THEORY

OF

DYNAMIC PROGRAMMING

This page intentionally left blank

Dynamic Programming (DP)

Credits : Erik Demaine

- General, powerful algorithmic design technique
- Good for optimization problem
- "Careful brute force"
- Invented by Richard Bellman in 1953, an american applied mathematician
- Also known as Bellman equation.
- Basic idea : subproblems + "reuse".

Fibonacci numbers :

Origin : Sanskrit "chandas", Pingala (200 BC), Virahanka (700 AD)

Ref : "Liber Abaci" 1202, Fibonacci

- (first western book to describe "arabic" numbers) a.k.a "The Book of Calculations" or "The Book of Abacus"
- Growth of an "idealized" rabbit population

$$F_1 = F_2 = 1$$

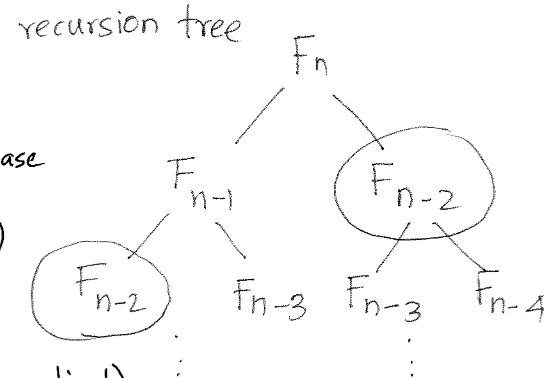
$$F_n = F_{n-1} + F_{n-2}$$

Goal: Compute F_n

Naïve recursive algorithm :

```
fib(n):
  (*) [ if n ≤ 2 : f = 1          // base case
      [ else : f = fib(n-1) + fib(n-2)
  return f
```

recursion tree



Correct Algorithm, but not efficient (exponential)

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

$$T(n) > 2T(n-2)$$

$$T(n) = \Theta(2^{n/2})$$

Memoized DP algorithm :

memo :- memo pad
- write down

```
memo = {}
fib(n):
    if n in memo : return memo[n]
  (*) [ if n ≤ 2 : f = 1          // base case
      [ else : f = fib(n-1) + fib(n-2)
    memo[n] = f
    return f
```

Note : fib (k) only recurses the first time it's called. $\forall k$

- memoized calls are $\Theta(1)$
- # of non-memoized calls is n

  fib(1), fib(2), ..., fib(n)
- non-recursive work per call $= \Theta(1)$

$\Rightarrow$ time $= \Theta(n)$

DP core idea $\approx$ recursion + memoization
- memoize (remember)
- & reuse solutions

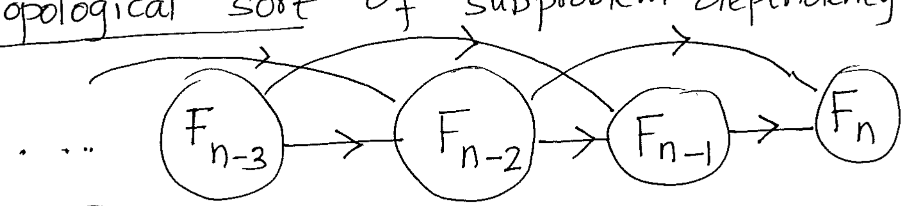
  to subproblems that help solve the problem

$\Rightarrow$ time $=$ # subproblems $\cdot \boxed{\dfrac{\text{time}}{\text{subproblem}}}$ ← ignore recursion

Bottom-up DP algorithm

```
fib = {}
for k in range (1, n+1):
    if k ≤ 2: f = 1        // base case
    else: f = fib[k-1] + fib[k-2]
    fib[k] = f
return fib[n]
```

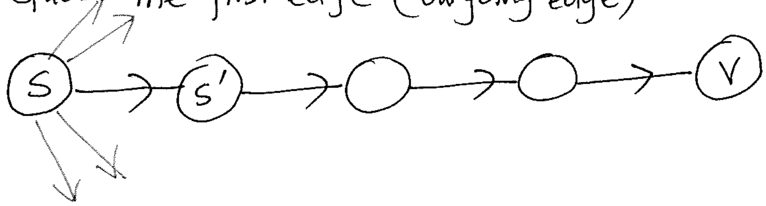
- exactly same computation
- topological sort of subproblem dependency DAG



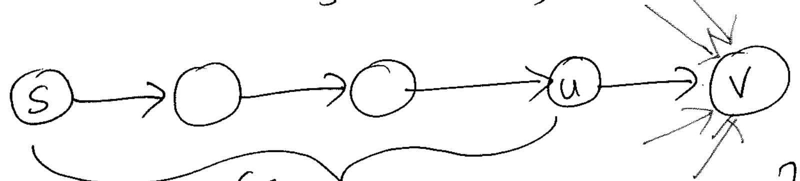
- can save space

Shortest paths $\quad \delta(s, v)\ \forall v \quad$ SSSP

- Guess the first edge (outgoing edge)



- Guess the last edge (incoming edge)



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

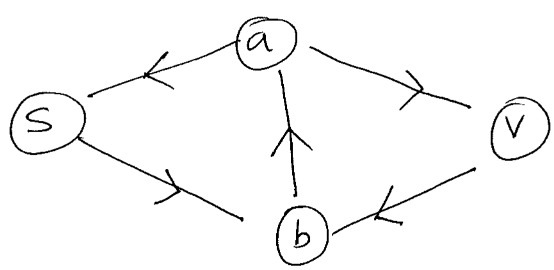
$\delta(s,s) = \emptyset$: base case

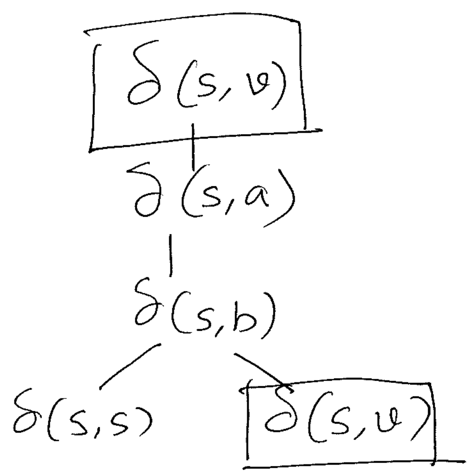
Guessing:
don't know the answer?
- guess
... try all guesses
& take the best one

Optimal substructure

subpath of shortest paths are shortest path

Is it a good algorithm?





"infinite algorithm"
on graphs with cycles

For DAG's : $\Theta(V+E)$

time for subproblem $\delta(s,v)$ = indegree(v) + 1

total time = $\sum_{v\in V}$ indegree$(v) + 1 = \Theta(E+V)$ $\quad$ Handshaking Lemma

" DFS to do a topological sort to do one round of Bellman Ford"

* subproblem dependecies should be acyclic"

- What about graphs with cycles? $\quad$ no negetive wt. cycles

$\delta_k(s,v)$ = weight of shortest path $s \to v$ that uses at most k edges

$$\delta_k(s,v) = \min_{(u,v)\in E} \left(\delta_{k-1}(s,u) + w(u,v) \right) \quad \forall v \in V,\ 0 \le k \le |V|$$

$\#$ of subproblems $= V^2$

goal: $\delta_{|V|-1}(s,v)$ total running time $= \Theta(VE)$

$\boxed{\text{DP} \approx \text{guessing} + \text{recursion} + \text{memoization}}$

5 steps in DP:

Step 1. Define subproblems and count # subproblems

Step 2. Guess (part of solution) and count # choices for guess

Step 3. Relate subproblem solutions and evaluate time/subproblem

Step 4. Recurse & memoize / build DP table bottom-up and make sure recurrences are acyclic

Step 5. Solve the original problem and evaluate total running time

5 steps of DP for Fibonacci and Shortest Path:

	Fibonacci	Shortest Path
Step 1	F_k for $k=1,\dots,n$ n	$\delta_k(s,v)$ for $v\in V$, $0\le k<\|V\|$ V^2
Step 2	nothing 1	edge into v (if any) indegree$(v)+1$
Step 3	$F_k = F_{k-1} + F_{k-2}$ $\Theta(1)$	$\delta_k(s,v) = \min_{(u,v)\in E} \{\delta_{k-1}(s,u) + w(u,v)\}$ $\Theta(\text{indegree}(v)+1)$
Step 4	for $k=1,\dots,n$	for $k=0,1,\dots,\|V\|-1$: for $v\in V$
Step 5	F_n, $\Theta(n)$	$\delta_{\|V\|-1}(s,v)\ \forall v\in V$, $\Theta(VE)$

N.B DP is NOT the ONLY way to compute SSSP

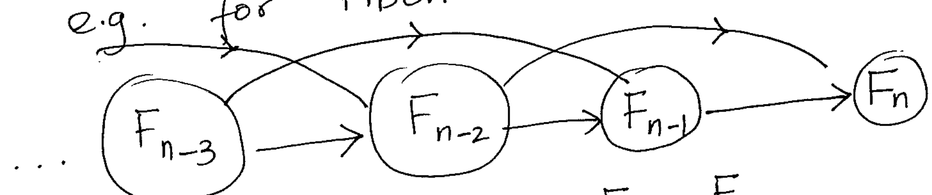
Single Source Shortest Paths (SSSP) Algorithms and Complexities by Senarios

Senarios	Algorithm	Complexity
unweighted ($w=1$)	BFS	$O(V+E)$
non negetive edge weights	Dijkstra	$O(V\log V + E)$ using Fibonacci Heap datastructure
general	Bellman - Ford	$O(VE)$
acyclic (DAG)	topological sort + 1 round of Bellman-Ford	$O(V+E)$

Traceback / Backtracking

- If you have the topological sort of subproblem dependency DAG, simply enumerate it to identify ~~print~~ the order in which the bigger subproblem is broken down into smaller ones (i.e. order of subprob)

e.g. for Fibonacci



1, 1, 2, ... F_{n-3} F_{n-2} F_{n-1} F_n

- This is used for the optimal answer rather than the optimal value. e.g. in case of SSSP, we could be interested in ~~the~~ a shortest path itself rather than the sum of edge weights of shortest path.

- Can be implemented in conjuction with memoization.

Longest Palindromic Subsequence Problem

Credits: Srinivas Devadas

Note:
- sequence : the original enumerated collection of objects (repetitions are allowed)
- subsequence : ordered; it can be non-contiguos.
- substring : ordered; must be contiguos.

Examples of palindrome : a, bb, radar

Problem Statement : Given a string $X[1, \ldots n]$ $n \geq 1$.
find out longest palindrome that is a subsequence of X having length ≥ 1.

e.g. turboventilator

rotor (5) rotator (7)

Solution Define $LPS(i, j)$ as the length of the longest palindromic subsequence of $X[i \ldots j]$; $i \leq j$

(*) look at $LPS[i, j]$ and memoize

```
LPS(i, j):
    (*) if i = j : return 1
    if X[i] = X[j] :
        if i+1 = j : return 2
        else : return 2 + LPS(i+1, j-1)
    else : return max( LPS(i+1, j), LPS(i, j-1) )
```

Without memoization; (exponential)

$$T(n) = \begin{cases} 1 & n = 1 \\ 2T(n-1) & n > 1 \end{cases}$$

$$= 2^{n-1}$$

After memoization :

$$\#\text{subproblems} \cdot * \frac{\text{time}}{\text{subproblem}} = n^2 \cdot \Theta(1) = \Theta(n^2)$$