Dynamic Programming 1

Memoization, Fibonacci, Shortest Paths, Guessing

|V|-1 iterations on Bellman-Ford Algorithm?

Step 2: "V - 1" is used to calculate the number of iterations. Because the shortest distance to an edge can be adjusted V - 1 time at most, the number of iterations will increase the same number of vertices.

Path can have max |V|-1 edges

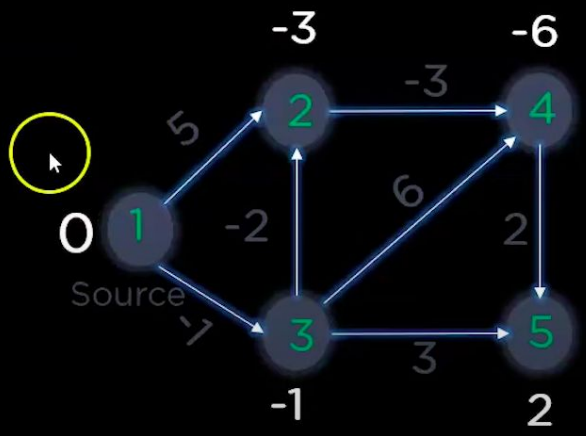
Bellman-Ford(G,W,s)

```
Initialize ()
for i = 1 to |V| − 1
    for each edge (u, v) ∈ E:
        Relax(u, v)
for each edge (u, v) ∈ E
    do if d[v] > d[u] + w(u, v)
        then report a negative-weight cycle exists
```

At the end, $d[v] = \delta(s, v)$, if no negative-weight cycles.

List of Edges

(1 , 2)
(1 , 3)
(2 , 4)
(3 , 2)
(3 , 4)
(3 , 5)
(4 , 5)



Number of relaxation

Vertices

	1	2	3	4	5
1	0	-3	-1	2	2
2	0	-3	-1	-6	-4
3	0	-3	-1	-6	-4
4	0	-3	-1	-6	-4

Dynamic Programming (DP)

Large class of seemingly **exponential** problems have a **polynomial** solution via DP.

Particularly for **optimization** problems (**min** / **max**) (e.g., **shortest** paths, **longest** common substring)

DP: controlled brute force / recursion + re-use

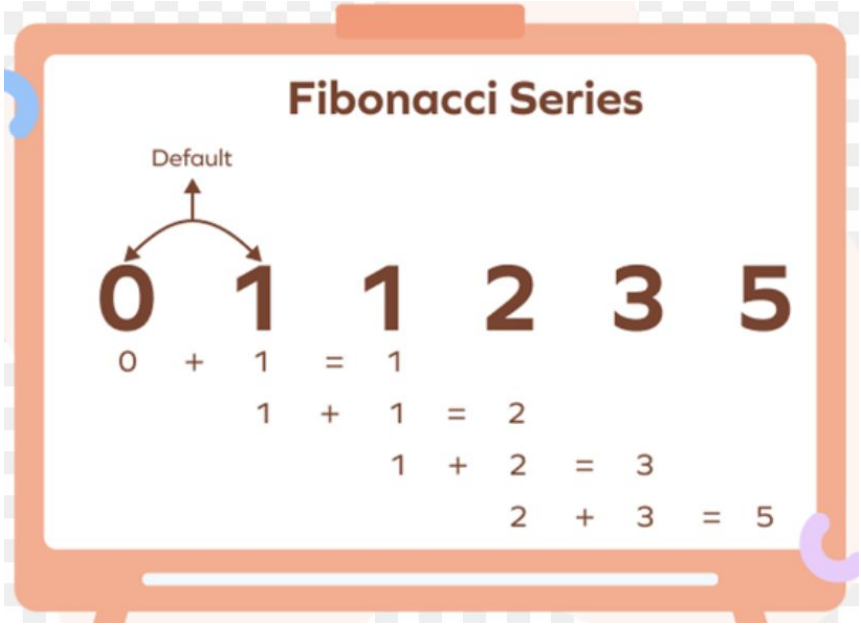
Precursor of Bellman-Ford Algorithm!

*“when one can break a **problem** into more minor issues that they can break down even further, into even more minor problems. Additionally, these **subproblems** have **overlapped**. That is, they require **previously calculated values** to be **recomputed**.”*

Fibonacci Numbers

$$F_1 = F_2 = 1; \quad F_n = F_{n-1} + F_{n-2}$$

Goal: compute F_n



The Fibonacci Sequence

1,1,2,3,5,8,13,21,34,55,89,144,233,377…

1+1=2
1+2=3
2+3=5
3+5=8
5+8=13
8+13=21
13+21=34
21+34=55
34+55=89
55+89=144
89+144=233
144+233=377

Naive Algorithm (Recursion)

fib(n):

If n <= 2: return f = 1

Else: **return f = fib(n-1) + fib(n-2)**

=> T(n) = T(n-1) + T(n-2) +Θ(1)

>= Fn

>= 2T(n-2) + Θ(1) >= 2^(n/2)

Exponential! Bad.

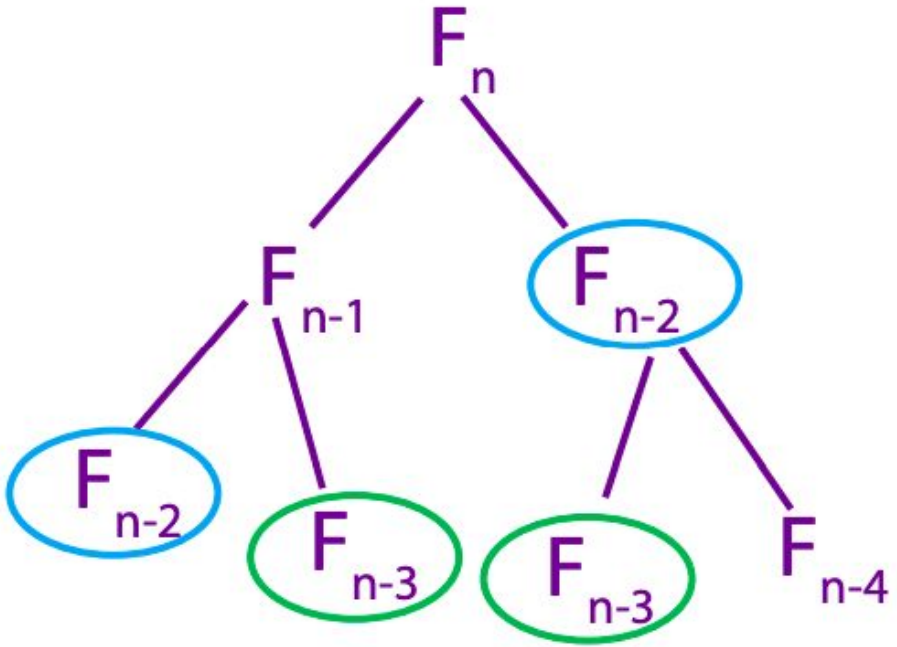


Figure 1: Naive Fibonacci Algorithm.

Memoized DP Algorithm

```
memo = {}

fib(n):
    If n in memo: return memo[n]
    else: if n <= 2: f = 1
        else: f = fib(n-1) + fib(n-2)
            memo[n] = f
            return f
```

=> fib(k) only recurses first time called, for all k

=> only n non-memoized (recursive) calls: k=n,n-1,...,1

(fib(1), fib(2), …, fib(n) **only once**!)

=> memoized calls free ($\Theta(1)$ time)

=> $\Theta(1)$ time per call (ignoring recursion)

Polynomial! Good.

DP = recursion + memoization (re-use)

- Memoize (remember) & re-use solution to subproblems that help solve problem
  - In Fibonacci, subproblems are F1, F2, ..., Fn

=> **Total running time = # of subproblems X time/subproblem**

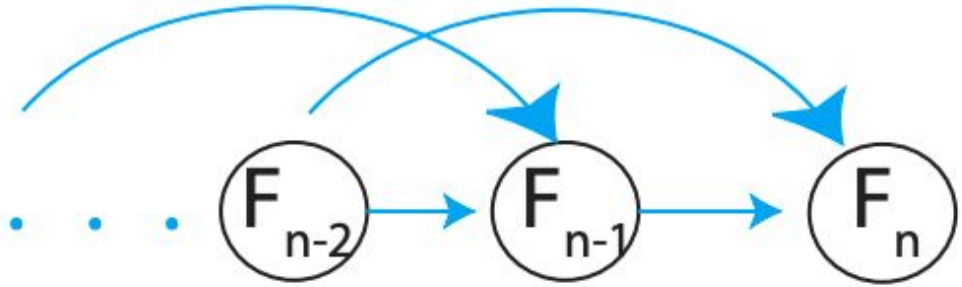
- Fibonacci:
  - # of subproblems: n (F1, F2, ..., Fn)
  - time/subproblem: $\Theta(1)$
  - Total running time = n X $\Theta(1)$ = $\Theta(n)$ (ignore recursion)

Bottom-up DP Algorithm

```
fb = {}

for k in [1, 2, …, n]:

    if k <= 2: f = 1

    else: f = fib[k-1] + fib[k-2]

    fib[k] = f

return fib[n]
```

O(1)

Θ(n)



1) Calculate each subproblem
2) Store the solutions to fb.
3) Return the very solution.

The same computation as memoized DP (recursion unrolled)

In general: **topological sort of subproblem dependency => DAG**

Practically faster: no recursion (no function call)

Can save space: just remember last 2 fibs => O(1)

[Sidenote: There is also an $O(\lg n)$-time algorithm for Fibonacci, via different techniques]

Shortest Paths in DP

Recursion formulation:

$\delta(s,v) = \min\{\delta(s,u) + w(u,v) \mid (u,v) \in E\}$
Memoized DP algorithm: takes infinite time if cycles!

Works for DAG in O(V+E)

Subproblem dependency should be acyclic (Topological sort)

Shortest Path with Dynamic Programming (MIT)

Shortest Path with Dynamic Programming

The shortest path problem has an optimal sub-structure. Supose $s \rightsquigarrow u \rightsquigarrow v$ is a shortest path from s to v. This implies that $s \rightsquigarrow u$ is a shortest path from s to u, and this can be proven by contradiction. If there is a shorter path between s and u, we can replace $s \rightsquigarrow u$ with the shorter path in $s \rightsquigarrow u \rightsquigarrow v$, and this would yield a better path between s and v. But we assumed that $s \rightsquigarrow u \rightsquigarrow v$ is a shortest path between s and v, so have a contradiction.

Based on this optimal sub-structure, we can write down the recursive formulation of the single source shortest path problem as the following:

$$\delta(s, v) = \min\{\delta(s, u) + w(u, v) | (u, v) \in E\}$$

Already remembered (re-use)

New value

```
19              result.parent[w] = v
20      return result
```

DAG

For a DAG, we can directly use memoized DP algorithm to solve this problem. The following is the Python code:

```
1  class ShortestPathResult(object):
2      def __init__(self):
3          self.d = {}
4          self.parent = {}
5
6  def shortest_path(graph, s):
7      '''Single source shortest paths using DP on a DAG.
8
9      Args:
10         graph: weighted DAG.
11         s: source
12     '''
13     result = ShortestPathResult()
14     result.d[s] = 0
```

memo?

Shortest Path DP (Memoized)

```
15      result.parent[s] = None
16      for v in graph.itervertices():
17          sp_dp(graph, v, result)
18      return result
19
20  def sp_dp(graph, v, result):
21      '''Recursion on finding the shortest path to v.
22
23      Args:
24          graph: weighted DAG.
25          v: a vertex in graph.
26          result: for memoization and keeping track of the result.
27      '''
28      if v in result.d:
29          return result.d[v]
30      result.d[v] = float('inf')
31      result.parent[v] = None
32      for u in graph.inverse_neighbors(v): # Theta(indegree(v))
33          new_distance = sp_dp(graph, u, result) + graph.weight(u, v)
34          if new_distance < result.d[v]:
35              result.d[v] = new_distance
36              result.parent[v] = u
37      return result.d[v]
```

s ~ **u** ~ v

Relaxation & memoization

Total running time of DP

The total running time of DP = number of subproblems × time per subproblem (ignoring recursion). In this case, the subproblem is represented by $\delta(s, v)$ which is parameterized by v because s is fixed. The number of possible values for v is $|V|$, so there are $|V|$ subproblems. Each subproblem takes $\Theta(\texttt{indegree}(v) + 1)$ time. So the total time is $\Theta(\sum_{v \in V} \texttt{indegree}(v) + 1) = \Theta(E + V)$ by Handshaking Lemma.

For the bottom-up version, we need to topologically sort the vertices to find the right order to compute $\delta(s, v)$.

# of subproblems = |V|
time/subproblem = O(indegree(v) + 1)
Total running time = O(**indegree(v)** + 1) for all v belongs to V
= O(**E** + V)

Shortest Path Bottom Up

```
1  def shortest_path_bottomup(graph, s):
2      '''Bottom-up DP for finding single source shortest paths on a DAG.
3
4      Args:
5          graph: weighted DAG.
6          s: source
7      '''
8      order = topological_sort(graph)
9      result = ShortestPathResult()
10     for v in graph.itervertices():
11         result.d[v] = float('inf')
12         result.parent[v] = None
13     result.d[s] = 0
14     for v in order:
15         for w in graph.neighbors(v):
16             new_distance = result.d[v] + graph.weight(v, w)
17             if result.d[w] > new_distance:
18                 result.d[w] = new_distance
19                 result.parent[w] = v
20     return result
```

Relaxation & update all subproblems

Graph with Cycles? Remove cyclic dependency (con't)

Graph with Cycles

In order for DP to work, the subproblem dependency should be acyclic, otherwise there will be infinte loops. We can create more subproblems to remove the cyclic dependencies. Let $\delta_k(s, v)$ be the shortest $s \rightsquigarrow v$ path using $\leq k$ edges. Then we can redefine the recurrence as the following:

$$\delta_k(s, v) = \texttt{min}\{\delta_{k-1}(s, u) + w(u, v) | (u, v) \in E\}$$

The base cases are:

$$\delta_0(s, v) = \infty \text{ for } v \neq s$$
$$\delta_k(s, s) = 0 \text{ for any } k$$

If there are no negative cycles, $\delta(s, v) = \delta_{|V|-1}(s, v)$ because the maximum possible number of edges of a simple path is $|V| - 1$.

We can visualize this as a graph transformation as well. Let $G = (V, E)$ be a directed graph with cycles. For every $v \in V$, make $|V|$ copies of v as $v_0, v_1, \ldots, v_{|V|-1}$ in the new graph G'. For every edge $(u, v) \in E$, create an edge (u_{k-1}, v_k) for $k = 1, \ldots, |V| - 1$ in G'.

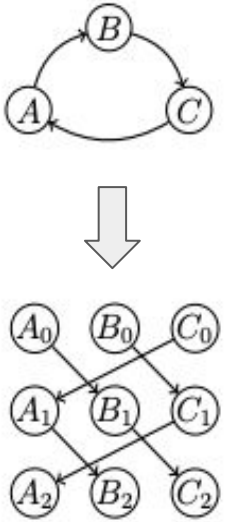


Figure 1: Transforming a cyclic graph into an acylic graph.

Graph with Cycles? Remove cyclic dependence

- Goal: $\delta(s, v) = \delta_{|V|-1}(s, v)$ (if no negative cycles)
- memoize
- time: $\underbrace{\#\text{ subproblems}}_{|V|\cdot|V|} \cdot \underbrace{\text{time/subproblem}}_{O(v)} = 0(V^3)$

Copied # of |V| on |V| subproblems

- actually $\Theta(\text{indegree}(v))$ for $\delta_k(s, v)$
- $\implies$ time $= \Theta(V \sum_{v \in V} \text{indegree}(V)) = \Theta(VE)$

BELLMAN-FORD!

The subproblem is parameterized by two variables k and v. The number of values k can take is $|V|$, and the number of values v can take is $|V|$ as well. Time per subproblem is the same as before: $\Theta(\texttt{indegree}(v)+1)$. To total time is $\Theta\big(V \sum_{v \in V} \texttt{indegree}(v)+1\big) = \Theta(VE)$. Note that this is the same running time as Bellmand-Ford algorithm, and you should observe the similarities between the two algorithms.

Guessing

How to design recurrence

- want shortest $s \to v$ path

- what is the last edge in path? dunno
- guess it is (u, v)
- path is shortest $s \to u$ path $+$ edge (u, v)
  by optimal substructure
- cost is $\delta_{k-1}(s, u) + w(u, v)$
  another subproblem
- to find best guess, try all ($|V|$ choices) and use best
- * key: small (polynomial) # possible guesses per subproblem — typically this dominates time/subproblem

* DP $\approx$ recursion + memoization + guessing