國立臺灣大學 National Taiwan University

CHAPTER 6 DYNAMIC PROGRAMMING

Iris Hui-Ru Jiang

Fall 2017

Department of Electrical Engineering
National Taiwan University

Outline

- Content:
  - Weighted interval scheduling: a recursive procedure
  - Principles of dynamic programming (DP)
    - Memoization or iteration over subproblems
  - Example: maze routing
  - Example: Fibonacci sequence
  - Subset sums and Knapsacks: adding a variable
  - Shortest paths in a graph
  - Example: traveling salesman problem
- Reading:
  - Chapter 6

Recap Divide-and-Conquer (D&C)

- Divide and conquer:
  - (Divide) Break down a problem into two or more sub-problems of the same (or related) type
  - (Conquer) Recursively solve each sub-problems and solve them directly if simple enough
  - (Combine) Combine these solutions to the sub-problems to give a solution to the original problem
- Correctness: proved by mathematical induction
- Complexity: determined by solving recurrence relations

Dynamic Programming (DP)

- Dynamic “programming” came from the term “mathematical programming”
  - Typically on optimization problems (a problem with an objective)
  - Inventor: Richard E. Bellman, 1953
- Basic idea: One implicitly explores the space of all possible solutions by
  - Carefully decomposing things into a series of subproblems
  - Building up correct solutions to larger and larger subproblems
- Can you smell the D&C flavor? However, DP is another story!
  - DP does not exam all possible solutions explicitly
  - Be aware of the condition to apply DP!!

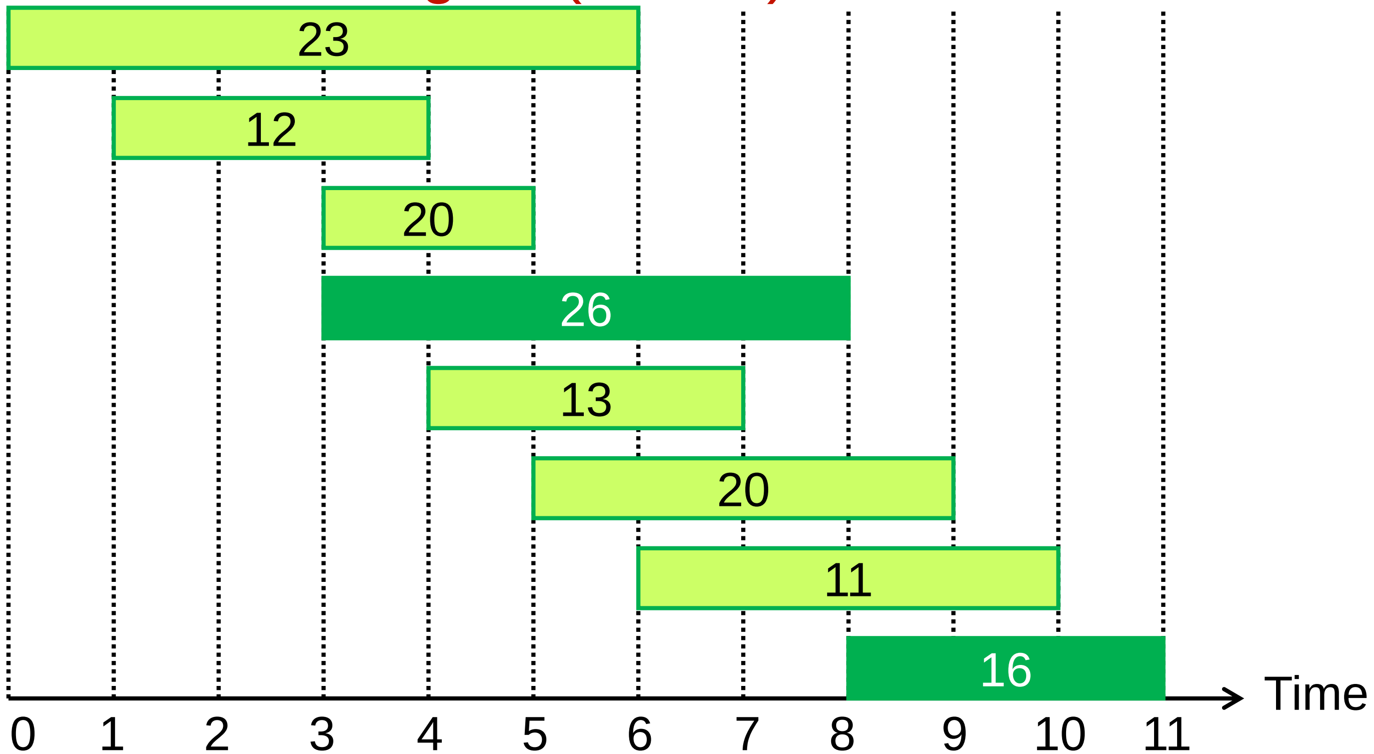
http://en.wikipedia.org/wiki/Dynamic_programming

Weighted Interval Scheduling

Thinking in an inductive way

Weighted Interval Scheduling

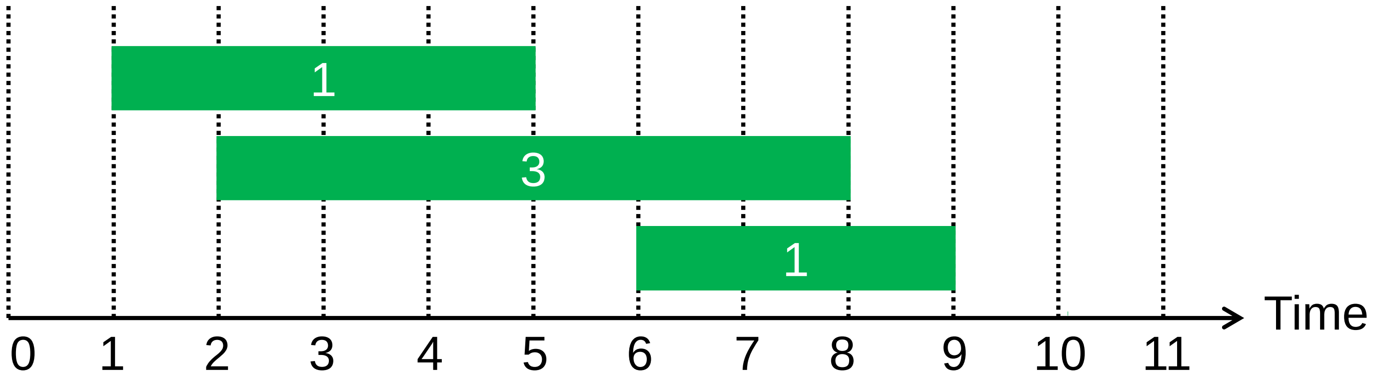
- Given: A set of n intervals with start/finish times, weights
  - Interval i: $[s_i, f_i)$, v_i, $1 \le i \le n$
- Find: A subset S of mutually compatible intervals with maximum total weights (values)



Maximum weighted compatible set {26, 16}

Greedy?

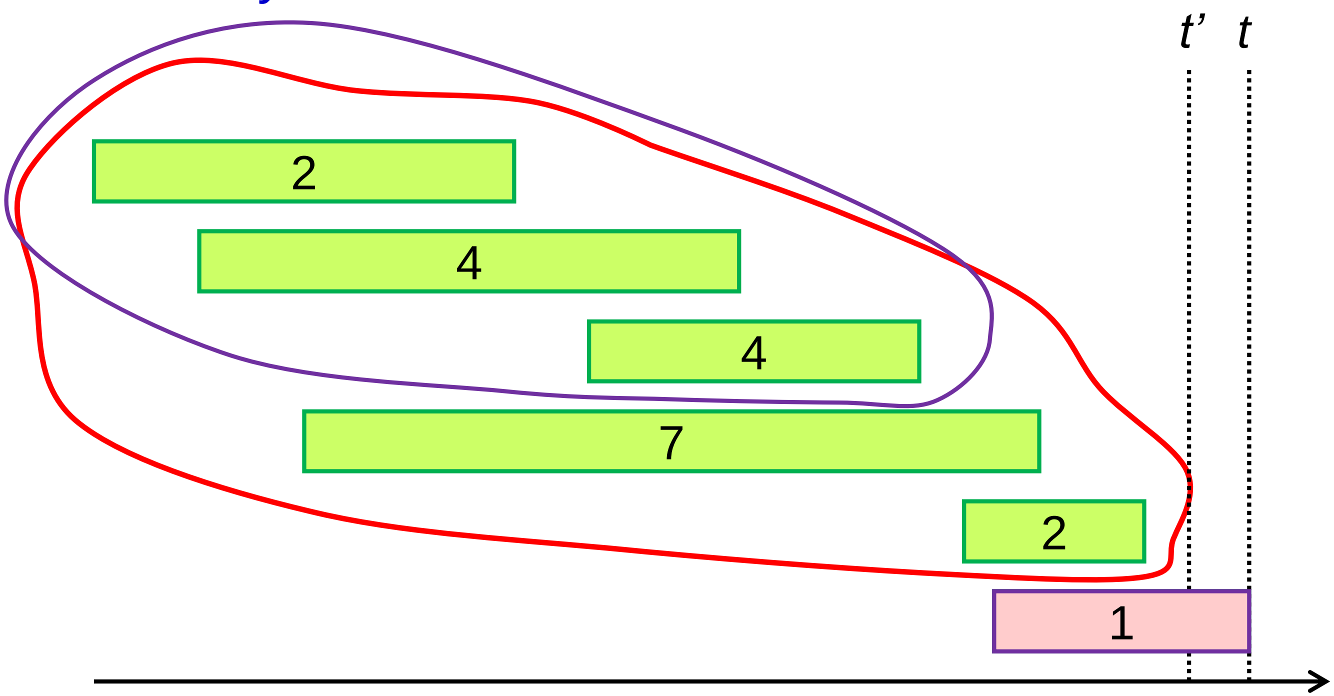
- The greedy algorithm of unit-weighted ($v_i = 1$, $1 \le i \le n$) intervals no longer works!
  - Sort intervals in ascending order of finish times
  - Pick up if compatible; otherwise, discard it



- Q: What if variable weights (values)?

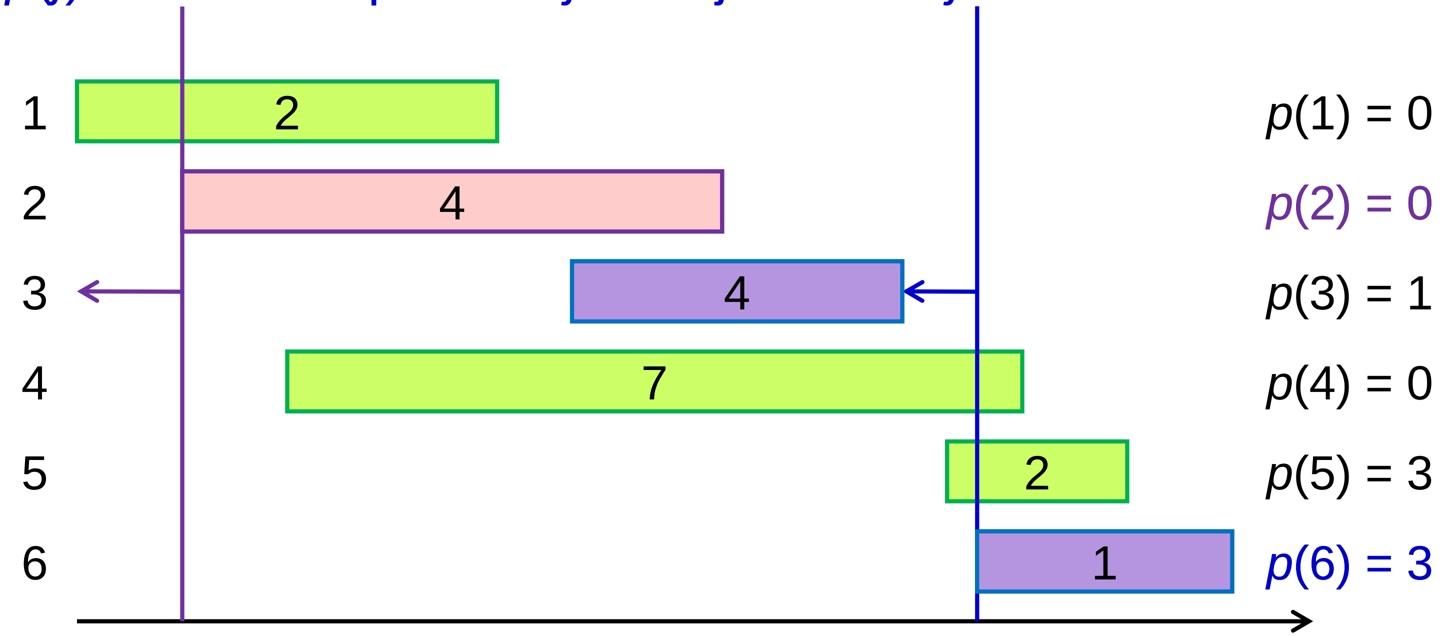
Designing a Recursive Algorithm (1/3)

- In the induction perspective, a recursive algorithm tries to compose the overall solution using the solutions of sub-problems (problems of smaller sizes)
- First attempt: Induction on time?
  - Granularity?



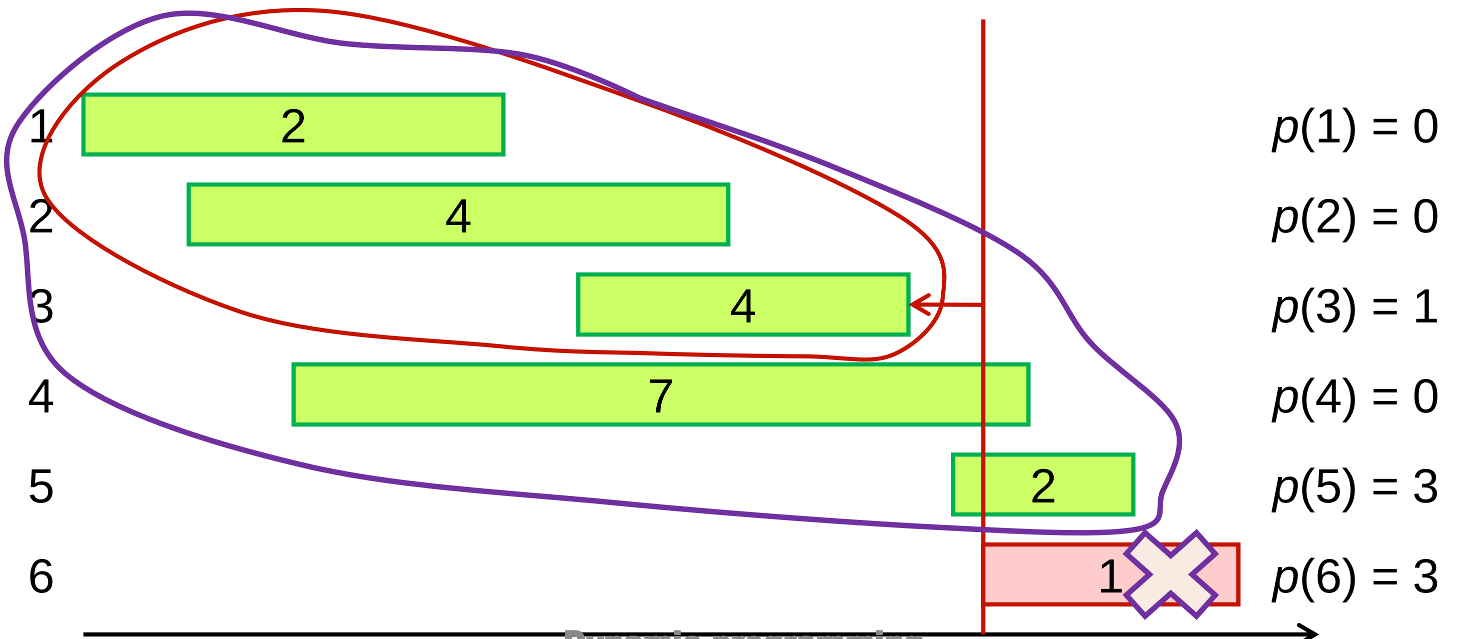
Designing a Recursive Algorithm (2/3)

- Second attempt: Induction on interval index
  - First of all, sort intervals in ascending order of finish times
  - In fact, this is also a trick for DP
- $p(j)$ is the largest index $i < j$ s.t. intervals i and j are disjoint
  - $p(j) = 0$ if no request $i < j$ is disjoint from j



Designing a Recursive Algorithm (3/3)

- O_j = the optimal solution for intervals 1, …, j
- OPT(j) = the value of the optimal solution for intervals 1, …, j
  - e.g., O_6 = ? Include interval 6 or not?
    - $\Rightarrow O_6 = \{6, O_3\}$ or O_5
    - OPT(6) = max{{v_6+OPT(3)}, OPT(5)}
  - $\boxed{\text{OPT}(j) = \max\{\{v_j+\text{OPT}(p(j))\}, \text{OPT}(j-1)\}}$



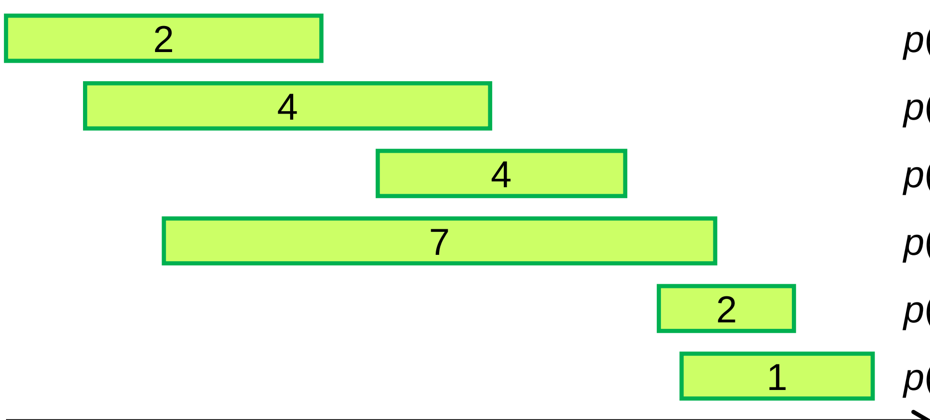
Direct Implementation

$\text{OPT}(j) = \max\{\{v_j + \text{OPT}(p(j))\}, \text{OPT}(j-1)\}$

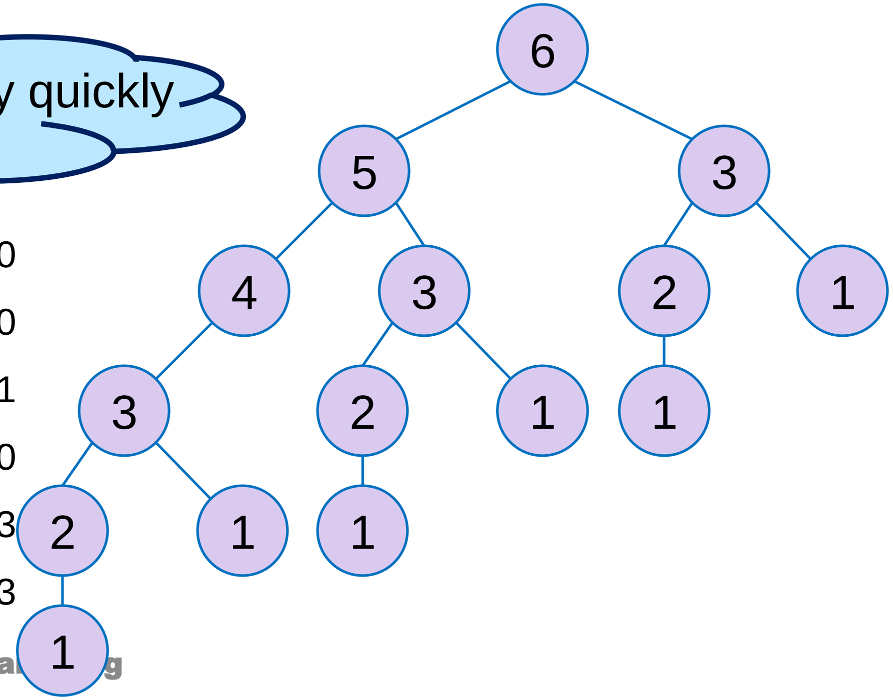
// Preprocessing:
// 1. Sort intervals by finish times: $f_1 \leq f_2 \leq ... \leq f_n$
// 2. Compute $p(1), p(2), \ldots, p(n)$

Compute-Opt(j)
1. **if** (j = 0) **then return** 0
2. **else return** $\max\{\{v_j + \text{Compute-Opt}(p(j))\}, \text{Compute-Opt}(j-1)\}$

The tree of calls widens very quickly due to recursive branching!



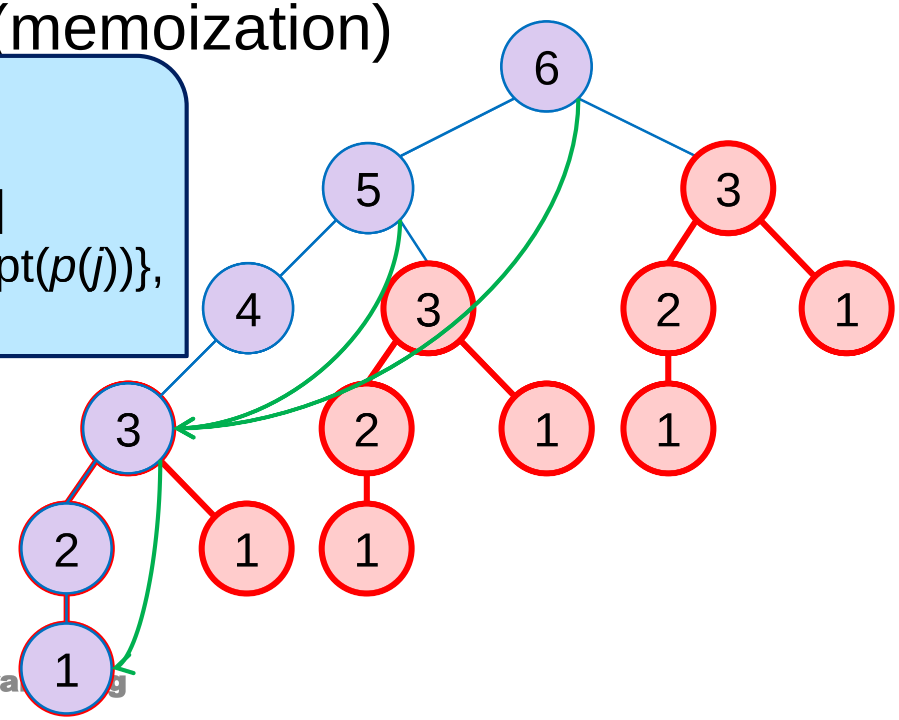
$p(1) = 0$
$p(2) = 0$
$p(3) = 1$
$p(4) = 0$
$p(5) = 3$
$p(6) = 3$



Memoization: Top-Down

- The tree of calls widens very quickly due to recursive branching!
  - e.g., exponential running time when $p(j) = j - 2$ for all j
- Q: What's wrong? A: Redundant calls!
- Q: How to eliminate this redundancy?
- A: Store the value for future! (memoization)

```
M-Compute-Opt(j)
if (j = 0) then return 0
else if (M[j] is not empty) then return M[j]
else return M[j] = max{{v_j+M-Compute-Opt(p(j))},
                        M-Compute-Opt(j-1)}
```



Running time:
$O(n)$

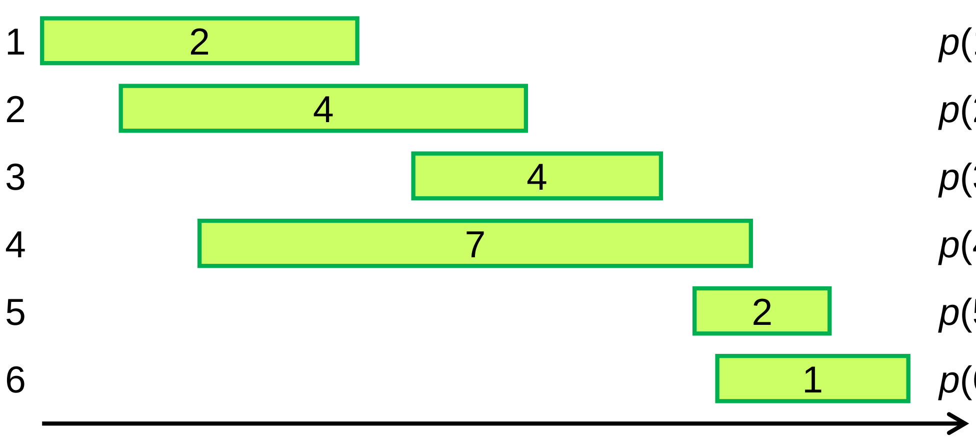
How to report the optimal solution O?

Iteration: Bottom-Up

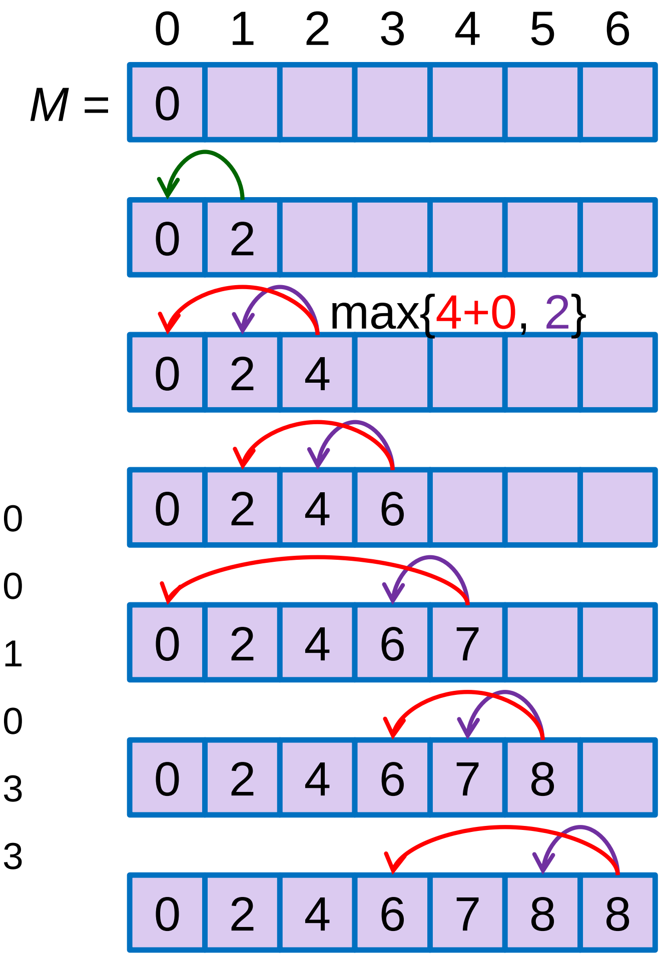
- We can also compute the array $M[j]$ by an iterative algorithm.

I-Compute-Opt
1. $M[0] = 0$
2. **for** $j = 1, 2, .., n$ **do**
3. $\quad M[j] = \max\{v_j + M[p(j)], M[j-1]\}$

Running time:
$O(n)$



$p(1) = 0$
$p(2) = 0$
$p(3) = 1$
$p(4) = 0$
$p(5) = 3$
$p(6) = 3$



Summary: Memoization vs. Iteration

- **Memoization**
- Top-down
- An recursive algorithm
  - Compute only what we need

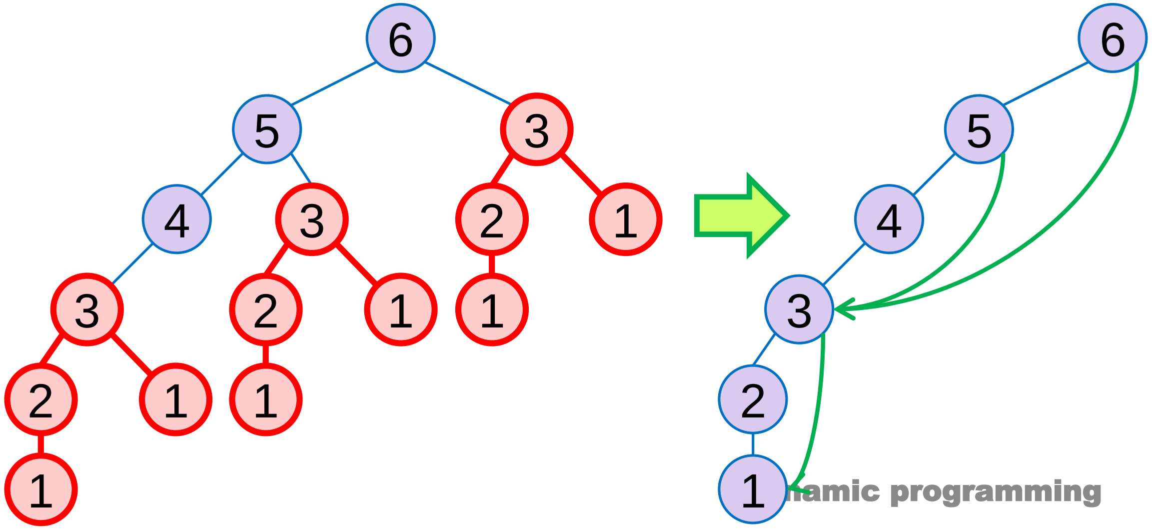
- **Iteration**
- Bottom-up
- An iterative algorithm
  - Construct solutions from the smallest subproblem to the largest one
  - Compute every small piece

Start with the recursive divide-and-conquer algorithm

The running time and memory requirement highly depend on the table size

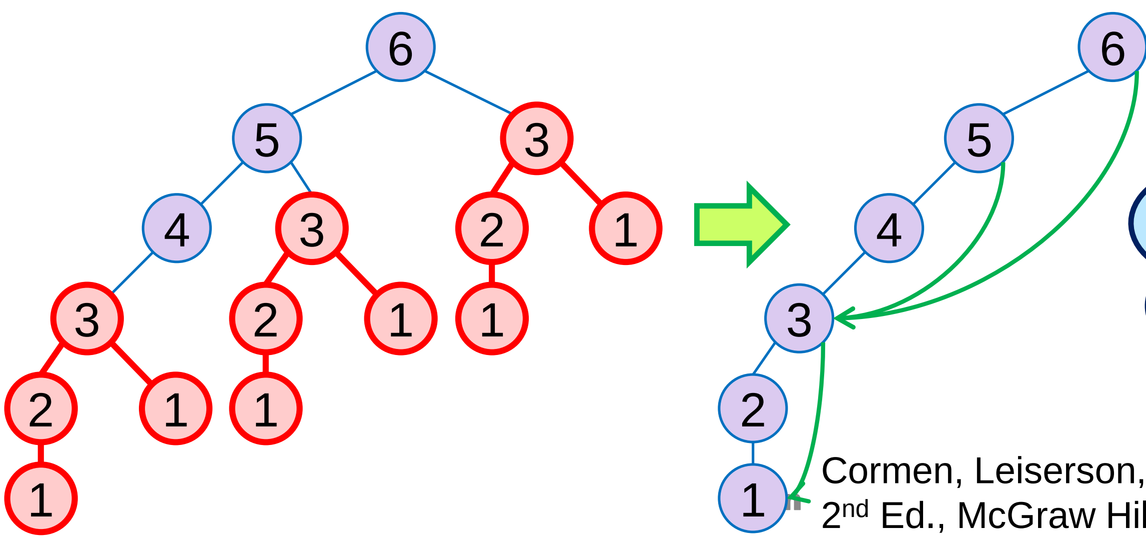
Keys for Dynamic Programming

- DP typically is applied to optimization problems.
- Dynamic programming can be used if the problem satisfies the following properties:
  - There are only a polynomial number of subproblems
  - The solution to the original problem can be easily computed from the solutions to the subproblems
  - There is a natural ordering on subproblems from “smallest” to “largest,” together with an easy-to-compute recurrence



Keys for Dynamic Programming

- DP works best on objects that are linearly ordered and cannot be rearranged
- Elements of DP
  - Optimal substructure: an optimal solution contains within its optimal solutions to subproblems.
  - Overlapping subproblem: a recursive algorithm revisits the same problem over and over again; typically, the total number of distinct subproblems is a polynomial in the input size.

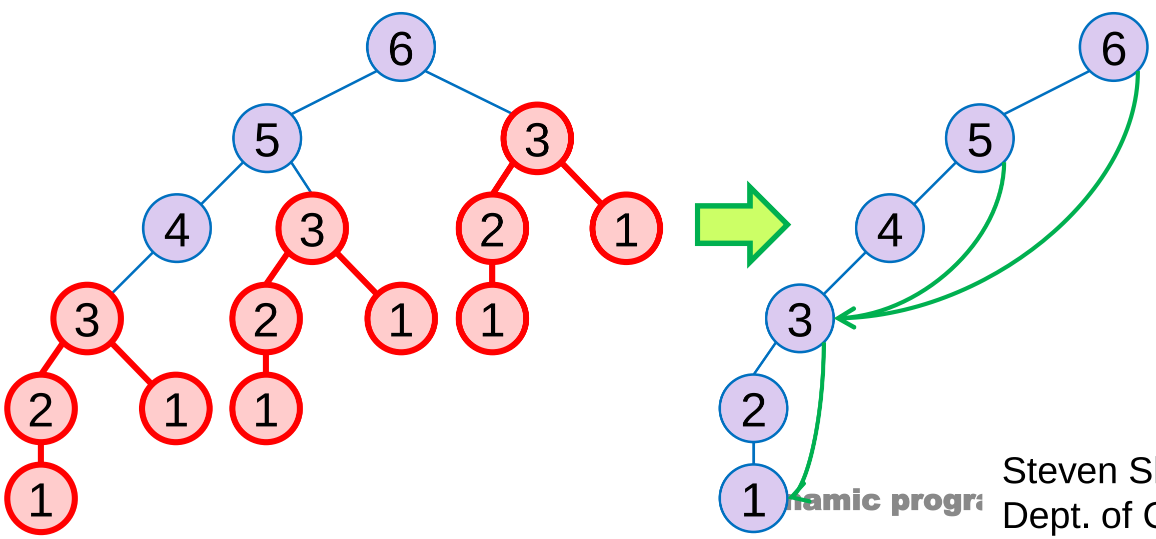


In optimization problems, we are interested in finding a *thing* which maximizes or minimizes some function.

Cormen, Leiserson, Rivest, Stein, *Introduction to Algorithms*, 2nd Ed., McGraw Hill/MIT Press, 2001

Keys for Dynamic Programming

- Standard operation procedure for DP:
  1. Formulate the answer as a recurrence relation or recursive algorithm. (Start with defining subproblems)
  2. Show that the number of different instances of your recurrence is bounded by a polynomial.
  3. Specify an order of evaluation for the recurrence so you always have what you need. (Also check boundary conditions)



Steven Skiena, Analysis of Algorithms lecture notes, Dept. of CS, SUNY Stony Brook

Algorithmic Paradigms

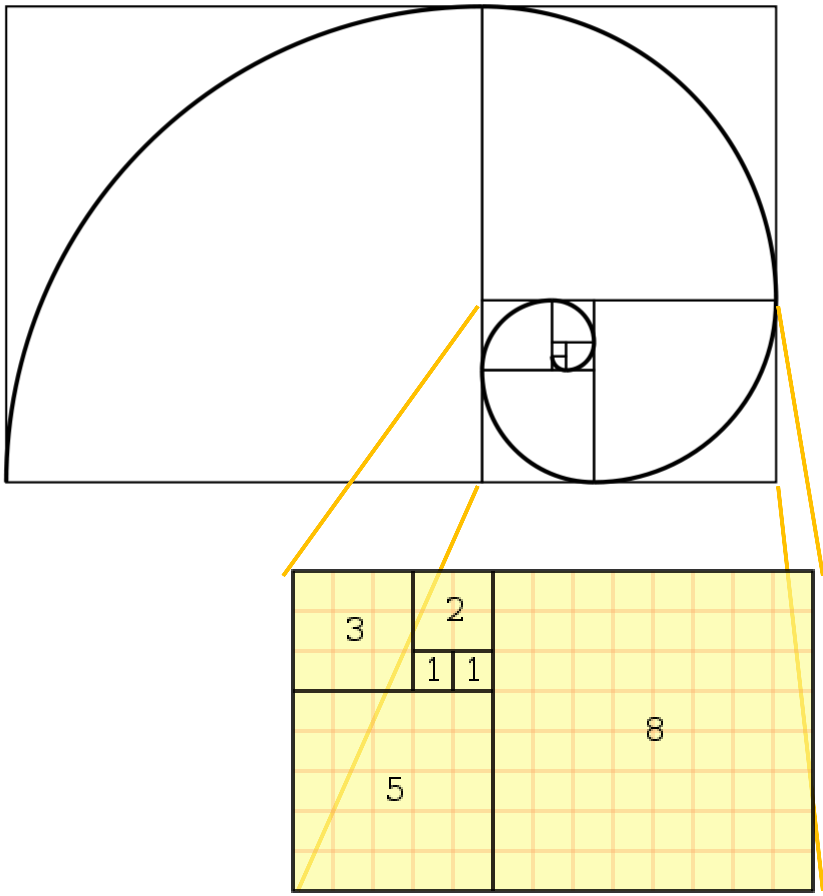
- Brute-force (Exhaustive): Examine the entire set of possible solutions explicitly
  - A victim to show the efficiencies of the following methods
- Greedy: Build up a solution incrementally, myopically optimizing some local criterion.
- Divide-and-conquer: Break up a problem into two or more sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming: Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Appendix: Fibonacci Sequence

Fibonacci Sequence

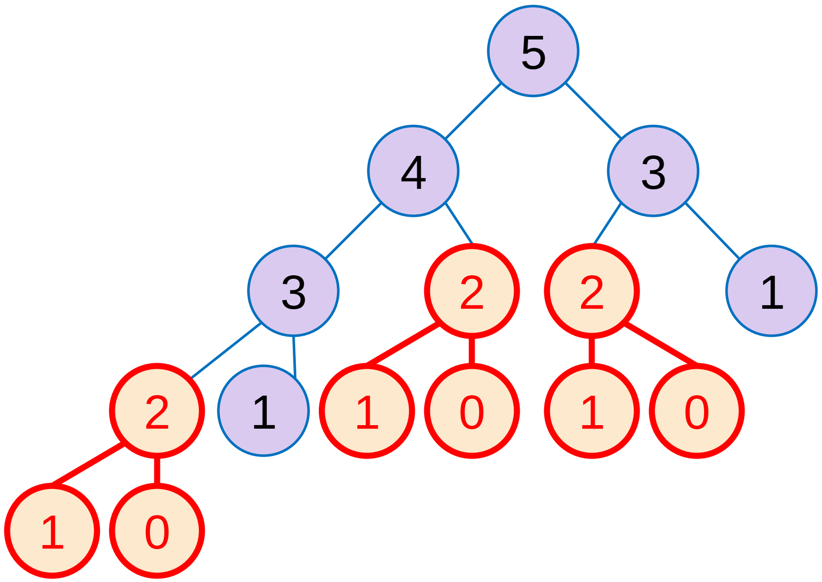
- Recurrence relation: $F_n = F_{n\text{-}1} + F_{n\text{-}2}$, F_0=0, F_1=1
  - e.g., 0, 1, 1, 2, 3, 5, 8, …
- Direct implementation:
  - Recursion!

```
fib(n)
1.  if n ≤ 1 return n
2.  return fib(n − 1) + fib(n − 2)
```



What's Wrong?

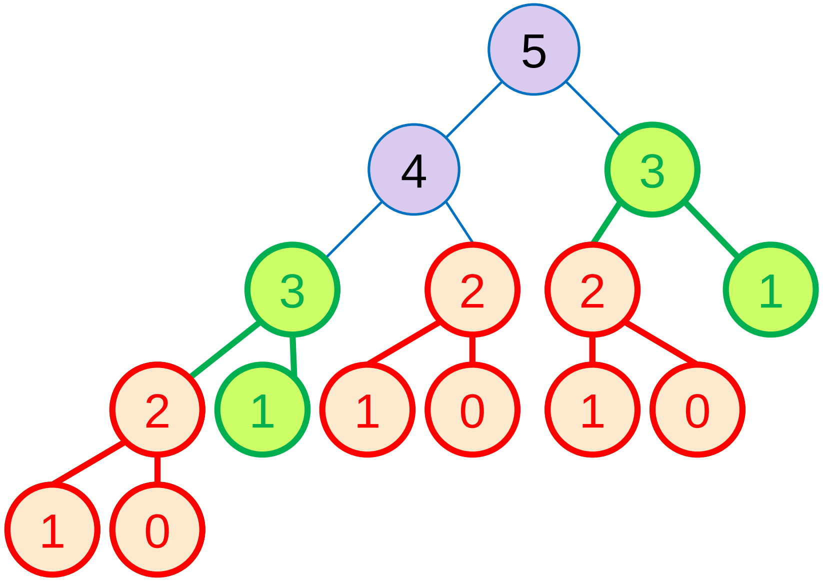
fib(n)
1. **if** $n \le 1$ **return** n
2. **return** fib($n - 1$) + fib($n - 2$)



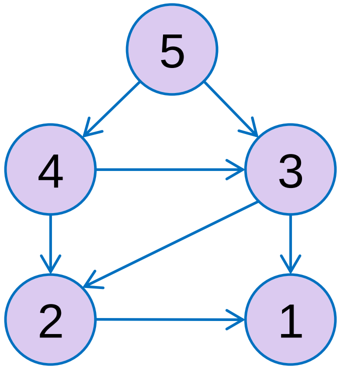
- What if we call fib(5)?
  - fib(5)
  - fib(4) + fib(3)
  - (fib(3) + fib(2)) + (fib(2) + fib(1))
  - ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
  - (((fib(1) + fib(0)) + fib(1))+(fib(1) + fib(0)))+((fib(1) + fib(0))+fib(1))
  - A call tree that calls the function on the same value many different times
    - fib(2) was calculated three times from scratch
    - Impractical for large *n*

Too Many Redundant Calls!

- **Recursion**



- **True dependency**
- How to remove redundancy?
  - Prevent repeated calculation

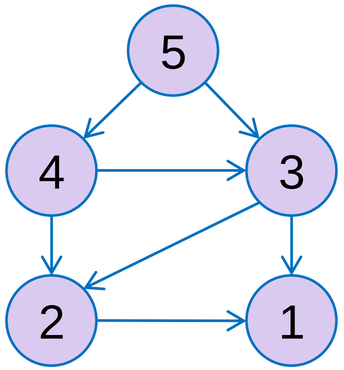


Dynamic Programming -- Memoization

- Store the values in a table
  - Check the table before a recursive call
  - Top-down!
    - The control flow is almost the same as the original one

fib(*n*)
1. Initialize *f*[0..*n*] with -1 // -1: unfilled
2. *f*[0] = 0; *f*[1] = 1
3. fibonacci(*n*, *f*)

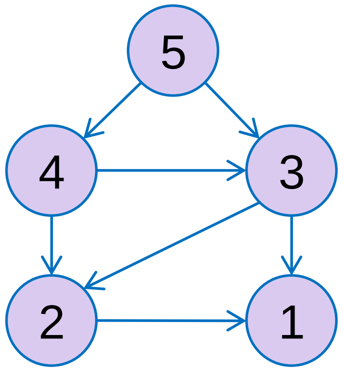
fibonacci(*n*, *f*)
1. **If** *f*[*n*] == -1 **then**
2. *f*[*n*] = fibonacci(*n* − 1, *f*) + fibonacci(*n* − 2, *f*)
3. **return** *f*[*n*] // if *f*[*n*] already exists, directly return



Dynamic Programming -- Bottom-up?

- Store the values in a table
  - Bottom-up
    - Compute the values for small problems first
  - Pretty much like induction

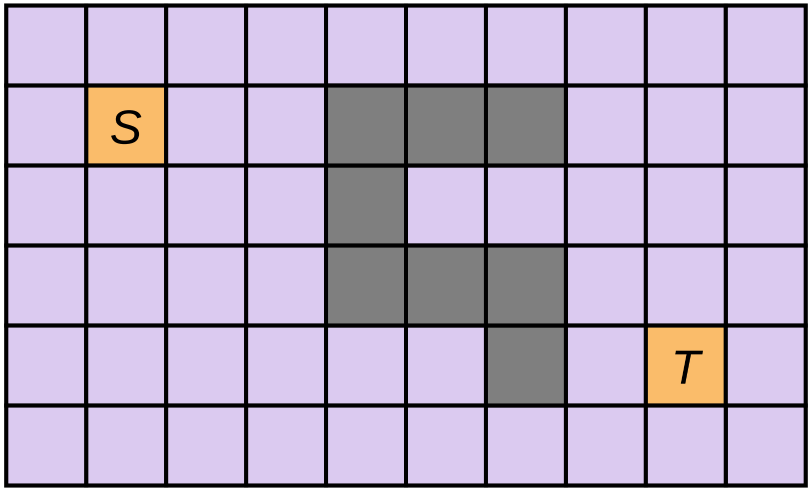
fib(n)
1. initialize $f[1..n]$ with -1 // -1: unfilled
2. $f[0] = 0$; $f[1] = 1$
3. **for** i=2 **to** n **do**
4. $f[i] = f[i-1] + f[i-2]$
5. **return** $f[n]$



Appendix: Maze Routing

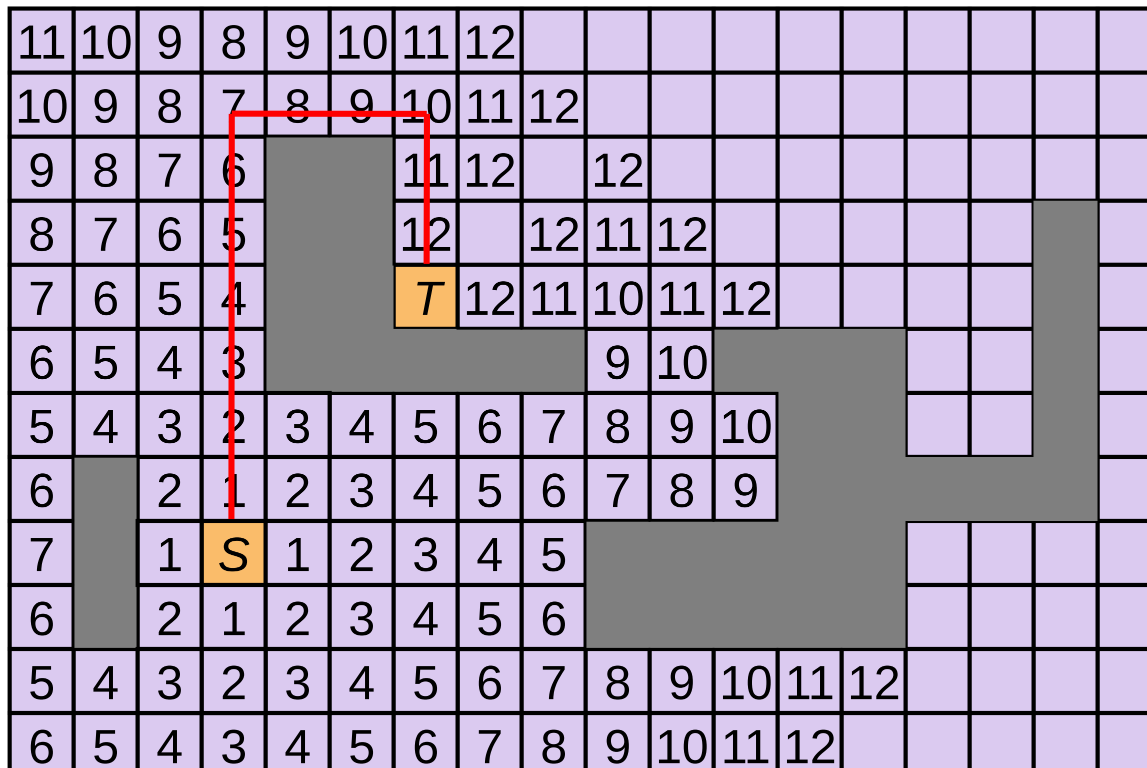
Maze Routing Problem

- Restrictions: Two-pin nets on single-layer rectilinear routing
- Given:
  - A planar rectangular grid graph
  - Two points *S* and *T* on the graph
  - Obstacles modeled as blocked vertices
- Find:
  - The shortest path connecting *S* and *T*
- Applications: Routing in IC design



Lee's Algorithm (1/2)

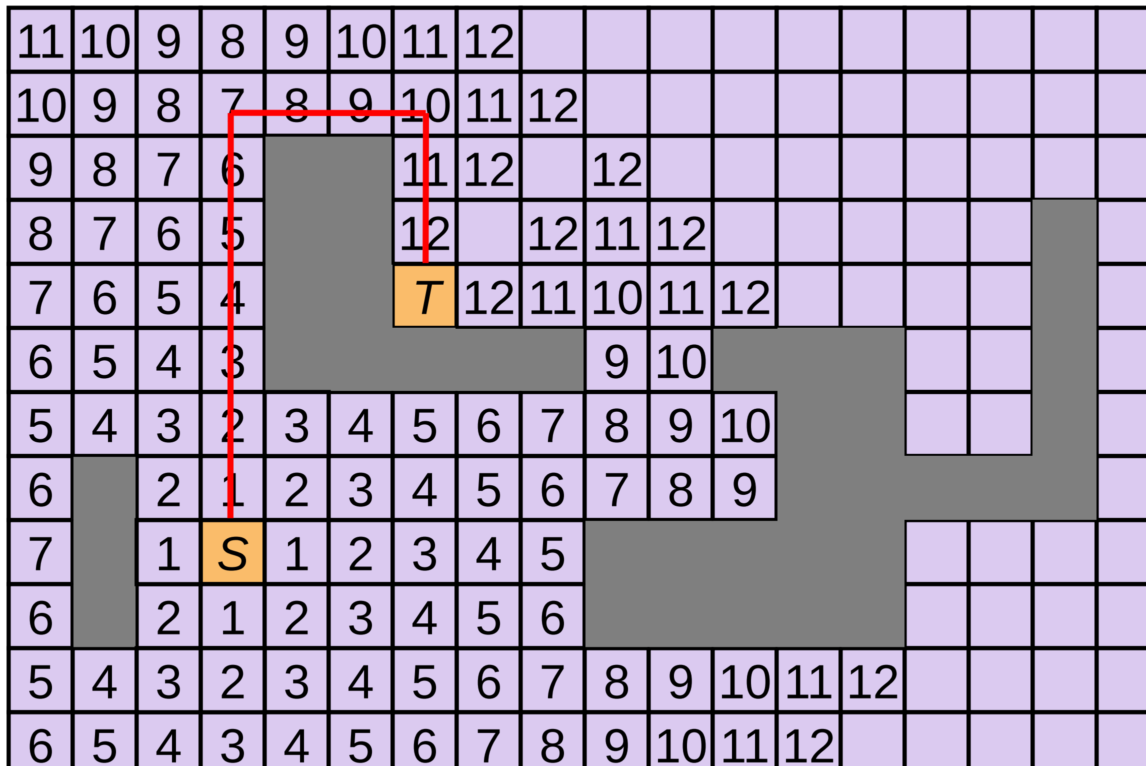
- Idea:
  – Bottom up dynamic programming: Induction on path length
- Procedure:
  1. Wave propagation
  2. Retrace



C. Y. Lee, "An algorithm for path connection and its application," *IRE Trans. Electronic Computer,* vol. EC-10, no. 2, pp. 364-365, 1961.

Lee's Algorithm (2/2)

- Strengths
  - Guarantee to find connection between 2 terminals if it exists
  - Guarantee minimum path
- Weaknesses
  - Large memory for dense layout
  - Slow
- Running time
  - O(MN) for $M \times N$ grid

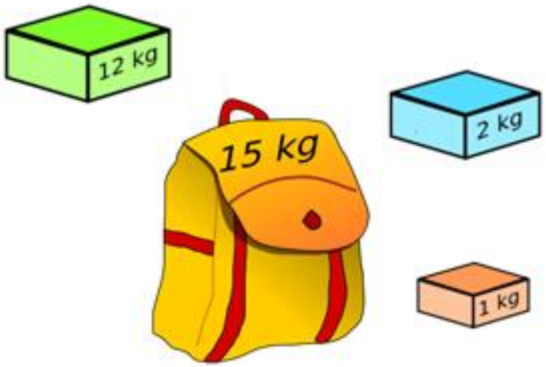


Subset Sums & Knapsacks

Adding a variable

Subset Sum

- Given
  - A set of n items and a knapsack
    - Item i weighs $w_i > 0$.
    - The knapsack has capacity of W.
- Goal:
  - Fill the knapsack so as to maximize total weight.
    - maximize $\Sigma_{i \in S} w_i$
- Greedy $\neq$ optimal
  - Largest w_i first: 7+2+1 = 10
  - Optimal: 5+6 = 11



$W = 11$

Item	Weight
1	1
2	2
3	5
4	6
5	7

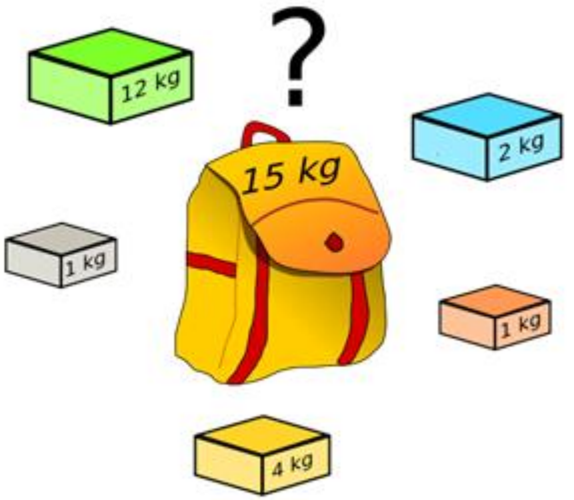
Karp's 21 NP-complete problems:
R. M. Karp, "Reducibility among combinatorial problems". *Complexity of Computer Computations*. pp. 85–103.

Dynamic Programming: False Start

- Optimization problem formulation
  - $\max \Sigma_{i \in S} w_i$ objective function
    s.t. $\Sigma_{i \in S} w_i < W$, $S \subseteq \{1, \ldots, n\}$ constraints
- OPT(i) = the total weight of optimal solution for items 1,…, i
  - $\mathrm{OPT}(i) = \max_S \Sigma_{j \in S} w_j$, $S \subseteq \{1, \ldots, i\}$
- Consider OPT(n), i.e., the total weight of the final solution O
  - Case 1: $n \notin O$ (OPT(n) does not count w_n)
    - $\mathrm{OPT}(n) = \mathrm{OPT}(n\text{-}1)$ (Optimal solution of $\{1, 2, \ldots, n\text{-}1\}$)
  - Case 2: $n \in O$ (OPT(n) counts w_n)
    - $\mathrm{OPT}(n) = w_n + \mathrm{OPT}(n\text{-}1)$

Q: What's wrong?

A: Accept item $n \Rightarrow$ For items $\{1, 2, \ldots, n\text{-}1\}$, we have less available weight, $W - w_n$



Adding a New Variable

- Optimization problem formulation
  - $\max \Sigma_{i \in S} w_i$
    s.t. $\Sigma_{i \in S} w_i < W$, $S \subseteq \{1, \ldots, n\}$
- OPT(i) depends not only on items $\{1, \ldots, i\}$ but also on W
  - (
- Consider OPT(n), i.e., the total weight of final solution O
  - Case 1: $n \notin O$ (OPT(n) does not count w_n)
    -
  - Case 2: $n \in O$ (OPT(n) counts w_n)
    -
- Recurrence relation:
  -

DP: Iteration

$$\text{OPT}(i, w) = \begin{cases} 0 & \text{if } i, w = 0 \\ \text{OPT}(i\text{-}1, w) & \text{if } w_i > w \\ \max\{\text{OPT}(i\text{-}1, w), w_i + \text{OPT}(i\text{-}1, w\text{-}w_i)\} & \text{otherwise} \end{cases}$$

⇩

Subset-sum(n, w_1,…, w_n, W)

1. **for** w = 0, 1, …, W **do**
2. $\quad M[0, w] = 0$
3. **for** i = 0, 1, …, n **do**
4. $\quad M[i, 0] = 0$
5. **for** i = 1, 2, .., n **do**
6. $\quad$ **for** w = 1, 2, .., W **do**
7. $\quad\quad$ **if** ($w_i > w$) **then**
8. $\quad\quad\quad M[i, w] = M[i\text{-}1, w]$
9. $\quad\quad$ **else**
10. $\quad\quad\quad M[i, w] = \max\{M[i\text{-}1, w], w_i + M[i\text{-}1, w\text{-}w_i]\}$

Example

```
Subset-sum(n, w_1,…, w_n, W)
for w = 0, 1, …, W do
    M[0, w] = 0
for i = 0, 1, …, n do
    M[i, 0] = 0
for i = 1, 2, .., n do
    for w = 1, 2, .., W do
        if (w_i > w) then
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max{M[i-1, w], w_i +M[i-1, w-w_i]}
```

Item	Weight
1	1
2	2
3	5
4	6
5	7

$W = 11$

Running time: $O(nW)$

$W + 1$ →

$n + 1$ ↓

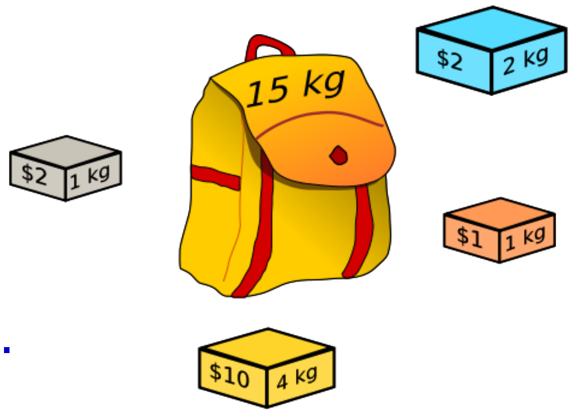
	0	1	2	3	4	5	6	7	8	9	10	11
∅	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	2	3	3	3	3	3	3	3	3	3
{ 1, 2, 3 }	0	1	2	3	3	5	6	7	8	8	8	8
{ 1, 2, 3, 4 }	0	1	2	3	3	5	6	7	8	9	9	11
{ 1, 2, 3, 4, 5 }	0	1	2	3	3	5	6	7	8	9	10	11

Pseudo-Polynomial Running Time

- Running time: $O(nW)$
  – W is not polynomial in input size
  – "Pseudo-polynomial"
  – In fact, the subset sum is a computationally hard problem!
    - r.f. Karp's 21 NP-complete problems:
      – R. M. Karp, "Reducibility among combinatorial problems". *Complexity of Computer Computations*. pp. 85--103.

The Knapsack Problem

- Given
  - A set of n items and a knapsack
  - Item i weighs $w_i > 0$ and has value $v_i > 0$.
  - The knapsack has capacity of W.
- Goal:
  - Fill the knapsack so as to maximize total value.
    - Maximize $\Sigma_{i \in S}\, v_i$
- Optimization problem formulation
  - $\max \Sigma_{i \in S}\, v_i$
    s.t. $\Sigma_{i \in S}\, w_i < W$, $S \subseteq \{1, \ldots, n\}$
- Greedy $\neq$ optimal
  - Largest v_i first: 28+6+1 = 35
  - Optimal: 18+22 = 40



$W = 11$

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Karp's 21 NP-complete problems:
R. M. Karp, "Reducibility among combinatorial problems". *Complexity of Computer Computations*. pp. 85–103.

Recurrence Relation

- We know the recurrence relation for the subset sum problem:

$$\mathrm{OPT}(i, w) = \begin{cases} 0 & \text{if } i, w = 0 \\ \mathrm{OPT}(i-1, w) & \text{if } w_i > w \\ \max\{\mathrm{OPT}(i-1, w),\ w_i + \mathrm{OPT}(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

- Q: How about the Knapsack problem?
- A:

$$\mathrm{OPT}(i, w) = \begin{cases} 0 & \text{if } i, w = 0 \\ \mathrm{OPT}(i-1, w) & \text{if } w_i > w \\ & \text{otherwise} \end{cases}$$

Shortest Path – Bellman-Ford

Richard E. Bellman
Lester R. Ford, Jr.

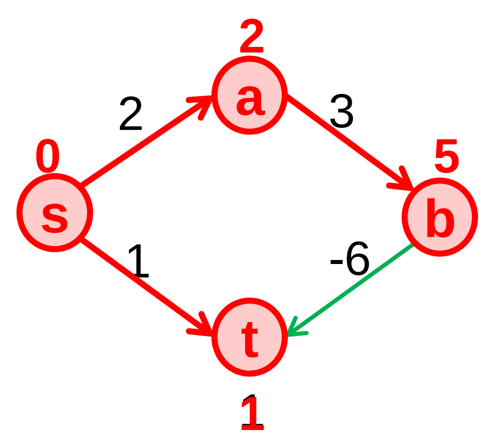
R. E. Bellman 1920—1984
Inventor of DP, 1953

Recap: Dijkstra's Algorithm

- The shortest path problem:
- Given:
  - Directed graph $G = (V, E)$, source s and destination t
    - cost c_{uv} = length of edge $(u, v) \in E$
- Goal:
  - Find the shortest path from s to t
    - Length of path P: $c(P) = \Sigma_{(u,v)\in P}\, c_{uv}$

Dijkstra(G,c)
// S: the set of explored nodes
// $d(u)$: shortest path distance from s to u
initialize $S = \{s\}$, $d(s) = 0$
while $S \neq V$ **do**
    select node $v \notin S$ with at least one edge from S
        $d'(v) = \min_{(u,v):\, u\in S} d(u) + c_{uv}$
    add v to S and define $d(v) = d'(v)$

$c_e \geq 0$

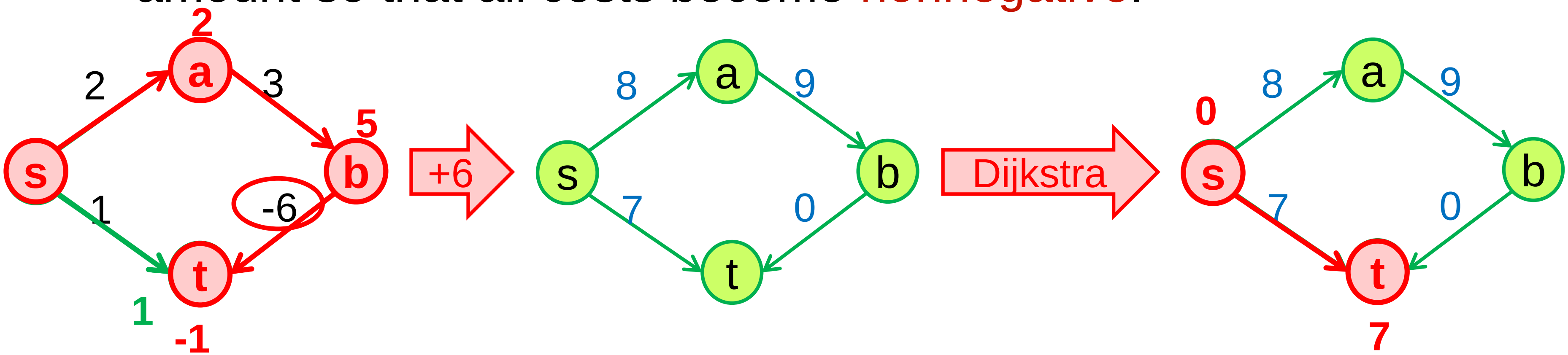


Q: What's wrong with s-a-b-t path?

- Q: What if negative edge costs?

Modifying Dijkstra's Algorithm?

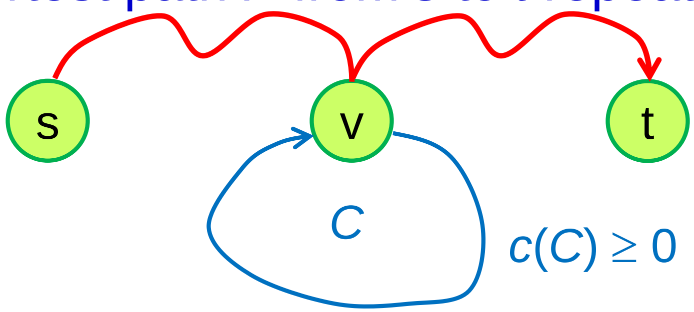
- Observation: A path that starts on a cheap edge may cost more than a path that starts on an expensive edge, but then compensates with subsequent edges of negative cost.
- Reweighting: Increase the costs of all the edges by the same amount so that all costs become nonnegative.



- Q: What's wrong?!
-

Bellman-Ford Algorithm (1/2)

- Induction either on nodes or on edges works!
- If G has no negative cycles, then there is a shortest path from s to t that is simple (i.e., does not repeat nodes), and hence has at most n-1 edges.
- Pf:
  - Suppose the shortest path P from s to t repeat a node v.

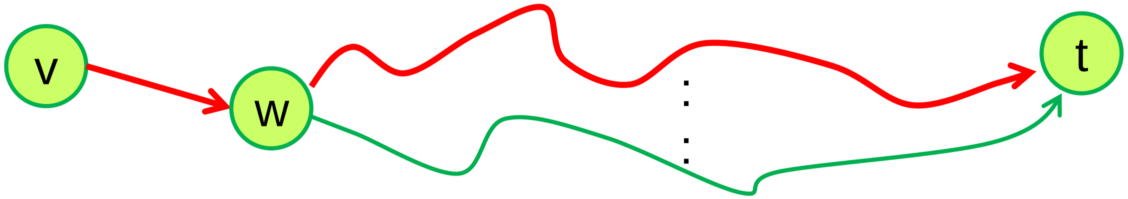


  - Since every cycle has nonnegative cost, we could remove the portion of P between consecutive visits to v resulting in a simple path Q of no greater cost and fewer edges.
    - ■ $c(Q) = c(P) - c(C) \leq c(P)$

Bellman-Ford Algorithm (2/2)

- Induction on edges
- OPT(i, v) = length of shortest v-t path P with at most i edges
  - OPT(n-1, s) = length of shortest s-t path.
  - Case 1: P uses at most i-1 edges.
    - OPT(i, v) = OPT(i-1, v)

  - Case 2: P uses exactly i edges.
    - OPT(i, v) = c_{vw} + OPT(i-1, w)
    - If (v, w) is the first edge, then P uses (v, w) and then selects the shortest w-t path using at most i-1 edges



$$\text{OPT}(i, v) = \begin{cases} 0 & \text{if } i = 0,\ v = t \\ \infty & \text{if } i = 0,\ v \neq t \\ \min\{\text{OPT}(i-1, v),\ \min_{(v,w)\in E}\{c_{vw} + \text{OPT}(i-1, w)\}\} & \text{otherwise} \end{cases}$$

Implementation: Iteration

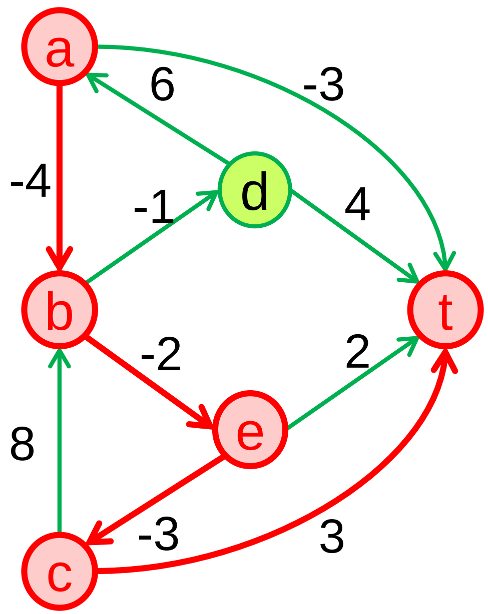
$$\mathrm{OPT}(i, v) = \begin{cases} 0 & \text{if } i = 0,\ v = t \\ \infty & \text{if } i = 0,\ v \neq t \\ \min\{\mathrm{OPT}(i\text{-}1, v), \min_{(v,\, w)\in E}\{c_{vw} + \mathrm{OPT}(i\text{-}1, w)\}\} & \text{otherwise} \end{cases}$$

Bellman-Ford(G, s, t)
// n = # of nodes in G
// M[0.. n-1, V]: table recording optimal solutions of subproblems
1. $M[0, t] = 0$
2. **foreach** $v \in V$-$\{t\}$ **do**
3. $M[0, v] = \infty$
4. **for** i = 1 **to** n-1 **do**
5. **for** $v \in V$ in any order **do**
6. $M[i, v]=\min\{M[i\text{-}1, v], \min_{(v,\, w)\in E}\{c_{vw} + M[i\text{-}1, w]\}\}$

Example

Space: $O(n^2)$
Running time:
1. naïve: $O(n^3)$
2. detailed: $O(nm)$

Bellman-Ford(G, s, t)
// n = # of nodes in G
// M[0.. n-1, V]: table recording optimal solutions of subproblems
1. $M[0, t] = 0$
2. **foreach** $v \in V - \{t\}$ **do**
3. $\quad M[0, v] = \infty$
4. **for** i = 1 **to** n-1 **do**
5. $\quad$ **for** $v \in V$ in any order **do**
6. $\quad\quad M[i, v] = \min\{M[i-1, v], \min_{(v, w) \in E} \{c_{vw} + M[i-1, w]\}\}$



$n \longrightarrow$

n	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

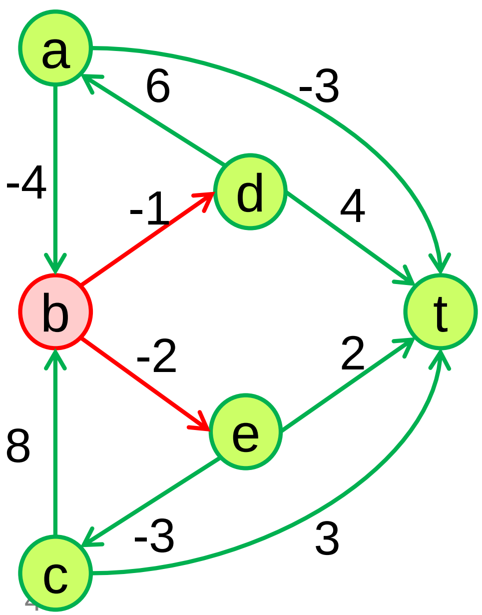
$M[d, 2] = \min\{M[d,1], c_{da} + M[a,1]\}$

Q: How to find the shortest path?
A: Record "successor" for each entry

Running Time

```
Bellman-Ford(G, s, t)
:
4.  for i = 1 to n-1 do
5.      for v∈V in any order do
6.          M[i, v]=min{M[i-1, v], min_(v, w)∈E {c_vw + M[i-1, w]}}
```

- Lines 5-6:
  - Naïve: for each *v*, check *v* and others: $O(n^2)$
  - Detailed: for each *v*, check *v* and its neighbors (out-going edges): $\sum_{v\in V}(\deg_{out}(v)+1) = O(m)$
- Lines 4-6:
  - Naïve: $O(n^3)$
  - Detailed: $O(nm)$



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Space Improvement

Computing Science is –and will always be– concerned with the interplay between mechanized and human symbol manipulation, which usually referred to as "computing" and "programming" respectively.

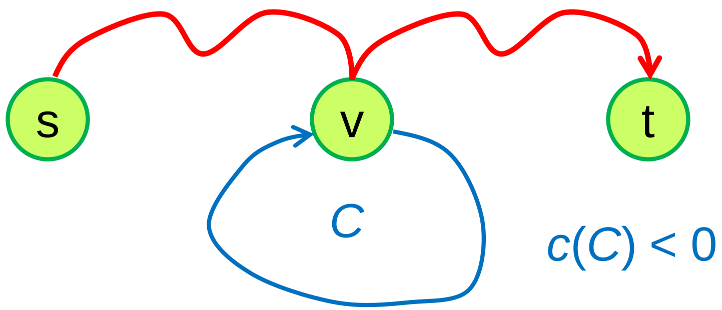
~ E. W. Dijkstra

- Maintain a 1D array instead:
  - $M[v]$ = shortest v-t path length that we have found so far.
  - Iterator i is simply a counter
  - No need to check edges of the form (v, w) unless $M[w]$ changed in previous iteration.
  - In each iteration, for each node v, $M[v]=\min\{M[v], \min_{w \in V}\{c_{vw} + M[w]\}\}$

- Observation: Throughout the algorithm, $M[v]$ is the length of some v-t path, and after i rounds of updates, the value $M[v]$ is no larger than the length of shortest v-t path using at most i edges.

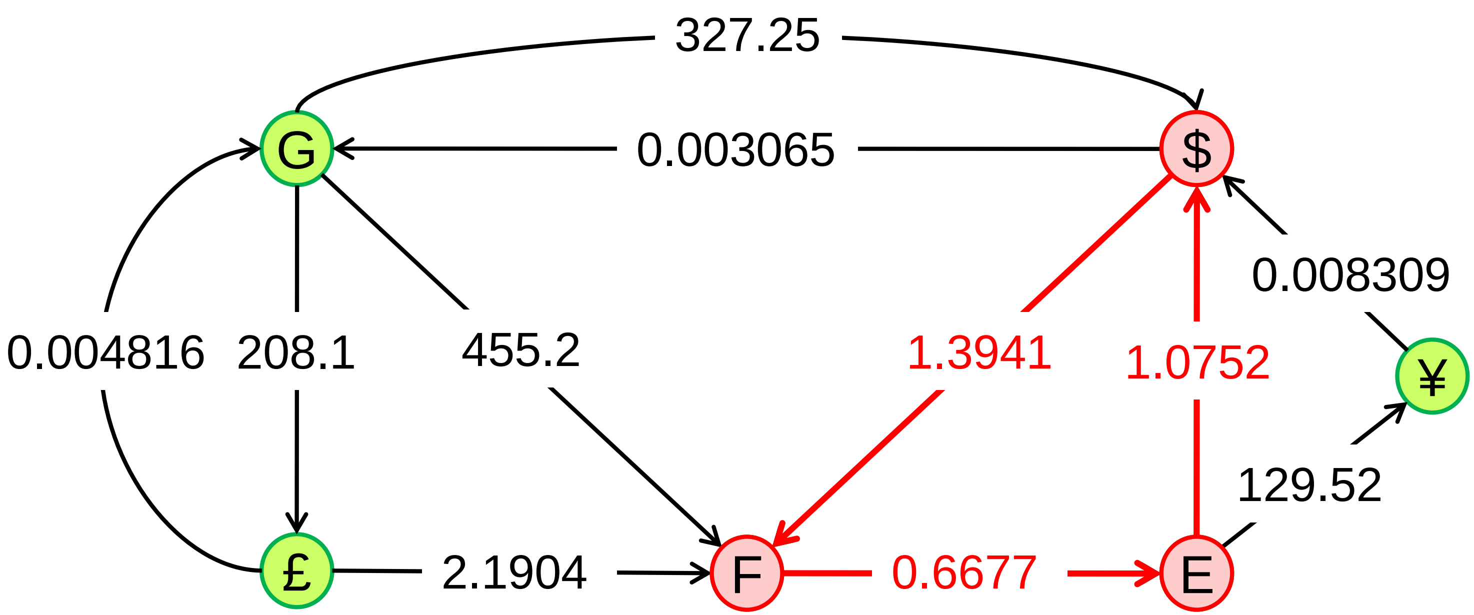
Negative Cycles?

- If a *s*-*t* path in a general graph *G* passes through node *v*, and *v* belongs to a negative cycle *C*, Bellman-Ford algorithm fails to find the shortest *s*-*t* path.
  - Reduce cost over and over again using the negative cycle



Application: Currency Conversion (1/2)

- Q: Given *n* currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?
  - The currency graph:
    - Node: currency; edge cost: exchange rate r_{uv}: $r_{uv} * r_{vu} < 1$
  - Arbitrage: a cycle on which product of edge costs >1
    - E.g., \$1 ⇒ 1.3941 Francs ⇒ 0.9308 Euros ⇒ \$1.00084



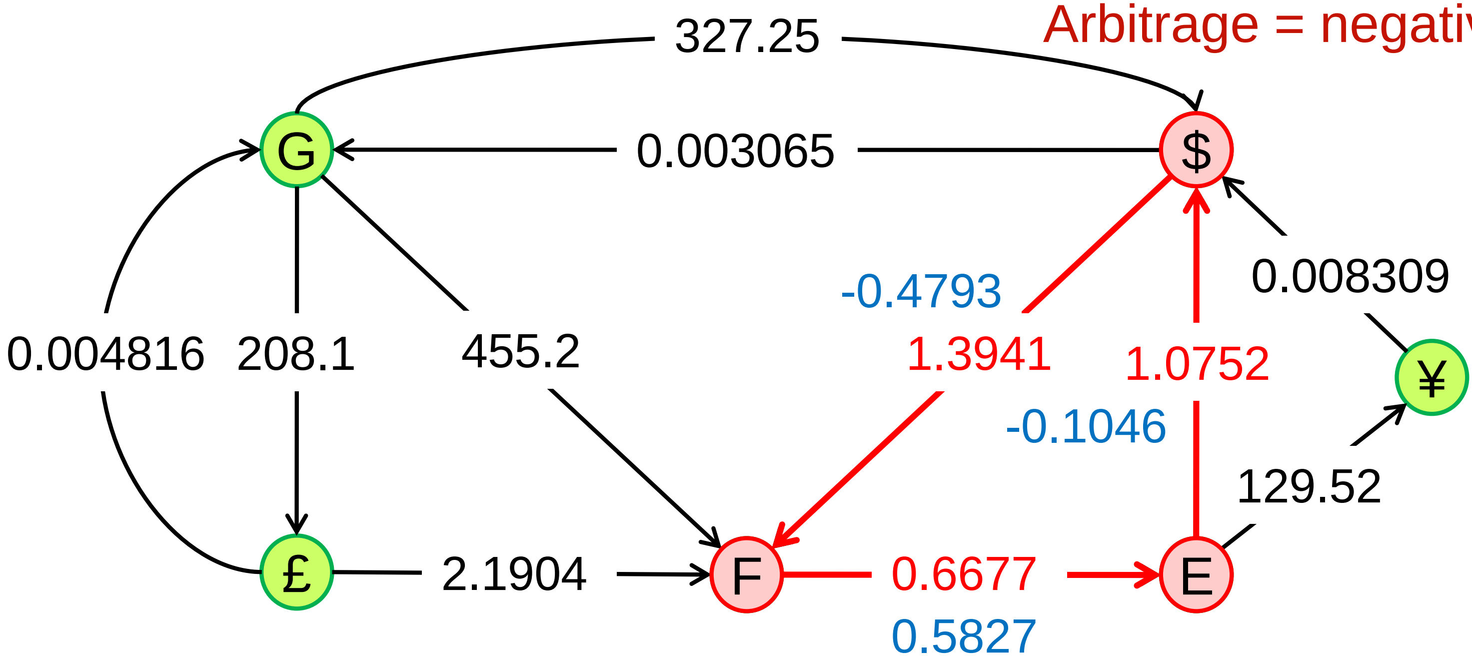
Courtesy of Prof. Kevin Wayne @ Princeton

Application: Currency Conversion (2/2)

- **Arbitrage**
  - Product of edge costs on a cycle $C = v_1, v_2, \ldots, v_1$
    - $r_{v1v2} * r_{v2v3} * \ldots * r_{vnv1}$
    - Arbitrage: $r_{v1v2} * r_{v2v3} * \ldots * r_{vnv1} > 1$

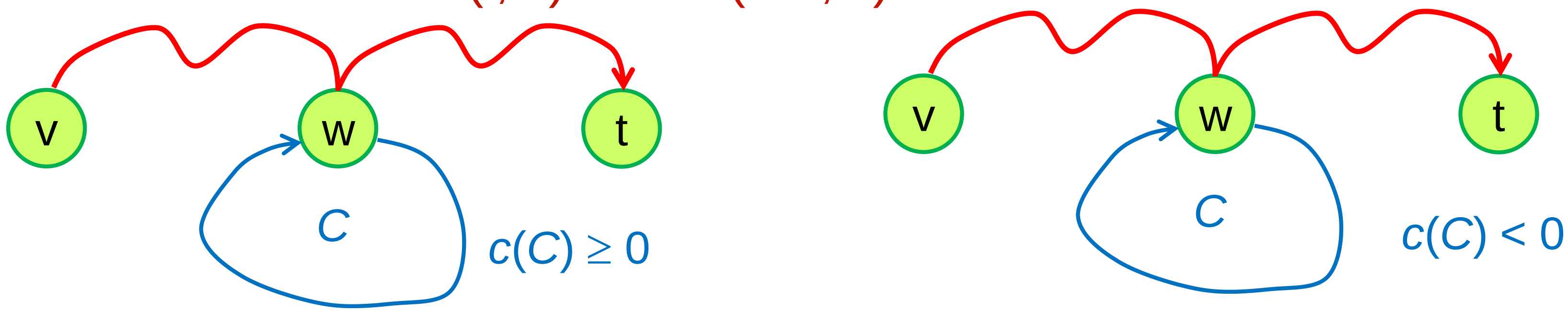
- **Negative cycle**
  - Sum of edge costs on a cycle $C = v_1, v_2, \ldots, v_1$
    - $c_{v1v2} + c_{v2v3} + \ldots + c_{vnv1}$
    - $c_{uv} = - \lg r_{uv}$

Arbitrage = negative cycle



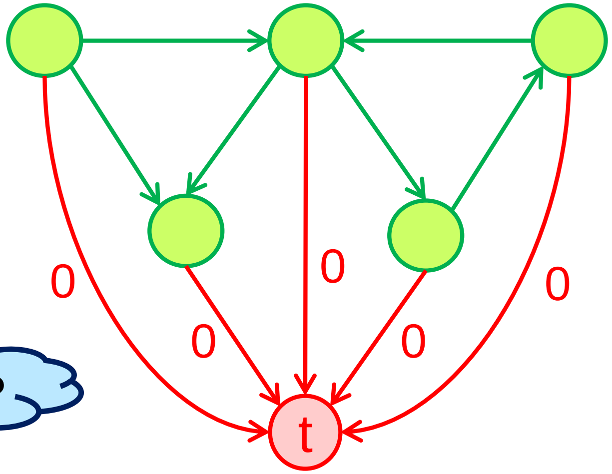
Negative Cycle Detection

- If OPT(n, v) = OPT(n-1, v) for all v, then no negative cycles.
  - Bellman-Ford: OPT(i, v) = OPT(n-1, v) for all v and $i \geq n$.



- If OPT(n, v) < OPT(n-1, v) for some v, then shortest path contains a negative cycle.
- Pf: by contradiction
  - Since OPT(n, v) < OPT(n-1, v), P has exactly n edges.
  - Every path using at most n-1 edges costs more than P.
  - (By pigeonhole principle,) P must contain a cycle C.
  - If C were not a negative cycle, deleting C yields a v-t path with < n edges and no greater cost. $\rightarrow\leftarrow$

Detecting Negative Cycles by Bellman-Ford



- Augmented graph *G'* of *G*
  1. Add new node *t*
  2. Connect all nodes to *t* with 0-cost edge
- *G* has a negative cycle
  iff *G'* has a negative cycle reaching *t*

Q: Why?

- Check if OPT(n, v) = OPT(n-1, v):
  – If yes, no negative cycles
  – If no, then extract cycle from shortest path from v to t

- Procedure:
  – Build the augmented graph *G'* for *G*
  – Run Bellman-Ford on *G'* for n iterations (instead of n-1).
  – Upon termination, Bellman-Ford successor variables trace a negative cycle if one exists.

Traveling Salesman Problem

Richard E. Bellman, 1962

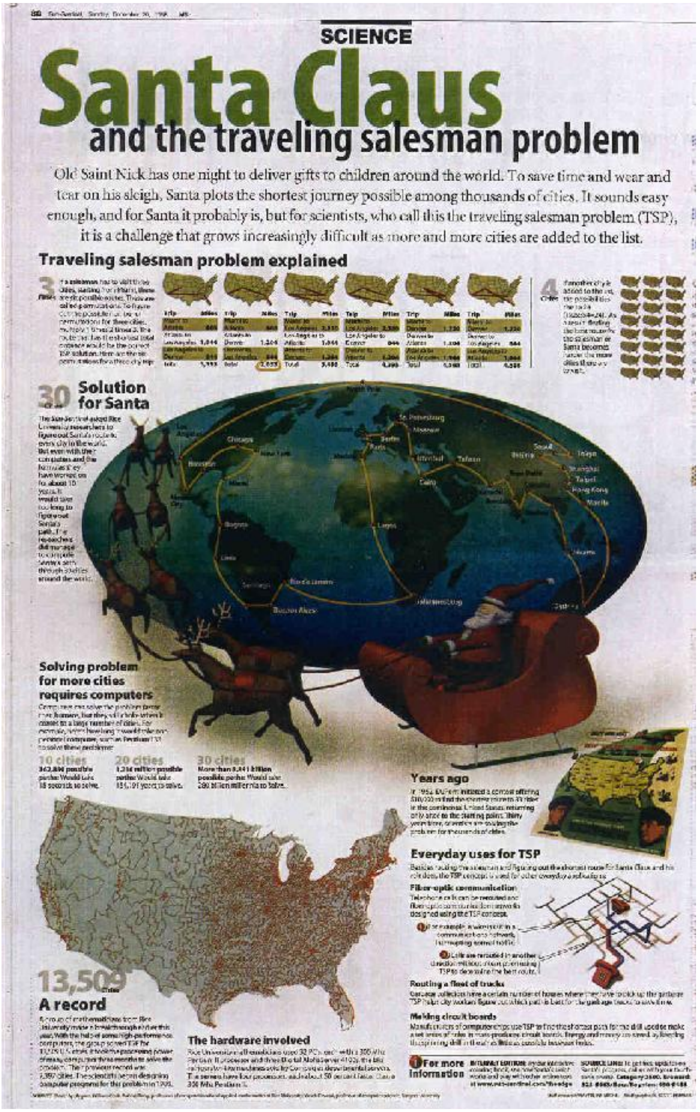
R. Bellman, Dynamic programming treatment of the travelling salesman problem. *J. ACM* 9, 1, Jan. 1962, pp. 61-63.

Travelling Salesman Problem

- TSP: A salesman is required to visit once and only once each of n different cities starting from a base city, and returning to this city. What path minimizes the total distance travelled by the salesman?
  - The distance between each pair of cities is given

- TSP contest
  - http://www.tsp.gatech.edu

- Brute-Force
  - Try all permutations: $O(n!)$



SCIENCE

Santa Claus
and the traveling salesman problem

Old Saint Nick has one night to deliver gifts to children around the world. To save time and wear and tear on his sleigh, Santa plots the shortest journey possible among thousands of cities. It sounds easy enough, and for Santa it probably is, but for scientists, who call this the traveling salesman problem (TSP), it is a challenge that grows increasingly difficult as more and more cities are added to the list.

Traveling salesman problem explained

30 Solution for Santa

Solving problem for more cities requires computers

Years ago

Everyday uses for TSP

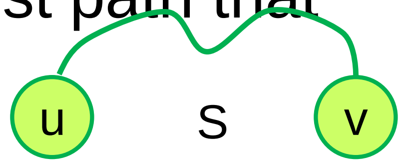
13,509 A record

The hardware involved

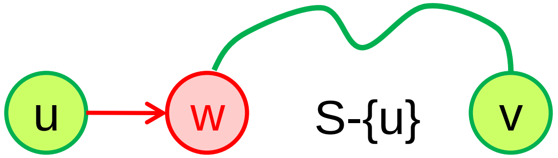
The Florida Sun-Sentinel, 20 Dec. 1998.

Dynamic Programming

- For each subset S of the cities with $|S| \geq 2$ and each u, $v \in S$, OPT(S, u, v) = the length of the shortest path that starts at u, ends at v, visits all cities in S



- Recurrence
  - Case 1: $S = \{u, v\}$
    - OPT(S, u, v) = $d(u, v)$
  - Case 2: $|S| > 2$
    - Assume $w \in S - \{u, v\}$ is visited first: OPT(S, u, v) = $d(u, w)$ + OPT(S-u, w, v)
    - OPT(S, u, v) = $\min_{w \in S - \{u,v\}}\{d(u, w) + \text{OPT}(S\text{-}u, w, v)\}$



- Efficiency
  - Space: $O(2^n n^2)$
  - Running time: $O(2^n n^3)$
    - Although much better thatn O(n!), DP is suitable when the number of subproblems is polynomial.

Summary: Dynamic Programming

- Smart recursion: In a nutshell, dynamic programming is recursion without repetition.
  - Dynamic programming is NOT about filling in tables; it's about smart recursion.
  - Dynamic programming algorithms store the solutions of intermediate subproblems often but not always in some kind of array or table.
  - A common mistake: focusing on the table (because tables are easy and familiar) instead of the much more important (and difficult) task of finding a correct recurrence.
- If the recurrence is wrong, or if we try to build up answers in the wrong order, the algorithm will NOT work!

Summary: Algorithmic Paradigms

- Brute-force (Exhaustive): Examine the entire set of possible solutions explicitly
  - A victim to show the efficiencies of the following methods
- Greedy: Build up a solution incrementally, myopically optimizing some local criterion.
  - Optimization problems that can be solved correctly by a greedy algorithm are very rare.
- Divide-and-conquer: Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming: Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.