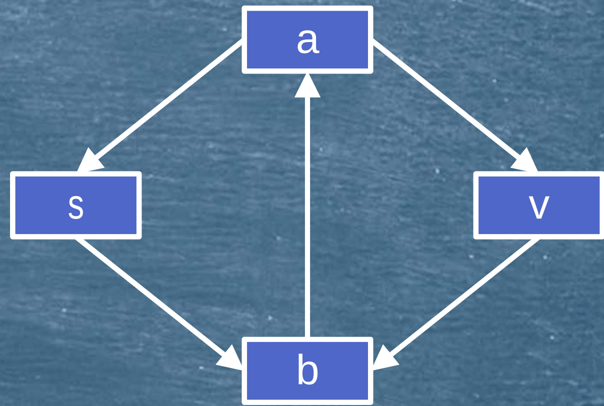
Step 2: The Memoized Algorithm

- D: dictionary {}

```
Procedure shortDP(V{}: vertex, E{}: edge, W(): weight, s: vertex, v:
vertex)
     if v==s then
            d=0
     else
            d = ∞
            for each u where (u,v) ∈ E
                   if (u in D) then
                          d = min(d, D[u] + w(u,v))
                   else
                          d = min(d, shortDP(V, E, W, s, u) + w(u,v))
                   fi
            rof
     fi
     D[v]=d
     return d
End procedure shortDP
```

Some Analysis

- Consider the following graph:



- To find the shortest path $D(s, v)$ we proceed as follows:

$$\begin{tikzcd}[column sep=small]
& D(s, v) \arrow[d] & \\
& D(s, a) \arrow[d] & \\
& D(s, b) \arrow[dl] \arrow[dr] & \\
D(s, s) & & D(s, v)
\end{tikzcd}$$

- We now have a problem…
  - …to find $D(s, v)$ we need to evaluate $D(s, v)$.

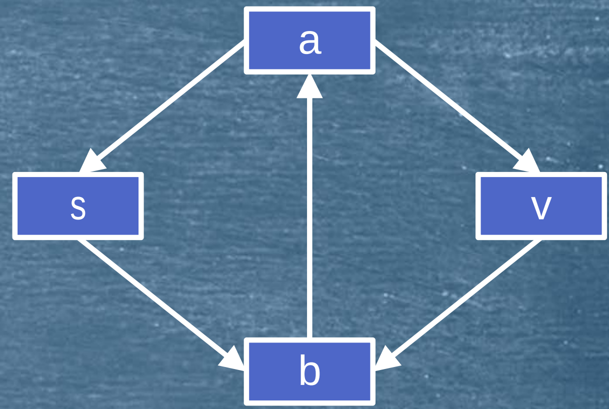
Oops

- Our “improved” algorithm has a problem.
- It takes infinite time if G has one or more cycles.
- If g is acyclic the algorithm is $O(|V|+|E|)$
- We should have anticipated this…
  - …remember the bottom up formulation.
- The order of evaluation of sub-problems is a topological sort of the dependency graph.
- You can only perform a topological sort on a DAG…
  - …no cycles allowed.

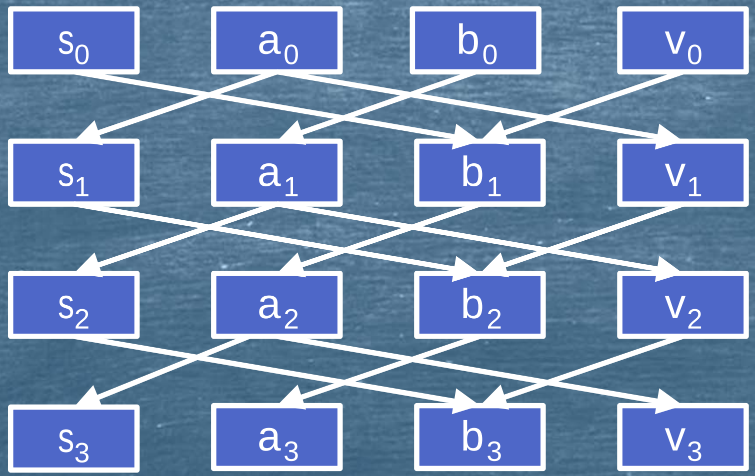
Decycling a Graph

- Is there some way to remove cycles from a graph?
- Yes, provided none of them are negative cost cycles.
- We replicate the graph $|V|$ times and construct a new graph as follows:
  - Eliminate all edges between vertices in the same copy:
  - If $(u, v) \in E$ in the original graph connect u_i to v_{i+1} in the new graph.
  - This is best seen with an example.

- Let us use our previous graph:



- This becomes:



- This new graph has $|V|^2$ vertices and $|V| \times |E|$ edges…
  - …but it has no cycles.
- We now define $D_k(s, v)$ as the shortest path from s to v that traverses exactly k edges.
- The shortest path is now the smallest of the $D_k(s, v)$ values.

So What?

- We now observe that:
  - $D_k(\underline{s}, v) = \min (D_{k-1}(s, u) + w(u, v))$.
- So, if we use our memorized DP shortest path solution algorithm on this graph we can solve our original problem, even though our graph has cycles.
- The bottom up version of this $O(|V| \times |E|)$ algorithm is exactly the same as the Bellman-Ford algorithm we saw last week.
- In fact, this is how the Bellman-Ford algorithm was discovered.