CS 580
ALGORITHM DESIGN AND ANALYSIS

Dynamic Programming

Vassilis Zikas

DYNAMIC PROGRAMMING

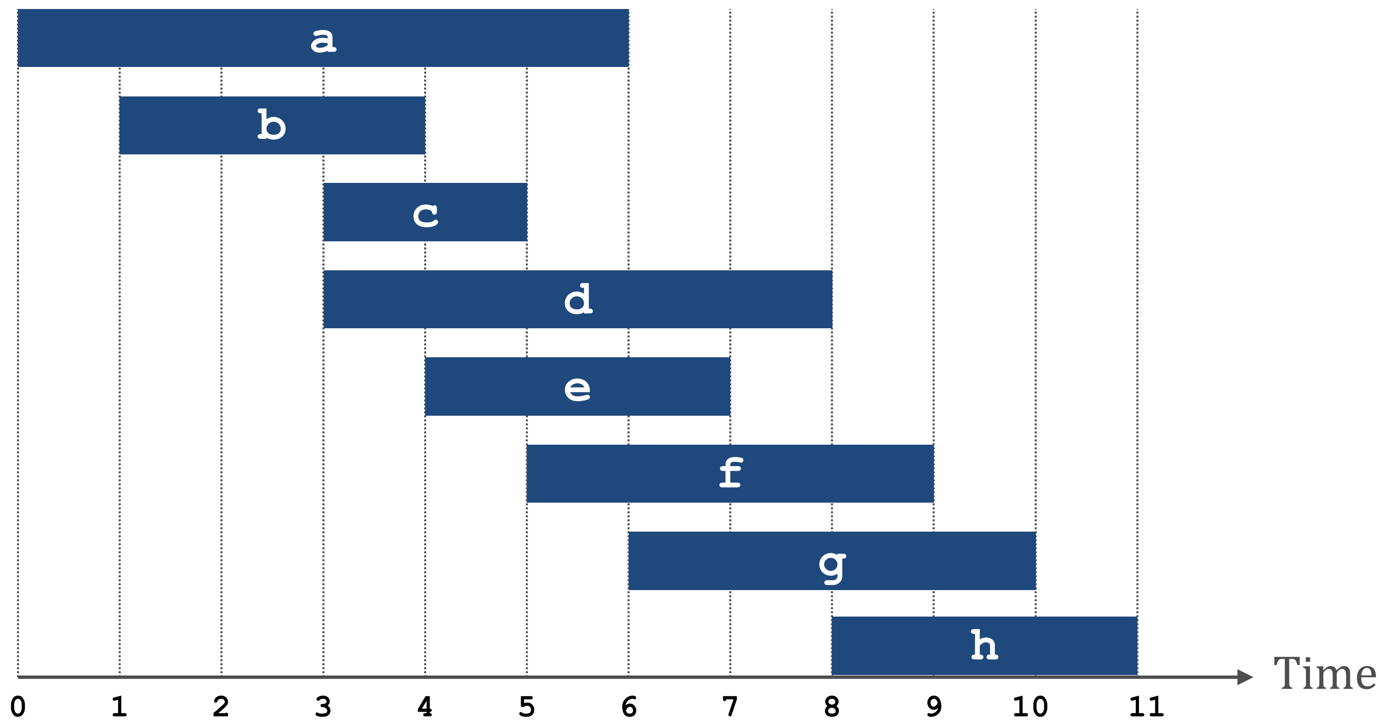
- The algorithmic tools so far:
  - Greedy and D&C
- Great tools
- But useful in very specific types of problems
  - E.g. the problems we talked in the D&C module have obvious poly-time solutions, and the smart D&C solution was faster
  - But we didn't see how to break "exponential barriers"
- Today:
  - Sledgehammer #1: Dynamic programming
- In the future:
  - Sledgehammer #2: Linear programming

DYNAMIC PROGRAMMING

- Greedy:
  - Build a solution incrementally
- D&C:
  - Break problem into smaller sub-problems of the same instance, solve recursively and combine
- Dynamic programming:
  - Break up problem into series of overlapping sub-problems and build up solutions to larger and larger sub-problems

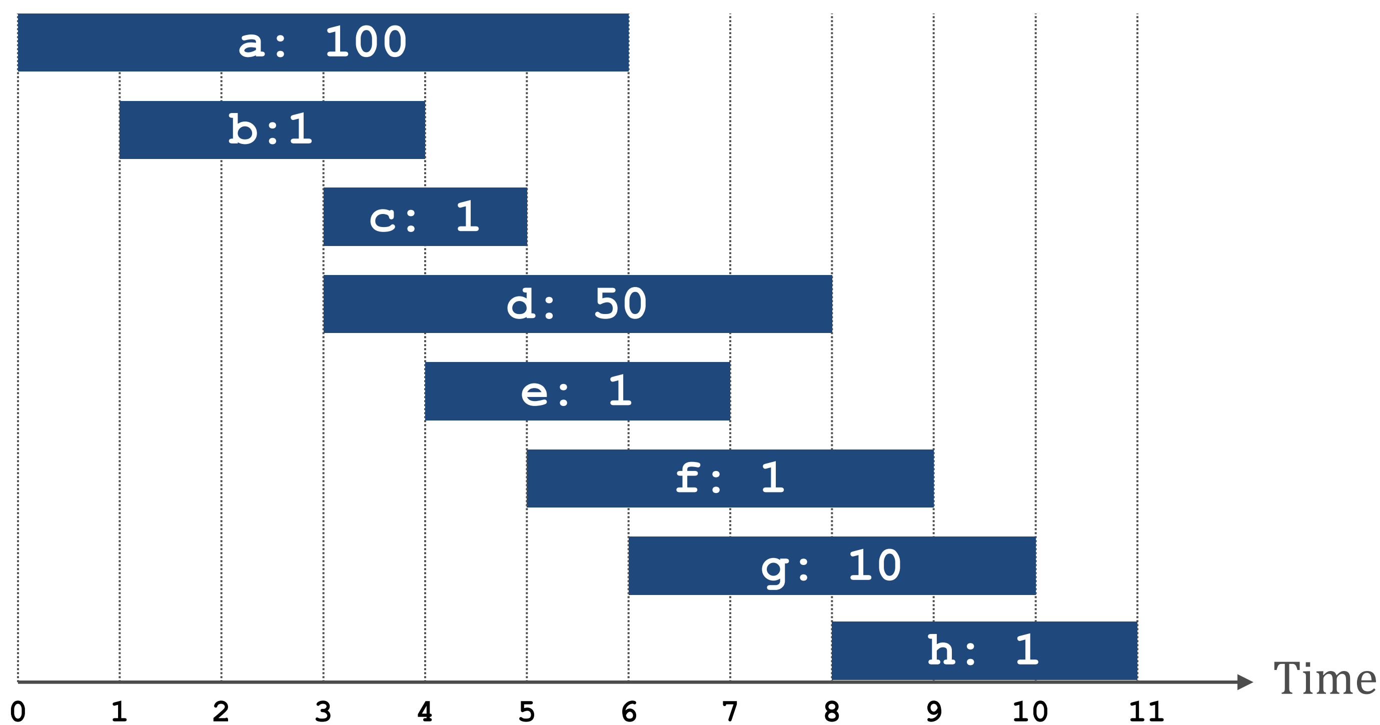
INTERVAL SCHEDULING

- There's an incoming set of jobs $\{1, \ldots, n\}$
- The ith job corresponds to an interval $[s_i, t_i]$
- Two jobs are compatible if they don't overlap
- Goal: find maximum subset of compatible jobs



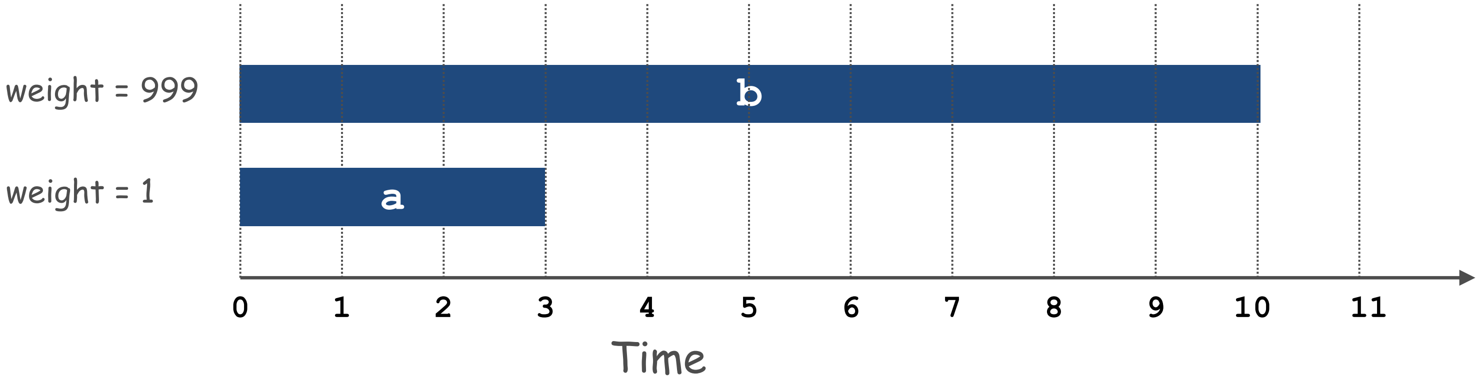
WEIGHTED INTERVAL SCHEDULING

- There's an incoming set of jobs $\{1, \ldots, n\}$
- The ith job corresponds to an interval $[s_i, t_i]$ and has weight v_i
- Two jobs are compatible if they don't overlap
- Goal: find maximum weight subset of compatible jobs



WEIGHTED INTERVAL SCHEDULING

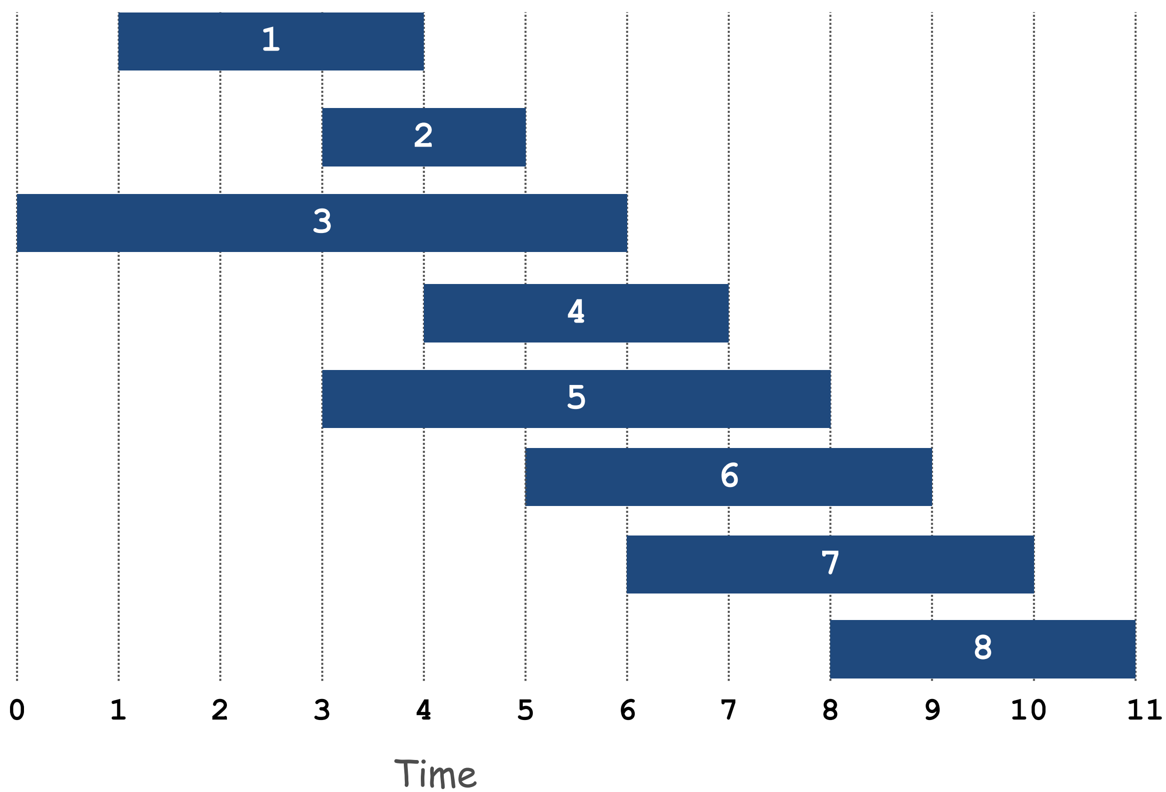
- Observation: Greedy can fail (spectacularly) if arbitrary weights are allowed



WEIGHTED INTERVAL SCHEDULING

- Sort in non decreasing finish time and re-name the jobs so that $f_1 \leq f_2 \leq \cdots \leq f_n$
- For job j let $p(j) =$ largest index $i < j$ such that job i is compatible with job j

- $p(2) = 0$
- $p(4) = 1$
- $p(7) = 3$
- $p(8) = 5$



WEIGHTED INTERVAL SCHEDULING

- $OPT(j) \coloneqq$ value of optimal solution to the problem with requests $1, \dots, j$
  - Typical dynamic programming formulation
- Case 1: $OPT(j)$ doesn't select job j
  - Easy observation: Then it's just the same as $OPT(j-1)$
- Case 2: $OPT(j)$ selects job j
  - Then it can't select all the jobs that are incompatible with j
  - Those are exactly $p(j)+1, \dots, j-1$
  - Therefore, $OPT(j)$ in this case should just be the same as $OPT(p(j))$ plus job j

WEIGHTED INTERVAL SCHEDULING

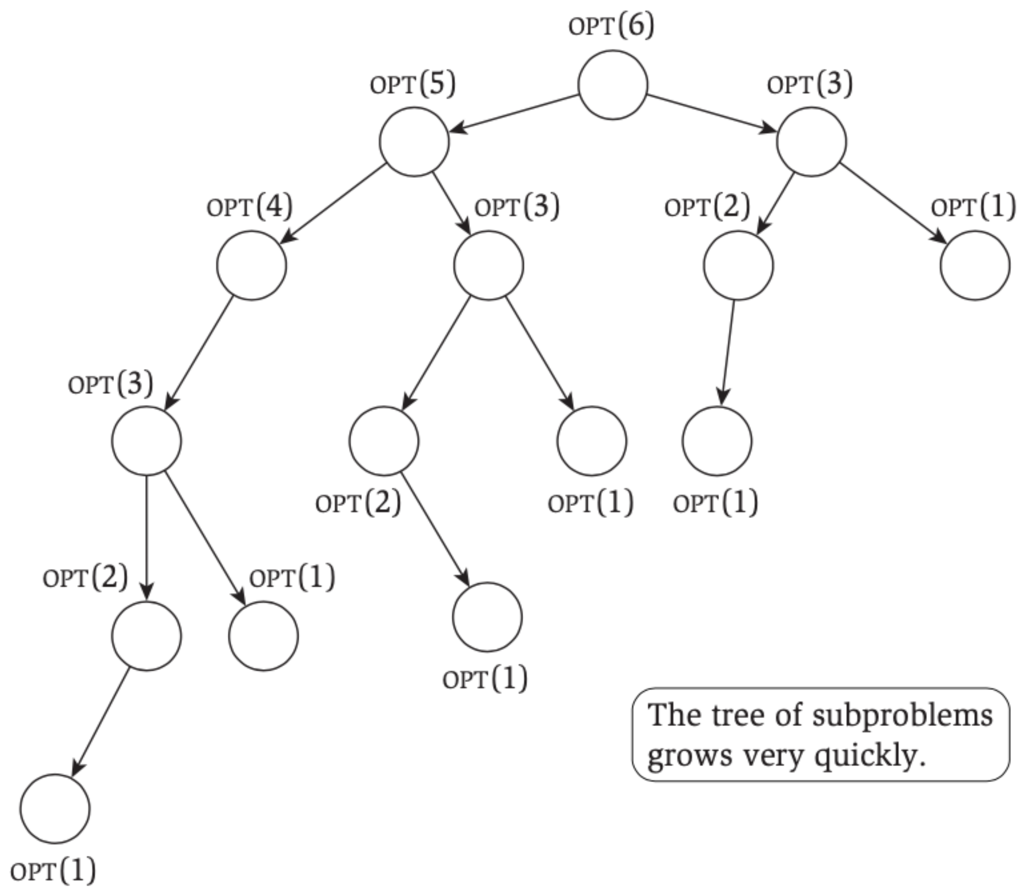
$$OPT(j) = \begin{cases} 0 & \text{if } \ j = 0 \\ \max \left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

WEIGHTED INTERVAL SCHEDULING

1. Sort by finish time and rename to get $f_1 \leq \cdots \leq f_n$: $O(nlog(n))$
2. Compute $p(j)$ for each job j: $O(n^2)$
3. For each job j run compute-$OPT(j)$:
   - If $j = 0$, return 0
   - Return $\max\{OPT\left(p(j)\right) + v_j, OPT(j-1)\}$

- For compute-$OPT(j)$:
  - $T(n) = T(n-1) + T\left(p(n)\right) + O(1)$
  - This is exponential!

WEIGHTED INTERVAL SCHEDULING



RECURSION? NO THANKS!

- Why recurse??
  - Just store the value of $OPT(j)$
- We've already seen this trick with Fibonacci!

3. For all j, $OPT[j] = 0$
4. For all j, run Compute-$OPT(j)$:
   - If $j = 0$ return 0
   - $OPT[j] = \max\{OPT[j-1], OPT[p(j)] + v_j\}$

- Step 4 now takes linear time!

EDIT DISTANCE

- Motivation:
  - A spellchecker finds a misspelled word
  - Wants to look for close by words in the dictionary
  - What's an appropriate notion of distance?

S	–	N	O	W	Y
S	U	N	N	–	Y

Cost: 3

- Minimum number of edits (insertions, deletions and replacements) needed for the two string to be identical
  - "-" above means that you can place anything

EDIT DISTANCE

- Input:
  - Strings $x = x[1, \dots, m]$ and $y = y[1, \dots, n]$
- Output:
  - Minimum edit distance

EDIT DISTANCE

- How about solving it on the prefix?
- $E(i, j) =$ minimum edit distance between $x[1, \ldots, i]$ and $y[1, \ldots, j]$

Figure 6.3 The subproblem $E(7, 5)$.

$\boxed{\text{E X P O N E N}}$ T I A L

$\boxed{\text{P O L Y N}}$ O M I A L

- What's $E(i, j)$ in terms of subproblems we've already solved?
  - E.g. $E(i-1, j)$, $E(i, j-1)$, $E(i-1, j-1)$, and so on?

EDIT DISTANCE

- In the alignment of $x[i]$ and y[j], one of three things can happen
  1. We have $x[i]$ and “ $-$ ”
  2. We have “ $-$ ” and $y[j]$
  3. $x[i]$ aligned with $y[j]$

EDIT DISTANCE

- In the alignment of $x[i]$ and y$[j]$, one of three things can happen
  1. We have $x[i]$ and " $-$ "
  2. We have " $-$ " and $y[j]$
  3. $x[i]$ aligned with $y[j]$
- Case 1:
  - The remainder problem aligns $x[1, \ldots, i-1]$ and $y[1, \ldots, j]$
  - That's just $E(i-1, j)$

EDIT DISTANCE

- In the alignment of $x[i]$ and y[j], one of three things can happen
  1. We have $x[i]$ and " $-$ "
  2. We have " $-$ " and $y[j]$
  3. $x[i]$ aligned with $y[j]$
- Case 2:
  - The remainder problem aligns $x[1, \ldots, i]$ and $y[1, \ldots, j-1]$
  - That's just $E(i, j-1)$

EDIT DISTANCE

- In the alignment of $x[i]$ and y[j], one of three things can happen
  1. We have $x[i]$ and “ $-$ ”
  2. We have “ $-$ ” and $y[j]$
  3. $x[i]$ aligned with $y[j]$
- Case 3:
  - The remainder problem aligns x$[1, \ldots, i-1]$ and y$[1, \ldots, j-1]$
  - That’s just $E(i-1, j-1)$
  - Almost!
    - $x[i] = y[j]$ and they’re aligned
    - Or, $x[i] \neq y[j]$ and they’re aligned
    - Define $diff(i,j) \coloneqq 0$ if $x[i] = y[j]$ and 1 otherwise

EDIT DISTANCE

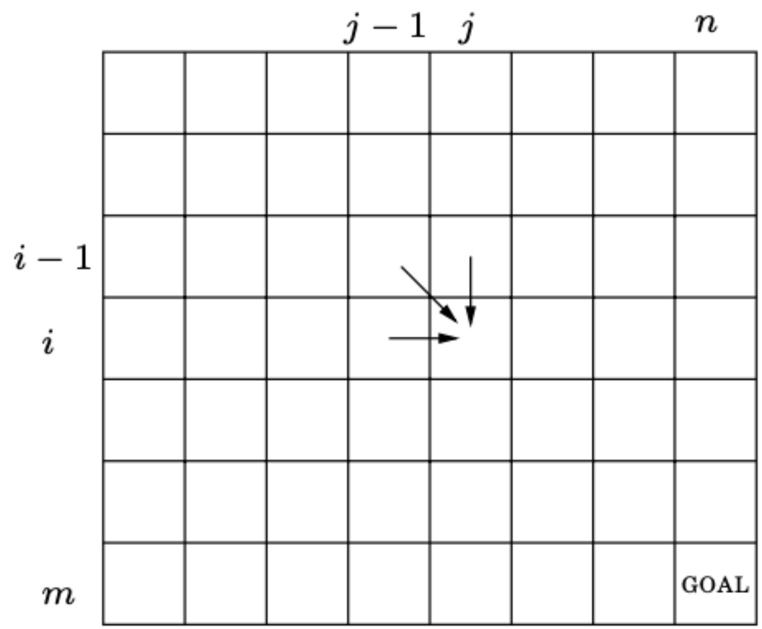
- In the alignment of $x[i]$ and y[j], one of three things can happen
  1. We have $x[i]$ and " $-$ "
  2. We have " $-$ " and $y[j]$
  3. $x[i]$ aligned with $y[j]$
- Overall

$$E(i,j) = min\begin{cases} diff(i,j) + E(i-1,j-1) \\ 1 + E(i-1,j) \\ 1 + E(i,j-1) \end{cases}$$

EDIT DISTANCE

- Example
  - EXPONENTIAL vs POLYNOMIAL
  - $E(4,3)$: EXPO vs POL
    - Either O and “ $-$ ”
    - Either “ $-$ ” and L
    - Or O aligned with L
    - $E(4,3) = \min\{1 + E(3,3), 1 + E(4,2), 1 + E(3,2)\}$

EDIT DISTANCE



EDIT DISTANCE

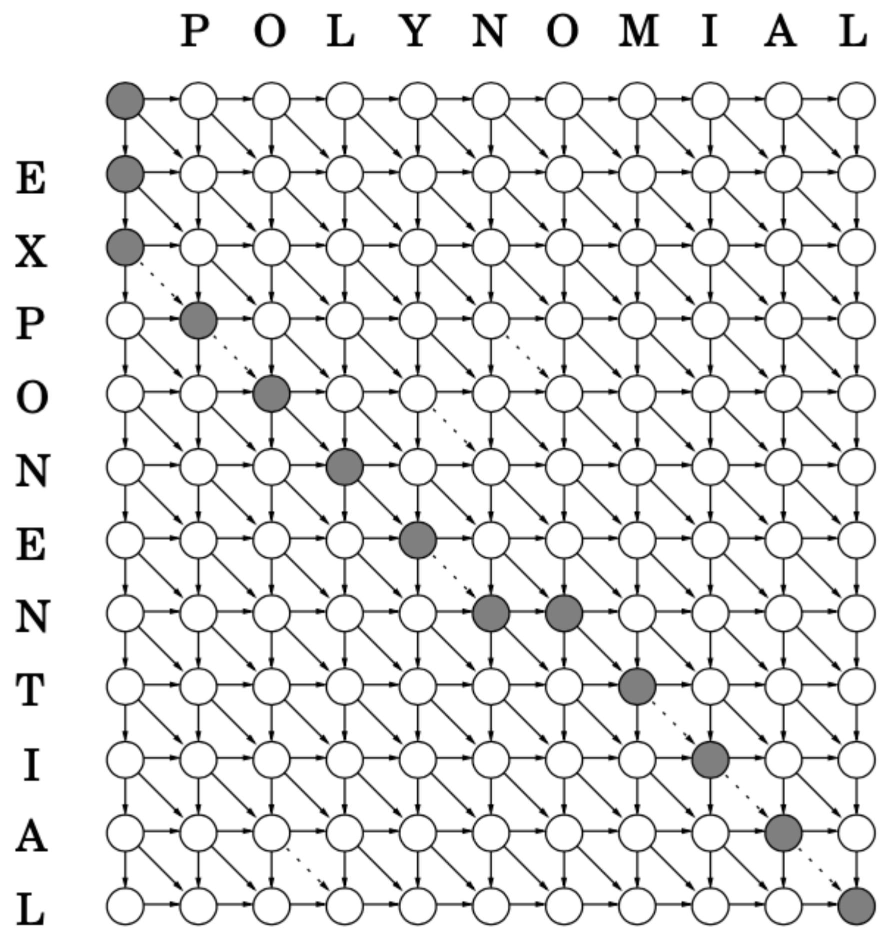
- Where do we start?

- For $i = 0, \ldots, m$: $E(i, 0) = i$
- For $j = 0, \ldots, n$: $\mathrm{E}(0, \mathrm{j}) = \mathrm{j}$
- For $i = 1, \ldots, m$
  - For $j = 1, \ldots, n$:
    - $E(i, j) = \min\{\ldots\}$
- Return $E(m, n)$

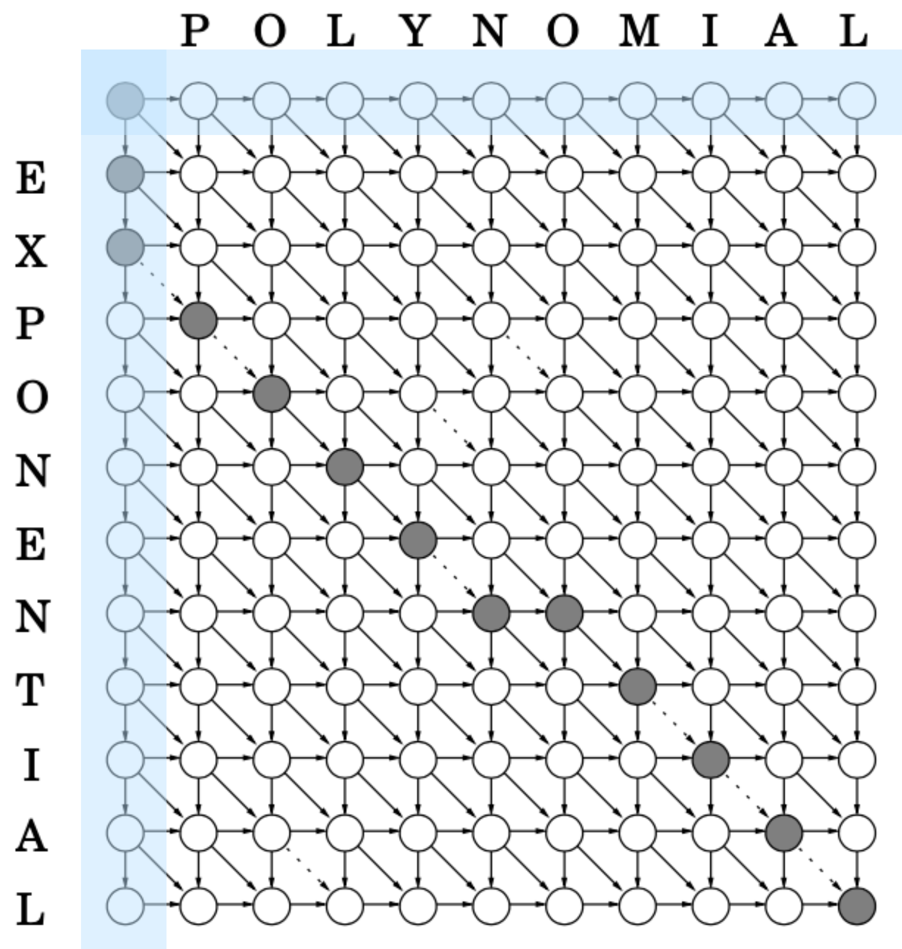
Make sure we have actually solved all subproblems we need!

- Running time: $O(mn)$

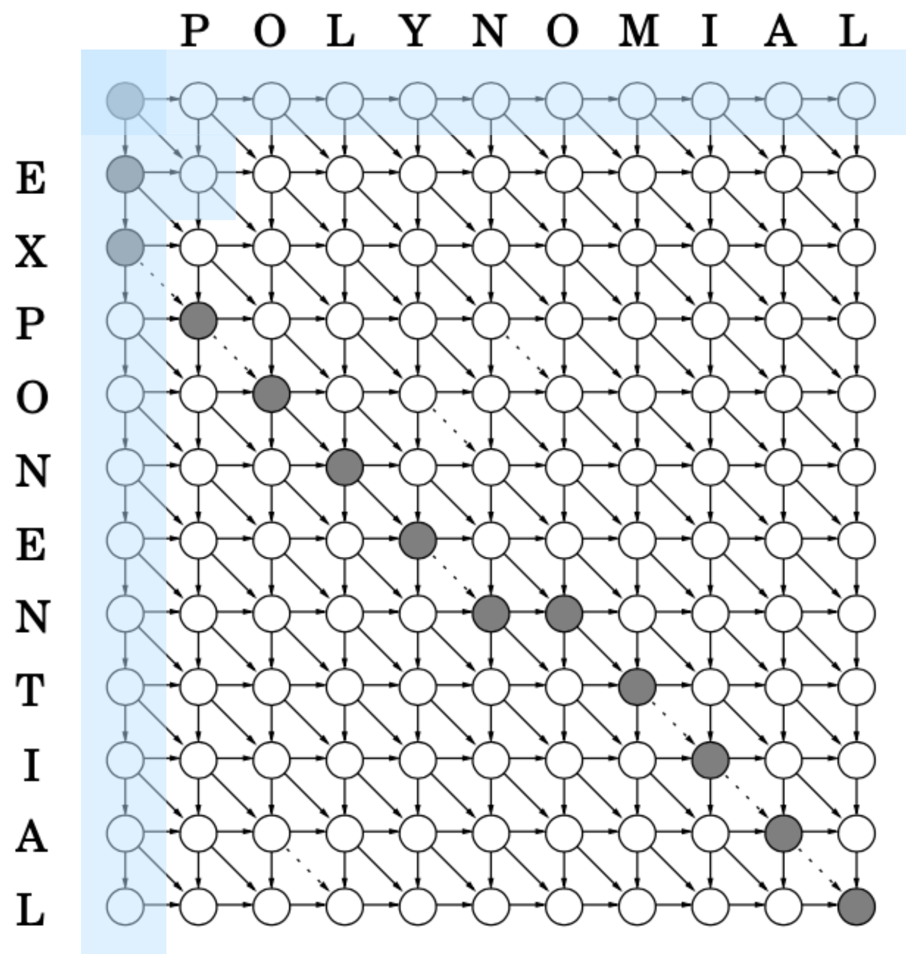
EDIT DISTANCE



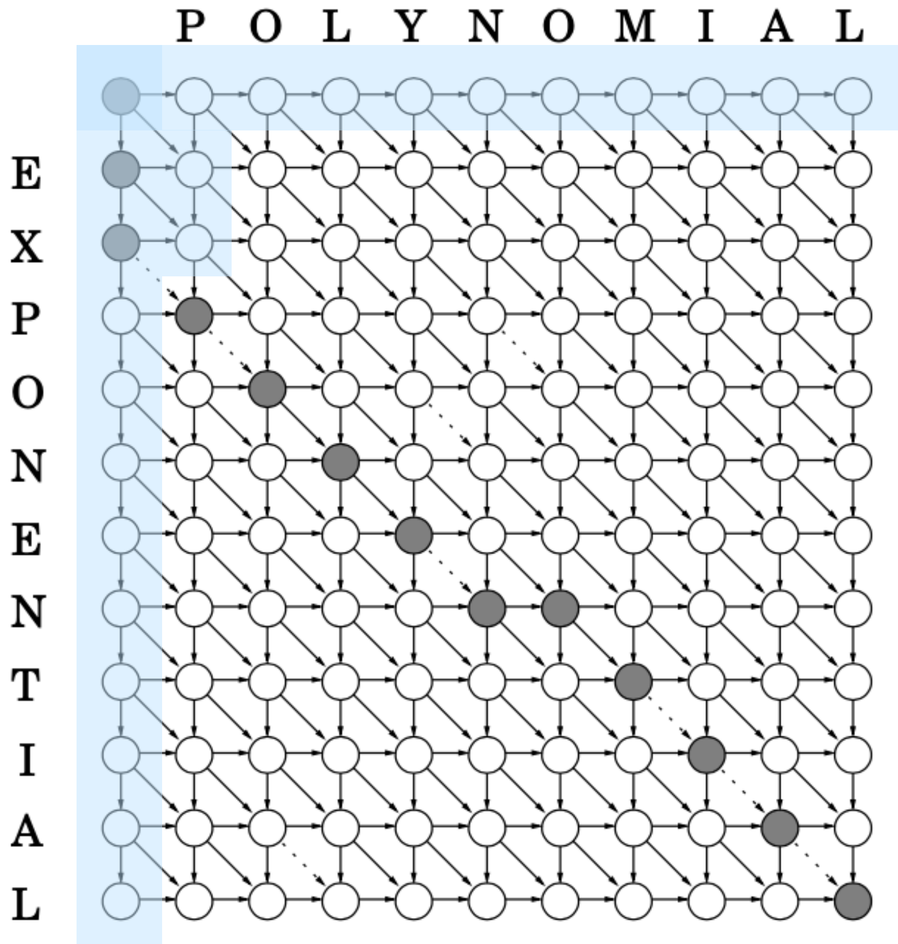
EDIT DISTANCE



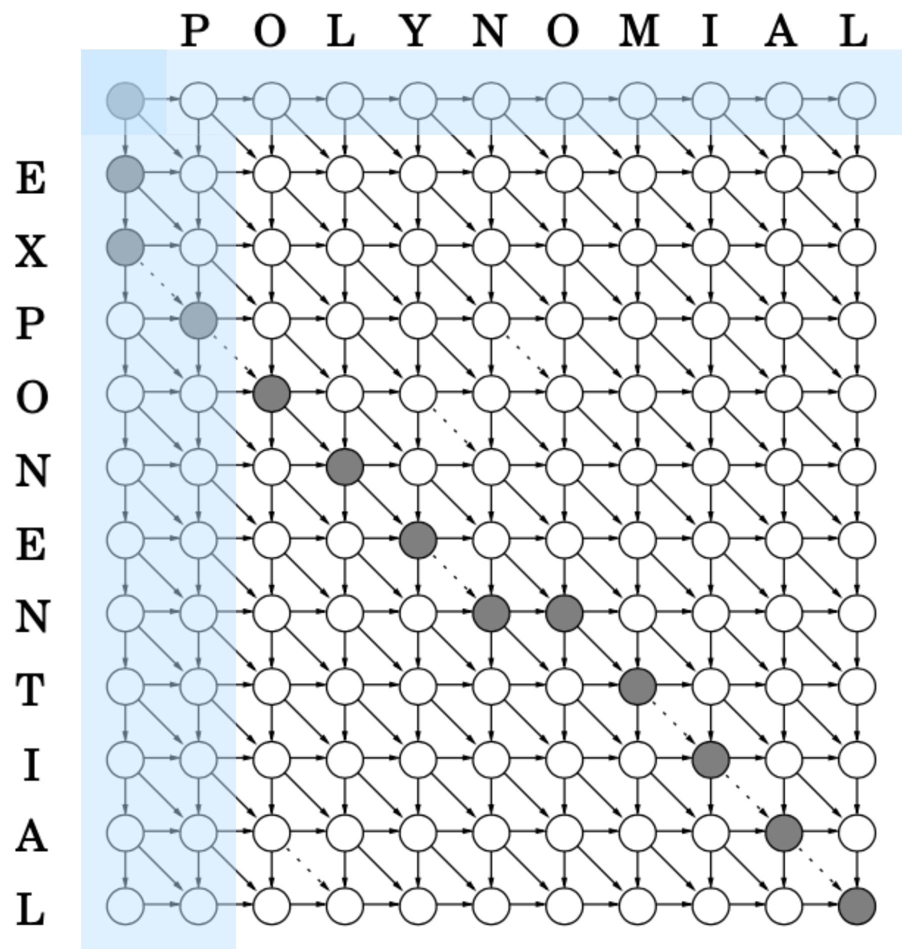
EDIT DISTANCE



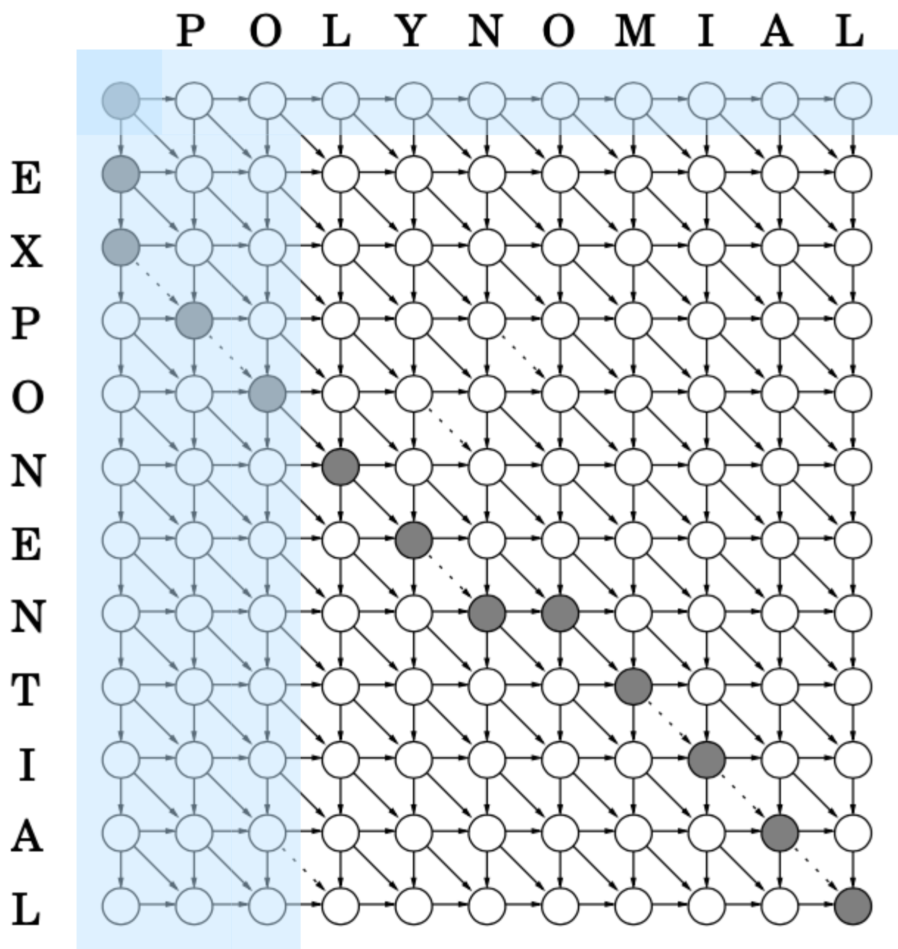
EDIT DISTANCE



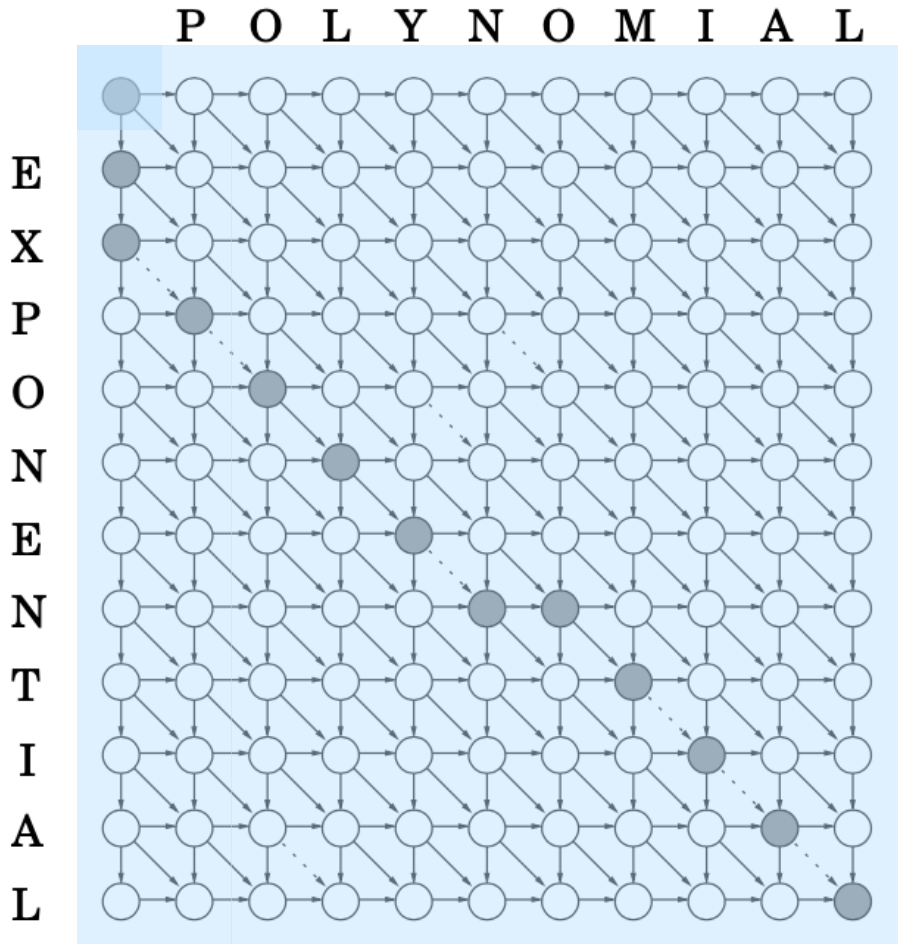
EDIT DISTANCE



EDIT DISTANCE



EDIT DISTANCE



EDIT DISTANCE

- Can you do better?
- Probably not…
  - Backurs and Indyk [2015]
  - If the edit-distance can be computed in time $O(n^{2-\delta})$ for some constant $\delta > 0$, then SAT can be solved in time $2^{n(1-\epsilon)}$ for some constant $\epsilon > 0$. Also known as SETH (Strong Exponential Time Hypothesis)
  - Not as widely believed as $P \neq NP$, but pretty reasonable
    - If you don't believe it, you don't have to worry about finding algorithms for SAT, just beat $O(n^2)$ for edit distance!

KNAPSACK

- During a robbery, a burglar finds much more loot than he had expected and has to decide what to take
- His bag can take total weight of W
- There are n items to pick from
  - Item i has weight w_i and value v_i
- What's the most valuable collection of items that can fit in the bag??

KNAPSACK

- $W = 10$
- Unlimited amount of each item:
  - Item 1 and two of item 4: \$48
- One of each item:
  - Item 1 and item 3: \$46

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

KNAPSACK

- Greedy?
- Clearly picking highest value would be a bad idea (not for this instance, but generally)
- What about sort by v_i/w_i (bang-per-buck)
- Also suboptimal:
  - Will put in items 1 and 2

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

KNAPSACK: WITH REPETITION

- THE question in dynamic programming:
  - What are the subproblems?
- Smaller knapsack (i.e. W)? Fewer items?
- $K(w)$ =maximum value with weight w
- What's the value of $K(w)$?
- If i is included, then it's $v_i + K(w - w_i)$
  - Which i??
  - Try all!
- $K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$

KNAPSACK: WITH REPETITION

Item	Weight	Value
1	6	$30
2	3	$14
3	4	$16
4	2	$9

- $K(0) = K(1) = 0$
- $K(2) = 9$ (only item 4 fits)
- $K(3) = \max\{\, K(0) + v_2, K(1) + v_4 \,\} = 14$
- $K(4) = \max\{\, K(1) + v_2, K(0) + v_3, K(2) + v_4\} = \max\{14{,}16{,}9 + 9\} = 18$
- …
- $K(W)$ is the solution

KNAPSACK: WITHOUT REPETITION

- Previous subproblem is useless
- It's not enough to know $K(w)$
  - We need to know the items used, so that we don't repeat them!
- $K(w, j) =$maximum value with knapsack of size w with items $1, \ldots, j$

KNAPSACK: WITHOUT REPETITION

- Case 1: OPT doesn't select i
  - $K(w, i) = K(w, i-1)$
- Case 2: OPT selects i
  - $K(w, i) = K(w - w_i, i-1) + v_i$
- Pick the best of the two!

$$K(w, i) = \max\{ K(w, i-1), K(w - w_i, i-1) + v_i\}$$

KNAPSACK

- Running time?
- In both versions, we have at least as many subproblems as W
  - Linear?
  - No!!!
- We only need $logW$ bits to represent W
- Exponential running time!!!!
- Can we do better?
- Probably not…

SHORTEST PATHS

- Input:
  - Directed graph $G = (\mathrm{V}, \mathrm{E})$ with weighted (and maybe negative) edges
- Output:
  - The shortest path between all pairs of vertices

SHORTEST PATHS

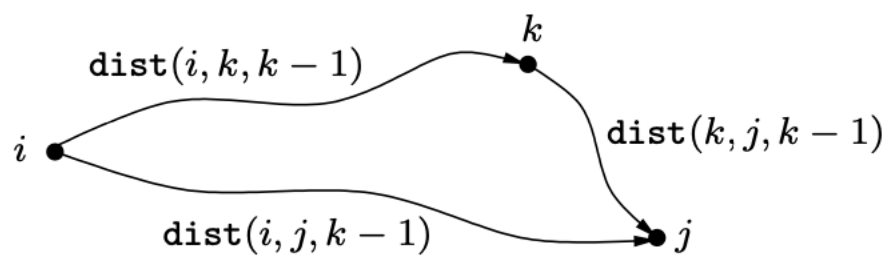
- Bellman-Ford from each vertex s
  - $O(|V|^2|E|)$
- We'll do $O(|V|^3)$
  - The Floyd-Warshall algorithm
- THE question:
  - What's a good subproblem?

SHORTEST PATHS

- Hint: Intermediate nodes
  - What if none are allowed?
  - Easy: shortest path is just an edge, if it exists
  - What if v_1 is the only allowed intermediate node?
  - Shortest (u, v) path is the smaller of (1) the edge $w_{u,v}$ if the edge exists, (2) the edge from u to v_1 plus the edge from v_1 to v
  - Keep growing this

SHORTEST PATHS

- $dist(i, j, k)$: the length of the shortest path from i to j in which the only intermediate nodes allowed are $\{1, \ldots, k\}$
  - $dist(i, j, 0) = w_{i,j}$, or ∞ if no edge exists
- Adding a vertex k
  - Need to re-examine all pairs i, j
  - Easy: either you use k or you don't



SHORTEST PATHS

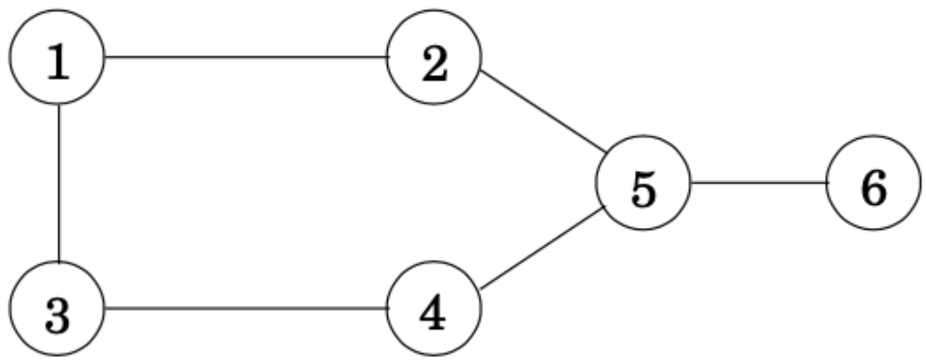
- Note: we are assuming no negative cycles
  - Where??
  - We implicitly assumed that the shortest path from i to j goes through k at most once!
  - We can detect these cycles by running Bellman-Ford
- Under the assumption:

$$dist(i,j,k) = min\begin{cases} dist(i,j,k-1) \\ dist(i,k,k-1) + dist(k,j,k-1) \end{cases}$$

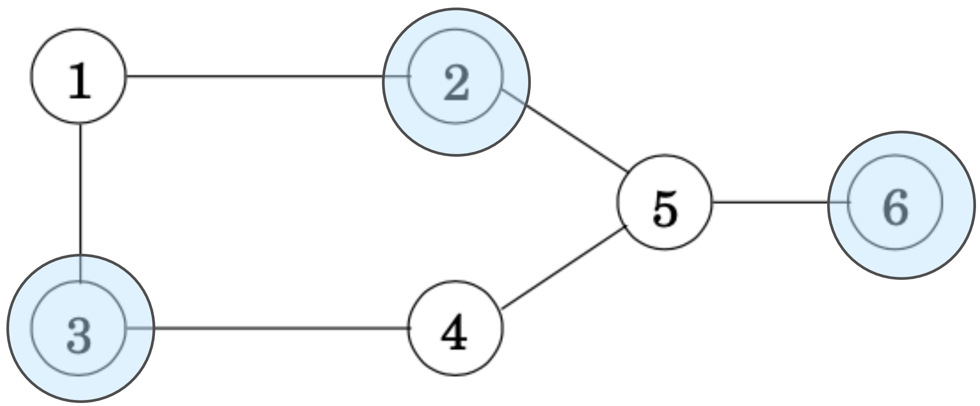
INDEPENDENT SETS IN TREES

- **<u>Definition</u>**: A subset of vertices S is an *independent set* in an undirected graph G if there is no edge (v, u) where both v and u are in S
- Problem:
  - Input: a tree $G = (V, E)$
  - Output: the (size of the) largest independent set

INDEPENDENT SETS IN TREES



INDEPENDENT SETS IN TREES



INDEPENDENT SETS IN TREES

- Independent set in general graphs is difficult
  - Very very very difficult
- Trees are a rare subcase where the problem is tractable
  - Neat dynamic program

INDEPENDENT SETS IN TREES

- THE question:
  - What's the subproblem?
- Root the tree from some node r
- OPT either has r or it doesn't
  - If it does, then it can't contain any of its children
  - If it doesn't then we can recurse in the subtrees
    - Since there are no edges between them we are in good shape: the optimal solutions are independent

INDEPENDENT SETS IN TREES

- $I(r)$ = weight of maximum independent set of subtree hanging from r
- $I(r) = max \begin{cases} 1 + \sum_{grandchildren\ u\ of\ r} I(u) \\ \sum_{children\ u\ of\ r} I(u) \end{cases}$
- Start from leaves and go up the tree
- Done!