國立臺灣大學
National Taiwan University

UNIT 4
DYNAMIC PROGRAMMING

Iris Hui-Ru Jiang
Spring 2024

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
  - Rod cutting
  - Matrix-chain multiplication
  - Longest common subsequence
  - Optimal binary search trees
  - Example: Subset sums and Knapsacks: adding a variable
  - Example: Traveling salesman problem
  - Example: Fibonacci sequence
- Reading:
  - Chapter 14

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-problem and solve it directly if simple enough
  - (Combine) Combine the solutions of sub-problems into an overall solution
- 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

- 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!!

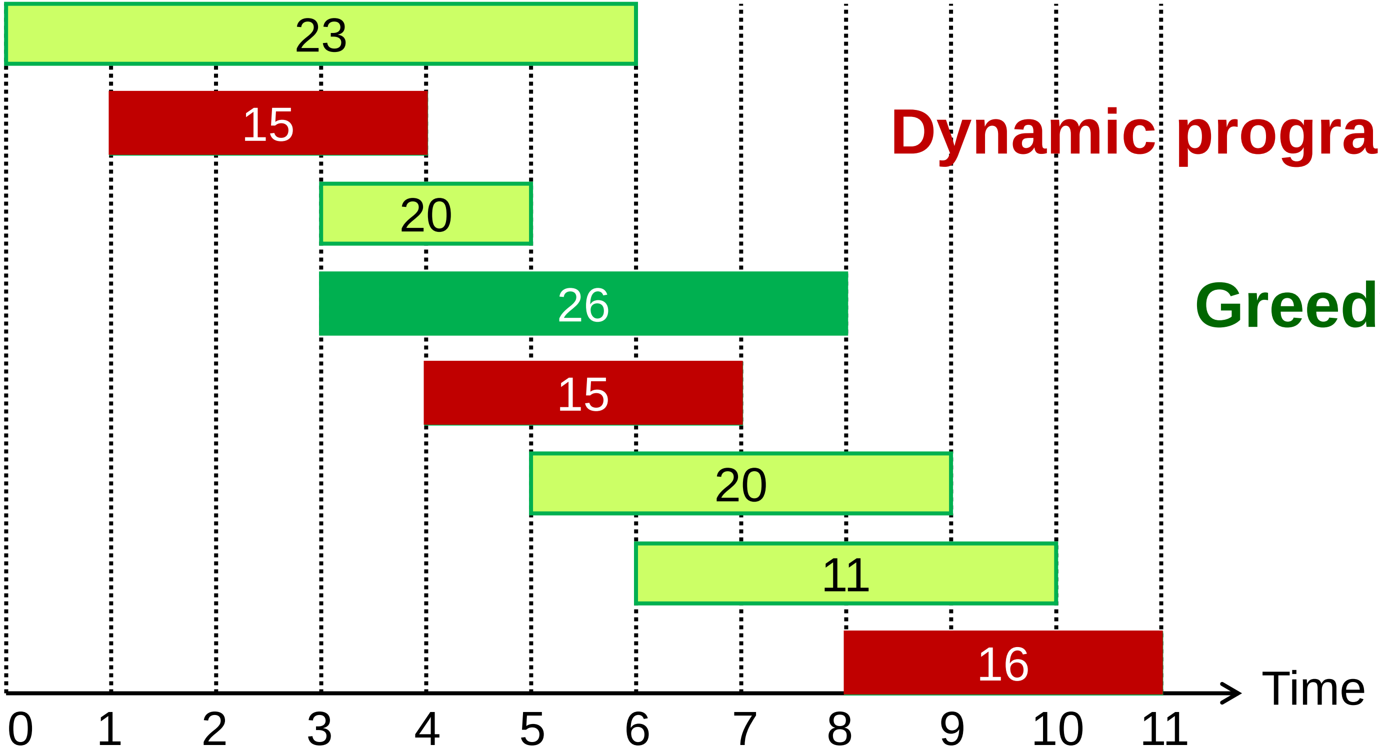
http://en.wikipedia.org/wiki/Dynamic_programming

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, $1 \leq i \leq n$
- Find: A subset *S* of mutually compatible intervals with maximum total weights (values)



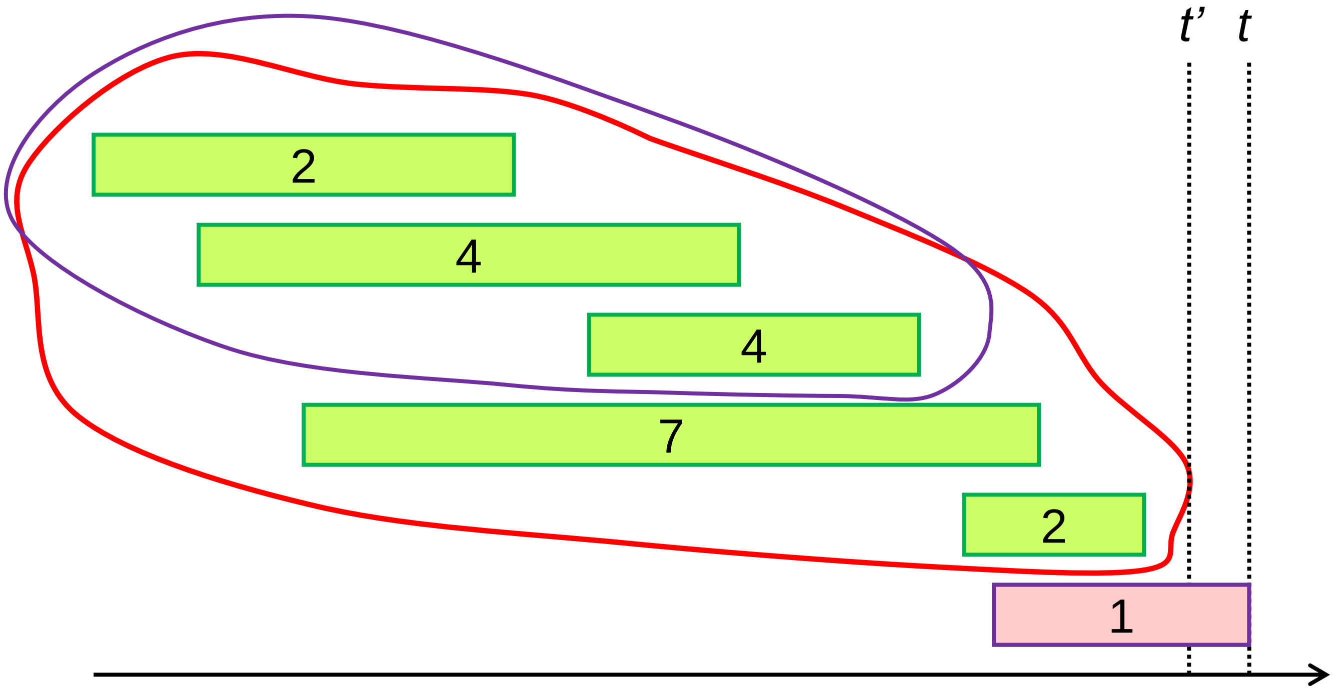
Dynamic programming!

Greedy?

Maximum weighted compatible set {15, 15, 16}

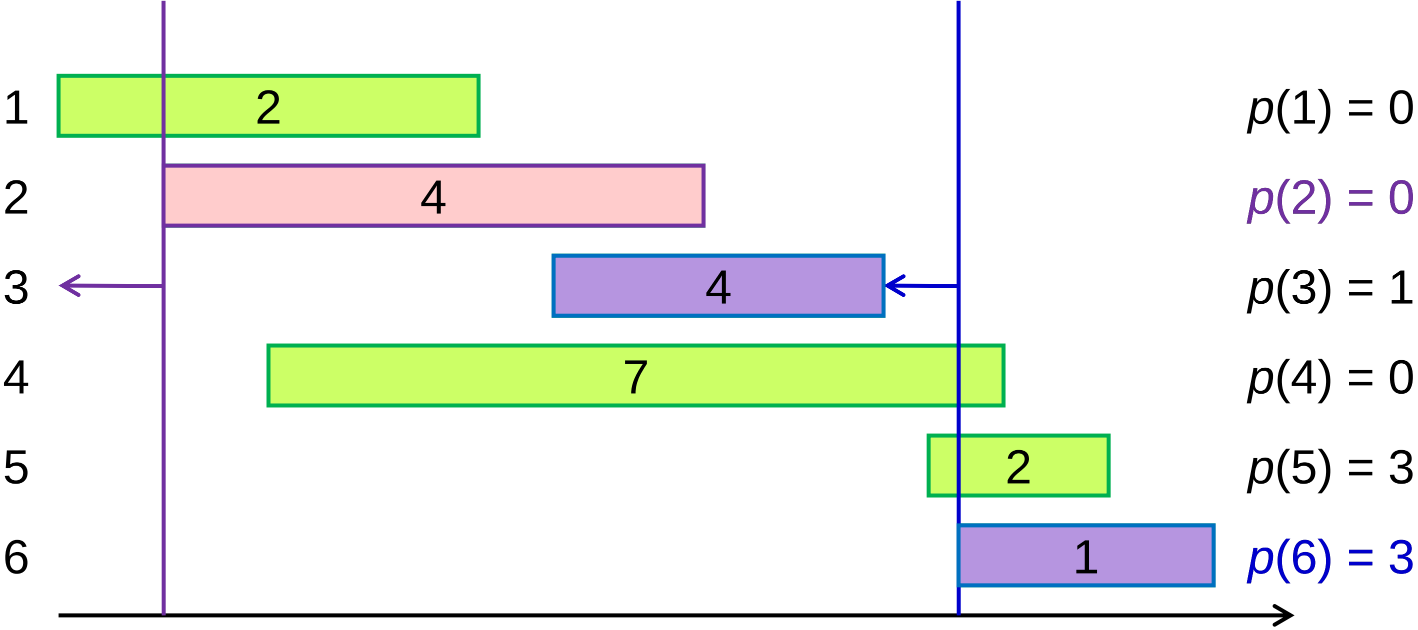
Designing a Recursive Algorithm (1/3)

- 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)

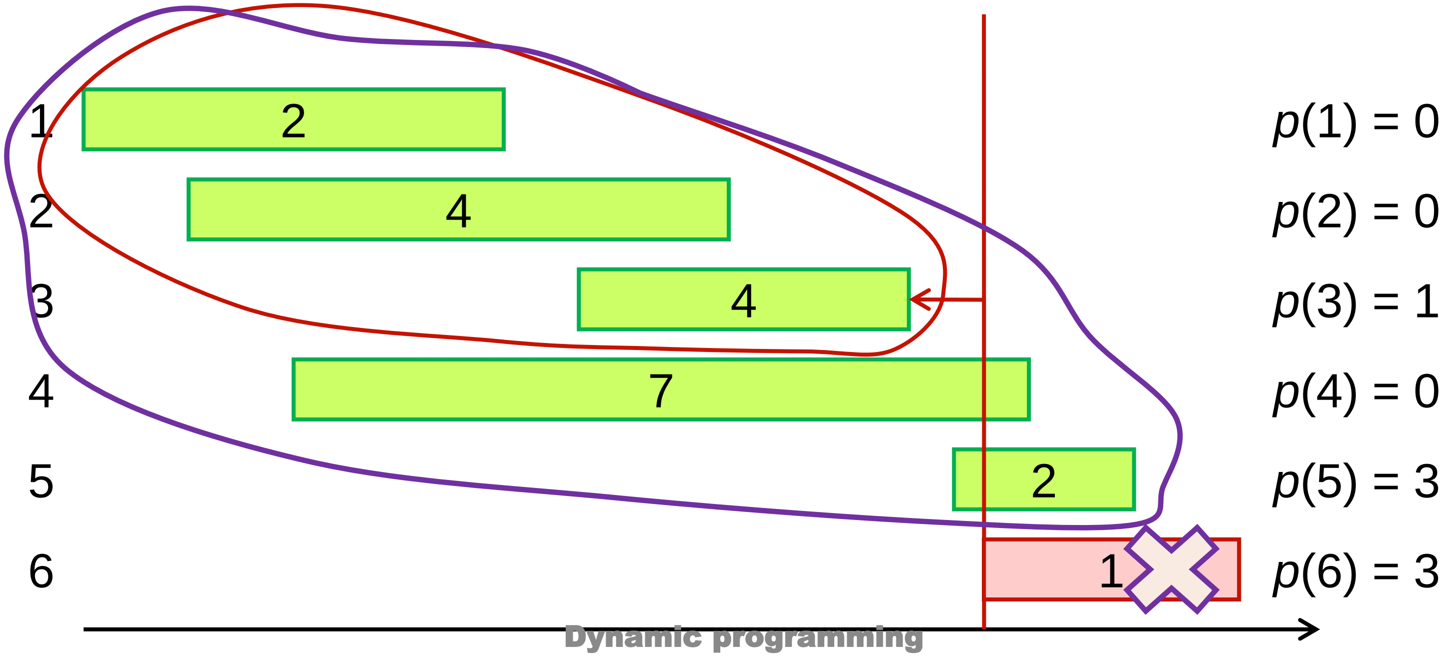
- Second attempt: Induction on interval index
  - First of all, sort intervals in ascending order of finish times
  - In fact, this is also a trick for DP
- *p*(*j*) is the largest index *i* < *j* s.t. intervals *i* and *j* are disjoint
  - *p*(*j*) = 0 if no request *i* < *j* is disjoint from *j*



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

First step of DP:
Define the subproblems!

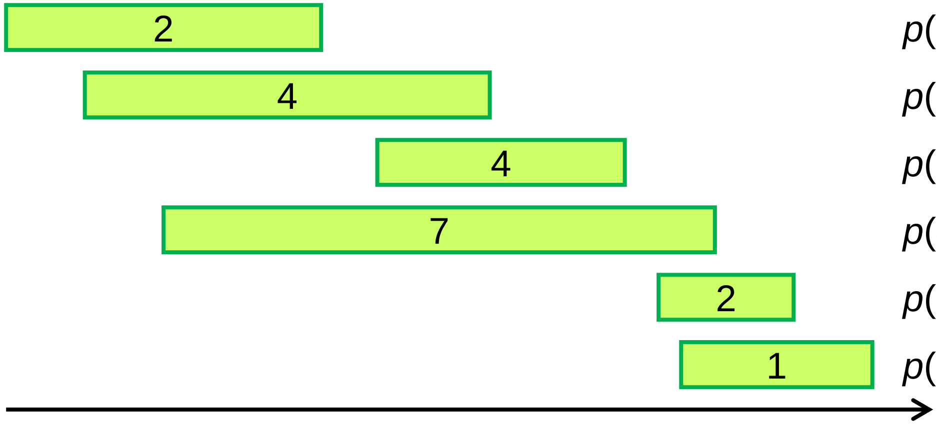


Direct Implementation

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

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

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

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



$p(1) = 0$

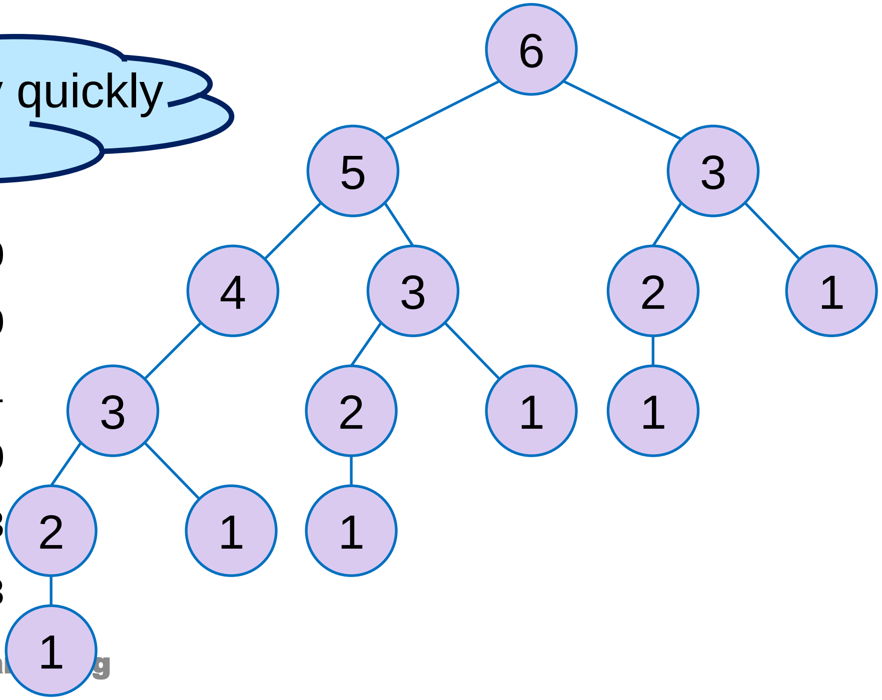
$p(2) = 0$

$p(3) = 1$

$p(4) = 0$

$p(5) = 3$

$p(6) = 3$



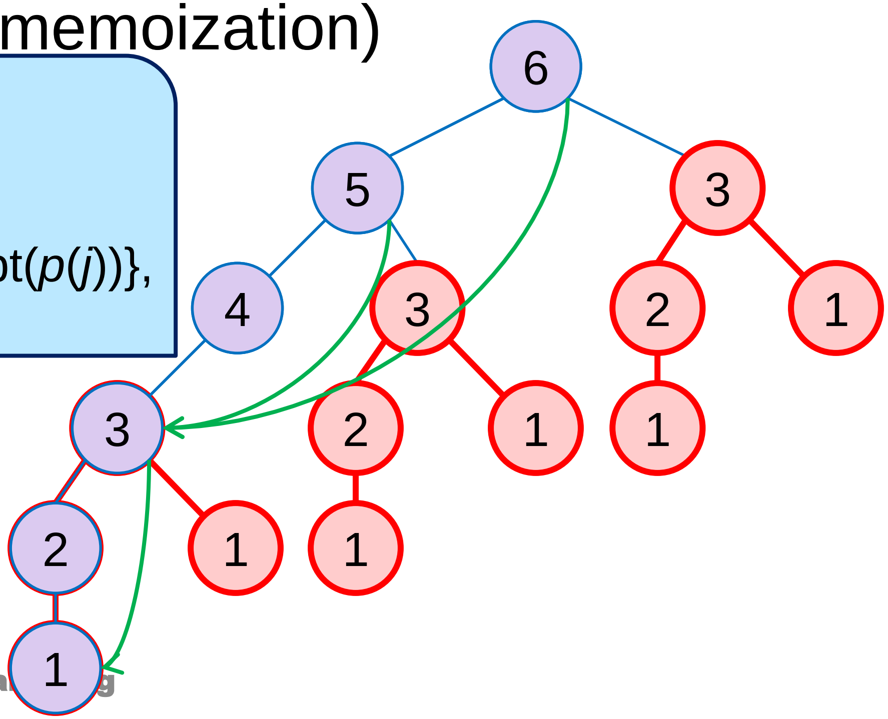
Memoization: Top-Down

- 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?

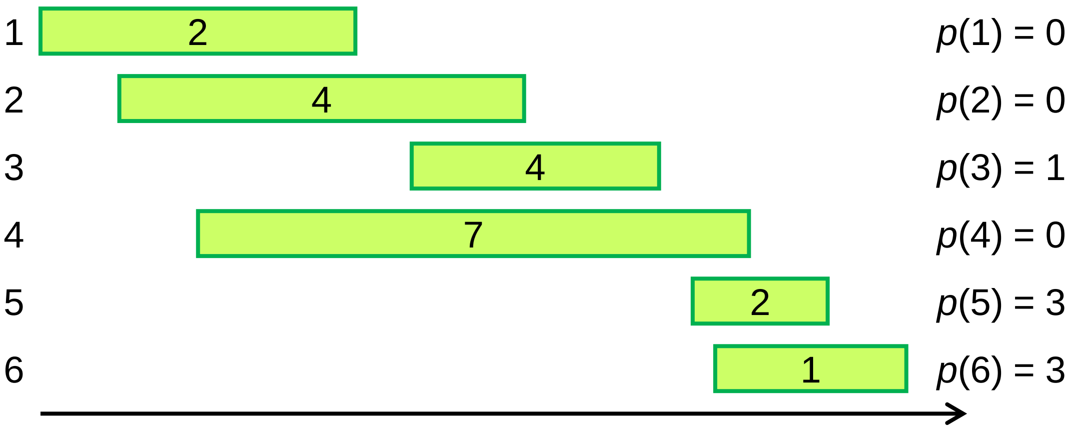


Iteration: Bottom-Up

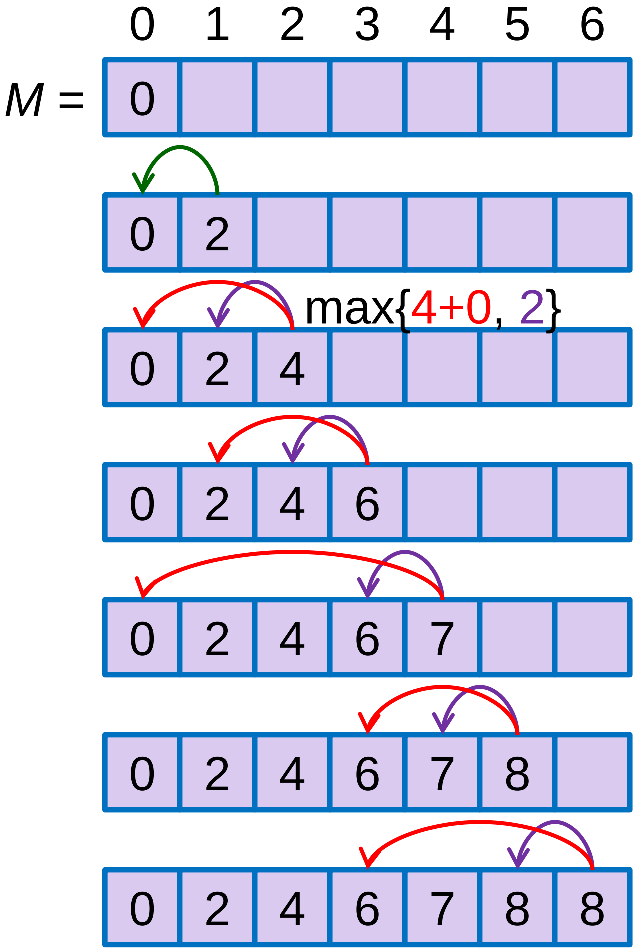
- We can also compute array *M*[*j*] by an iterative algorithm

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

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

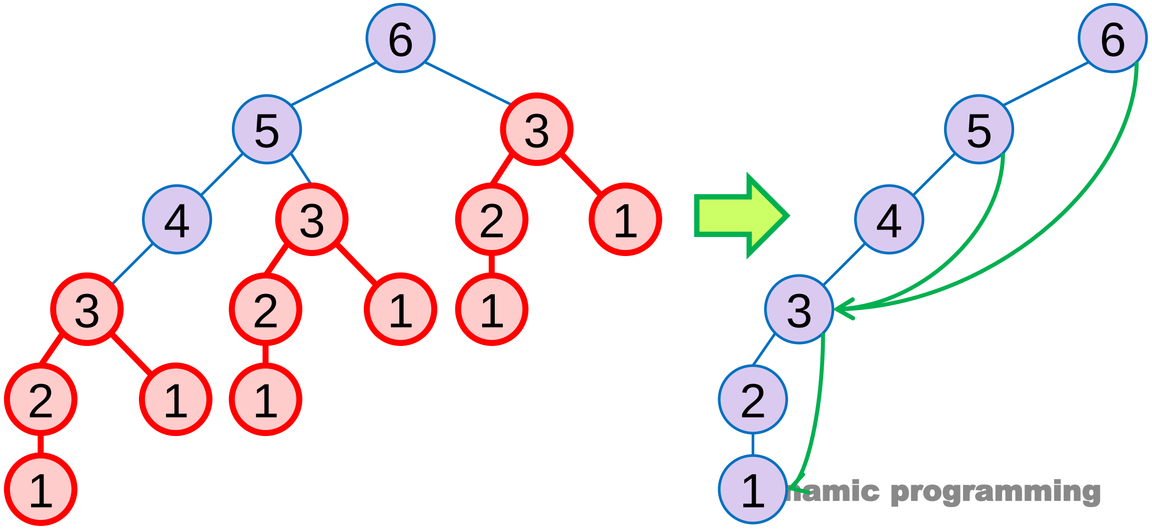
Start with the recursive divide-and-conquer algorithm

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

- If all subproblems must be solved at least once, bottom-up DP is better due to less overhead for recursion and for maintaining tables.
- If many subproblems need not be solved, top-down DP is better since it computes only those required

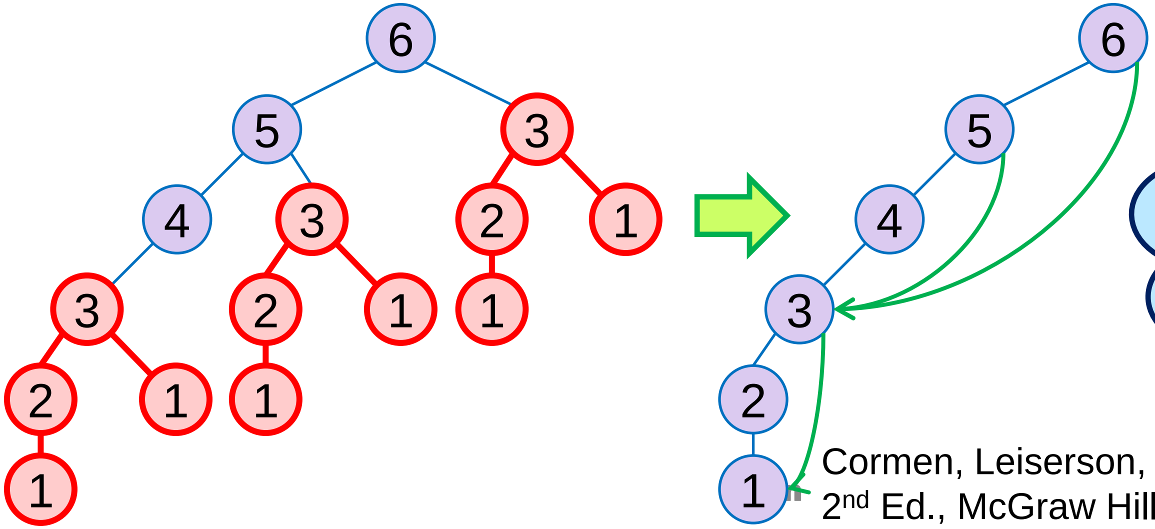
Keys for Dynamic Programming

- 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.

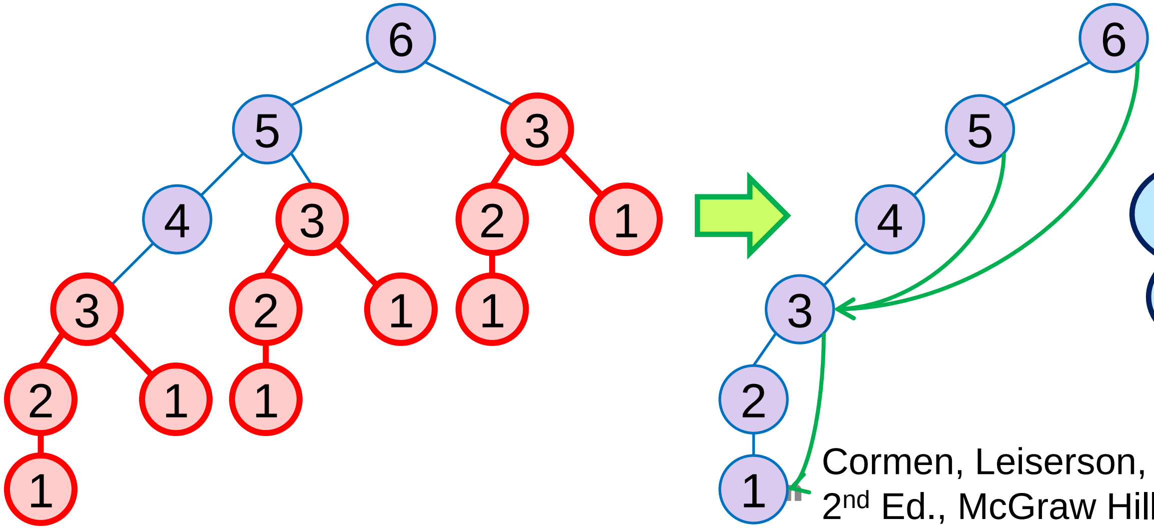


In optimization problems, we are interested in finding a *thing* which maximizes or minimizes some function.

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

Keys for Dynamic Programming

- 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.

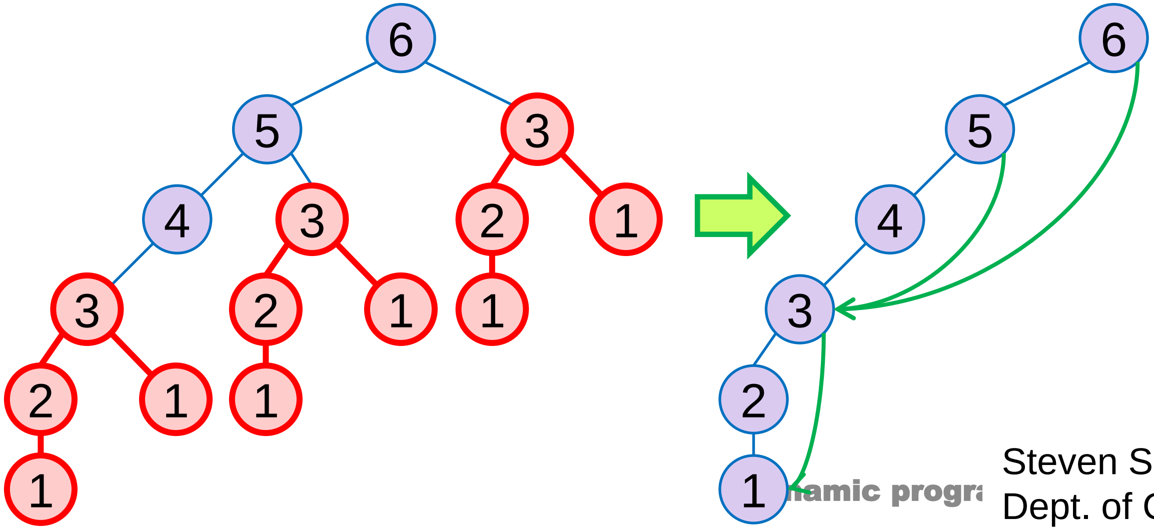


In optimization problems, we are interested in finding a *thing* which maximizes or minimizes some function.

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. Record **one** subproblem all the time

- Divide-and-conquer: Break up a problem into two or more subproblems, solve each sub-problem independently, and combine solution to subproblems to form solution to original problem **(disjoint subproblems)**

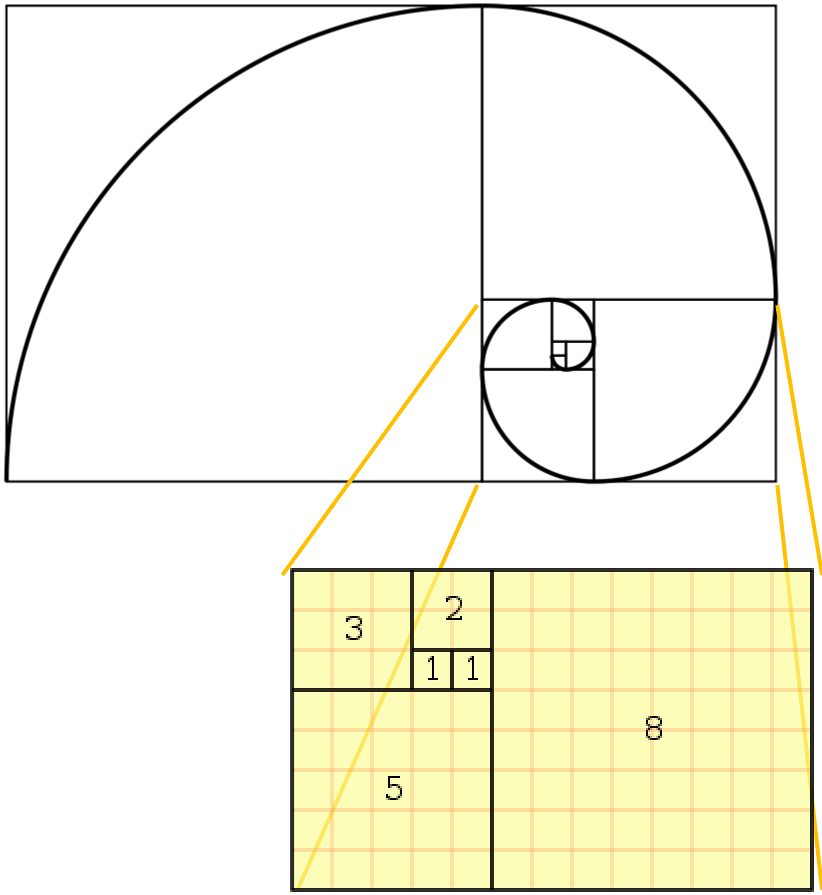
- Dynamic programming: Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems **(overlapping subproblems)**

Appendix: Fibonacci Sequence

Fibonacci Sequence

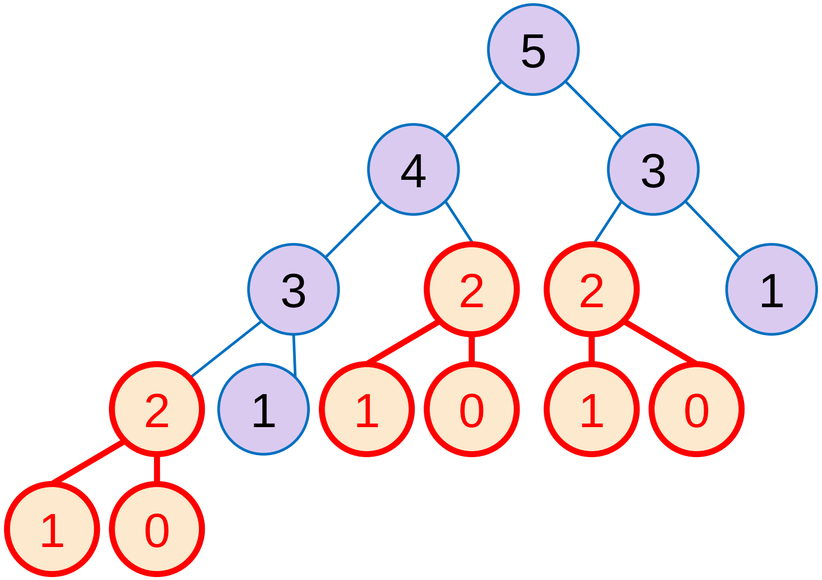
- Recurrence relation: $F_n = F_{n-1} + F_{n-2}$, $F_0=0$, $F_1=1$
  - e.g., 0, 1, 1, 2, 3, 5, 8, …
- Direct implementation:
  - Recursion!

```
fib($n$)
if $n \leq 1$ return $n$
return fib($n - 1$) + fib($n - 2$)
```



What’s Wrong?

```
fib($n$)
    **if** $n \leq 1$ **return** $n$
    **return** fib($n - 1$) + fib($n - 2$)
```

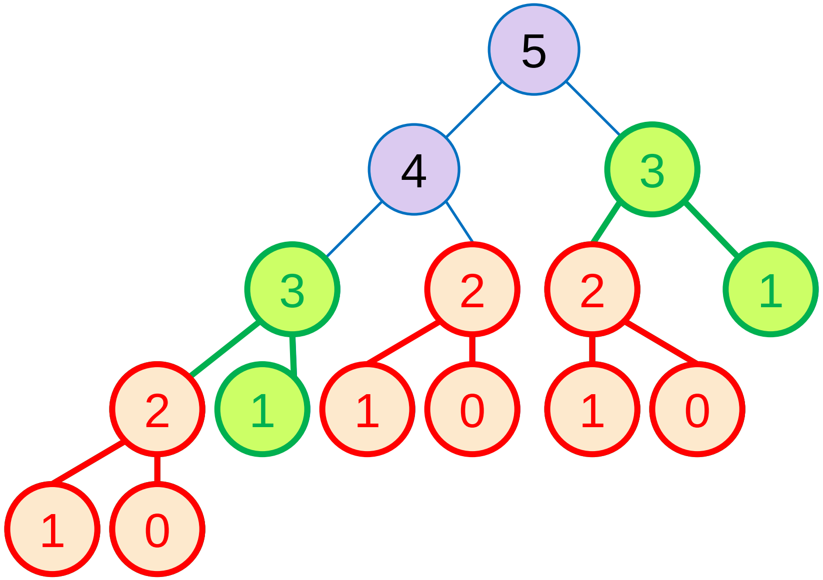


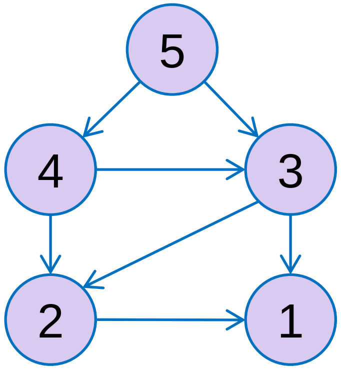
- What if we call fib(5)?
  - fib(5)
  - fib(4) + fib(3)
  - (fib(3) + fib(2)) + (fib(2) + fib(1))
  - ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
  - (((fib(1) + fib(0)) + fib(1))+(fib(1) + fib(0)))+((fib(1) + fib(0))+fib(1))
  - A call tree that calls the function on the 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

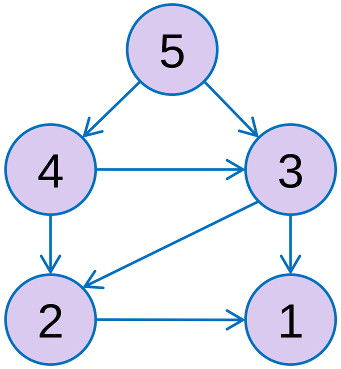




Dynamic Programming -- Memoization

- 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

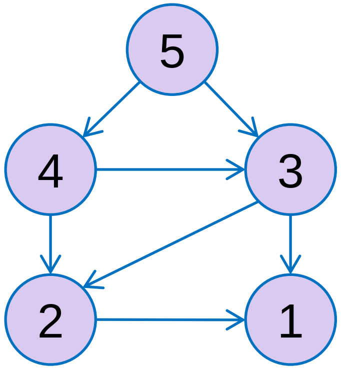
```
fib($n$)
    Initialize $f[0..n]$ with -1 // -1: unfilled
    $f[0] = 0$; $f[1] = 1$
    fibonacci($n$, $f$)

fibonacci($n$, $f$)
    If $f[n] == -1$ then
        $f[n]$ = fibonacci($n - 1$, $f$) + fibonacci($n - 2$, $f$)
    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($n$)
initialize $f[1..n]$ with -1 // -1: unfilled
$f[0] = 0$; $f[1] = 1$
for $i$=2 to $n$ do
    $f[i] = f[i-1]+ f[i-2]$
return $f[n]$
```



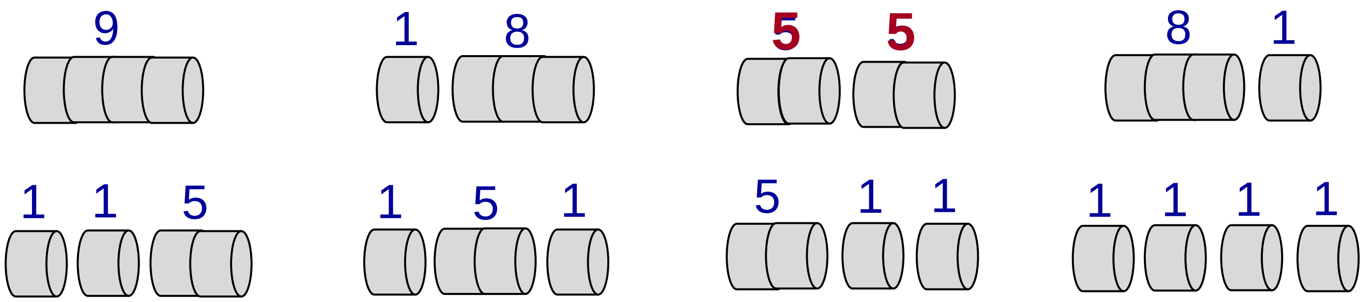
Rod Cutting

Rod Cutting

- Cut steel rods into pieces to maximize the revenue
  - Assumptions: Each cut is free; rod lengths are integers
- Input: A length *n* and table of prices p_i, for $i = 1, 2, \ldots, n$
- Output: The maximum revenue obtainable for rods whose lengths sum to *n*, computed as the sum of the prices for the individual rods

length *i*	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

length $i = 4$



max revenue
$r_4 = 5+5 = 10$

Objects are linearly ordered (and cannot be rearranged)??

Optimal Rod Cutting

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30
max revenue r_i	1	5	8	10	13	17	18	22	25	30

- If p_n is large enough, an optimal solution might require no cuts, i.e., just leave the rod as n unit long
- Solution for the maximum revenue r_i of length i

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) = \max_{1 \le i \le n} (p_i + r_{n-i})$$

$r_1 = 1$ from solution 1 = 1 (no cuts)

$r_2 = 5$ from solution 2 = 2 (no cuts)

$r_3 = 8$ from solution 3 = 3 (no cuts)

$r_4 = 10$ from solution 4 = 2 + 2

$r_5 = 13$ from solution 5 = 2 + 3

$r_6 = 17$ from solution 6 = 6 (no cuts)

$r_7 = 18$ from solution 7 = 1 + 6 or 7 = 2 + 2 + 3

$r_8 = 22$ from solution 8 = 2 + 6

Optimal Substructure

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30
max revenue r_i	1	5	8	10	13	17	18	22	25	30

- **Optimal substructure**: To solve the original problem, solve subproblems on smaller sizes. The optimal solution to the original problem incorporates optimal solutions to the subproblems
  - We may solve the subproblems independently
- After making a cut, we have **two subproblems**
  - Max revenue r_7: $r_7 = 18 = r_4 + r_3 = (r_2 + r_2) + r_3$ or $r_1 + r_6$

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

- Decomposition with **only one subproblem**: Some cut gives a first piece of length *i* on the left and a remaining piece of length *n* - *i* on the right $r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$

Recursive Top-Down “Solution”

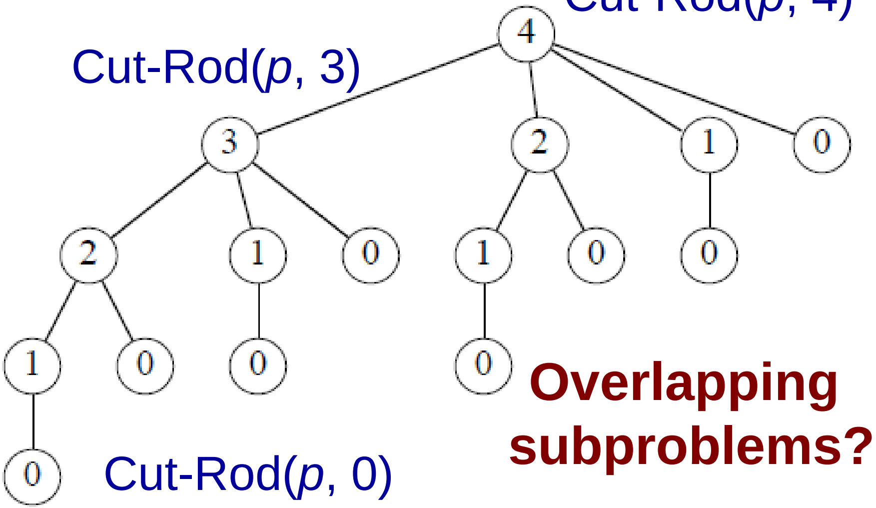
$$r_n = \max_{1\le i\le n}(p_i + r_{n-i})$$

```
Cut-Rod(p, n)
if n == 0
    return 0
q = - ∞
for i = 1 to n
    q = max(q, p[i] + Cut-Rod(p, n - i))
return q
```

- Inefficient solution: Cut-Rod calls itself repeatedly, even on subproblems it has already solved!!

$$T(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1 + \sum_{j=0}^{n-1} T(j), & \text{if } n > 0. \end{cases}$$

$T(n) = 2^n$



Cut-Rod(p, 4)

Cut-Rod(p, 3)

Cut-Rod(p, 1)

Cut-Rod(p, 0)

Overlapping subproblems?

Top-Down DP Cut-Rod with Memoization

- Complexity: $O(n^2)$ time
  - Solve each subproblem just once, and solves subproblems for sizes 0,1, …, n. To solve a subproblem of size n, the **for** loop iterates n times

```
Memoized-Cut-Rod($p$, $n$)
let $r[0..n]$ be a new array
for $i = 0$ to $n$
    $r[i] = -\infty$
return Memoized-Cut-Rod -Aux($p$, $n$, $r$)
```

```
Memoized-Cut-Rod-Aux($p$, $n$, $r$)
if $r[n] \geq 0$                        // Each overlapping subproblem
    return $r[n]$                           // is solved just once!!
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ to $n$
        $q = \max(q, p[i] +$ Memoized-Cut-Rod -Aux$(p, n-i, r))$
$r[n] = q$
return $q$
```

Bottom-Up DP Cut-Rod

- Complexity: $O(n^2)$ time
  - Sort the subproblems by size and solve smaller ones first.
  - When solving a subproblem, have already solved the smaller subproblems we need

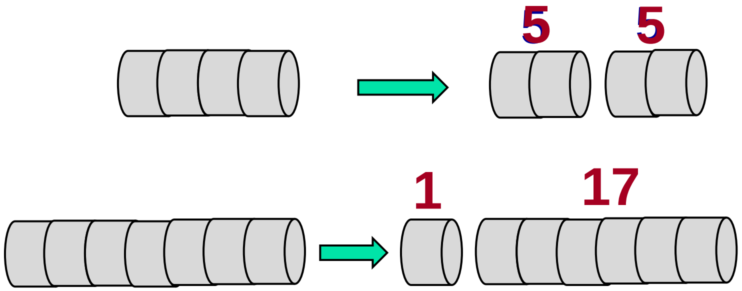
```
Bottom-Up-Cut-Rod($p$, $n$)
let $r[0..n]$ be a new array
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        $q = \max(q, p[i] + r[j - i])$
    $r[j] = q$
return $r[n]$
```

Bottom-Up DP with Solution Construction

- Extend the bottom-up approach to record not just optimal values, but optimal choices

- Saves the first cut made in an optimal solution for a problem of size *i* in *s*[*i*]

```
Extended-Bottom-Up-Cut-Rod($p$, $n$)
let $r[0..n]$ and $s[0..n]$ be new arrays
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        if $q < p[i] + r[j - i]$
            $q = p[i] + r[j - i]$
            $s[j] = i$
    $r[j] = q$
return $r$ and $s$
```



```
Print-Cut-Rod-Solution($p$, $n$)
$(r, s)$ = Extended-Bottom-Up-Cut-Rod($p$, $n$)
while $n > 0$
    print $s[n]$
    $n = n - s[n]$
```

i	0	1	2	3	4	5	6	7	8	9	10
r[*i*]	0	1	5	8	10	13	17	18	22	25	30
s[*i*]	0	1	2	3	2	2	6	1	2	3	10

Matrix-Chain Multiplication

DP Example: Matrix-Chain Multiplication

- If A is a p **x** q matrix and B a q **x** r matrix, then $C = AB$ is a p **x** r matrix

$$C[i,\ j] = \sum_{k=1}^{q} A[i,\ k]\ B[k,\ j]$$

time complexity: $O(pqr)$

```
Matrix-Multiply($A$, $B$)
if $A.columns \neq B.rows$
        error "incompatible dimensions"
else let $C$ be a new $A.rows * B.columns$ matrix
        for $i = 1$ to $A.rows$
            for $j = 1$ to $B.columns$
                $c_{ij} = 0$
                for $k = 1$ to $A.columns$
                    $c_{ij} = c_{ij} + a_{ik}b_{kj}$
        return $C$
```

DP Example: Matrix-Chain Multiplication

- **The matrix-chain multiplication problem**
  - Input: Given a chain <A_1, A_2, …, A_n> of n matrices, matrix $\boldsymbol{A_i}$ has dimension $\boldsymbol{p_{i-1}}$ **x** $\boldsymbol{p_i}$
  - Objective: Parenthesize the product A_1 A_2 … A_n to minimize the number of scalar multiplications

- **Exp:** dimensions: A_1: 4 **x** 2; A_2: 2 **x** 5; A_3: 5 **x** 1

  (A_1A_2)A_3: total multiplications = 4 **x** 2 **x** 5 + 4 **x** 5 **x** 1 = 60

  A_1(A_2A_3): total multiplications = 2 **x** 5 **x** 1 + 4 **x** 2 **x** 1 = 18

- So the order of multiplications can make a big difference!

Matrix-Chain Multiplication: Brute Force

- $A = A_1 A_2 \ldots A_n$: How to compute A using the minimum number of multiplications?
- Brute force: check all possible orders?
  - $P(n)$: number of ways to multiply n matrices.

$$P(n) = \begin{cases} 1 & \text{if } n = 1 , \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 . \end{cases}$$

  - $P(n) = \Omega(4^n / n^{3/2})$, **exponential** in n.
- Any efficient solution?
  - The matrix chain **is linearly ordered and cannot be rearranged!!**
  - Smell dynamic programming?

Matrix-Chain Multiplication

- $m[i, j]$: minimum number of multiplications to compute matrix $A_{i..j} = A_i A_{i+1} \dots A_j$, $1 \leq i \leq j \leq n$
  - $m[1, n]$: the cheapest cost to compute $A_{1..n}$

$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j\ . \end{cases}$$

$A_{i..j} = (A_i \dots A_k)(A_{k+1} \dots A_j)$

matrix $\boldsymbol{A_i}$ has dimension $\boldsymbol{p_{i-1}}$ **x** $\boldsymbol{p_i}$

- Applicability of dynamic programming
  - **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; only $\Theta(n^2)$ subproblems

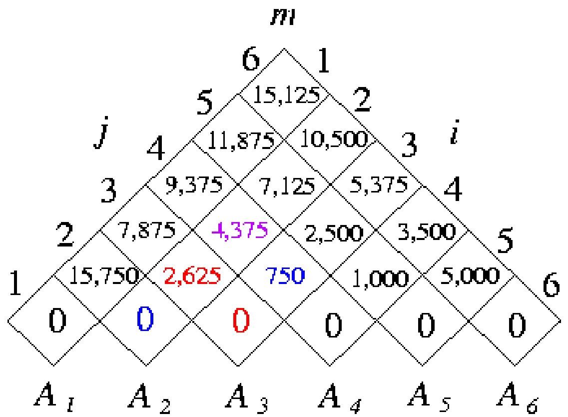
Bottom-Up DP Matrix-Chain Order

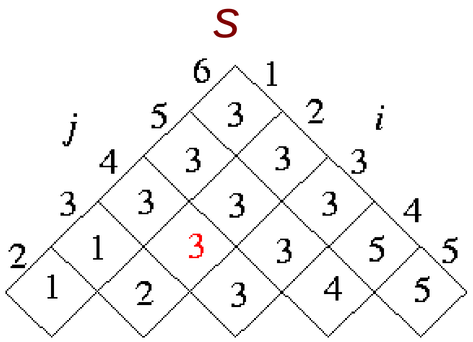
```
Matrix-Chain-Order(p)
n = p.length – 1
Let m[1..n, 1..n] and s[1..n-1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n        // l is the chain length
    for i = 1 to n – l +1
        j = i + l -1
        m[i, j] = ∞
        for k = i  to j -1
            q = m[i, k] + m[k+1, j] + $p_{i-1}p_kp_j$
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

A_i dimension

p_{i-1} x p_i

matrix	dimension
A_1	30 * 35
A_2	35 * 15
A_3	15 * 5
A_4	5 * 10
A_5	10 * 20
A_6	20 * 25





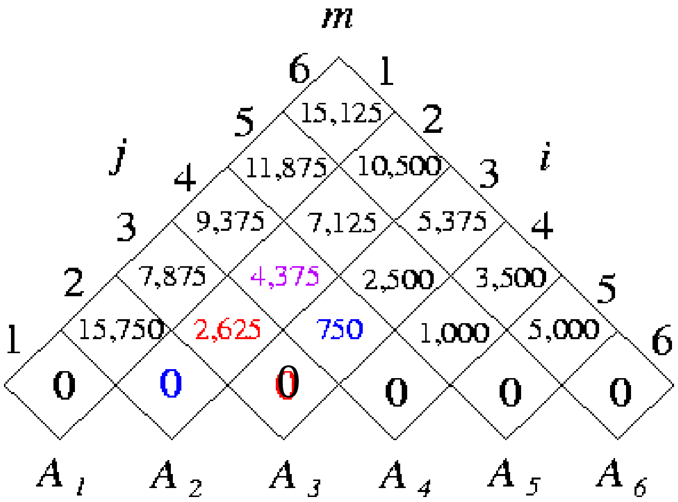
$$m[2,4] = \min \begin{cases} m[2,2] + m[3,4] + p_1p_2p_4 = 0 + 750 + 35 \times 15 \times 10 = 6000 . \\ m[2,3] + m[4,4] + p_1p_3p_4 = 2625 + 0 + 35 \times 5 \times 10 = 4375 . \end{cases}$$

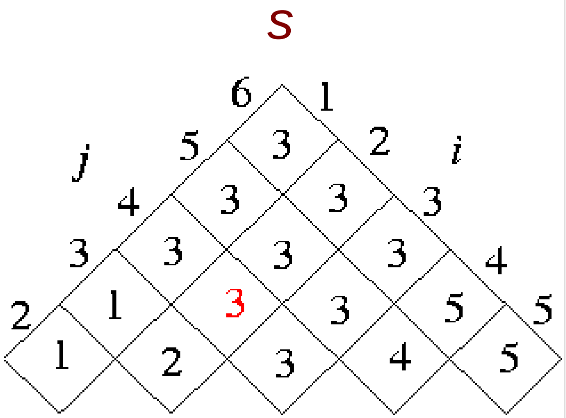
Constructing an Optimal Solution

- *s*[*i*, *j*]: value of *k* such that the optimal parenthesization of $A_i A_{i+1} \ldots A_j$ splits between A_k and A_{k+1}
- Optimal matrix $A_{1..n}$ multiplication: $A_{1..s[1,n]} A_{s[1,n]+1..n}$
- **Exp:** call Print-Optimal-Parens(*s*, 1, 6): $((A_1 (A_2 A_3))((A_4 A_5) A_6))$

```
Print-Optimal-Parens(s, i, j)
if i == j
        print "A"_i
else print "("
        Print-Optimal-Parens(s, i, s[i, j])
        Print-Optimal-Parens(s, s[i, j] + 1, j)
        print ")"
```

matrix	dimension
A_1	30 * 35
A_2	35 * 15
A_3	15 * 5
A_4	5 * 10
A_5	10 * 20
A_6	20 * 25

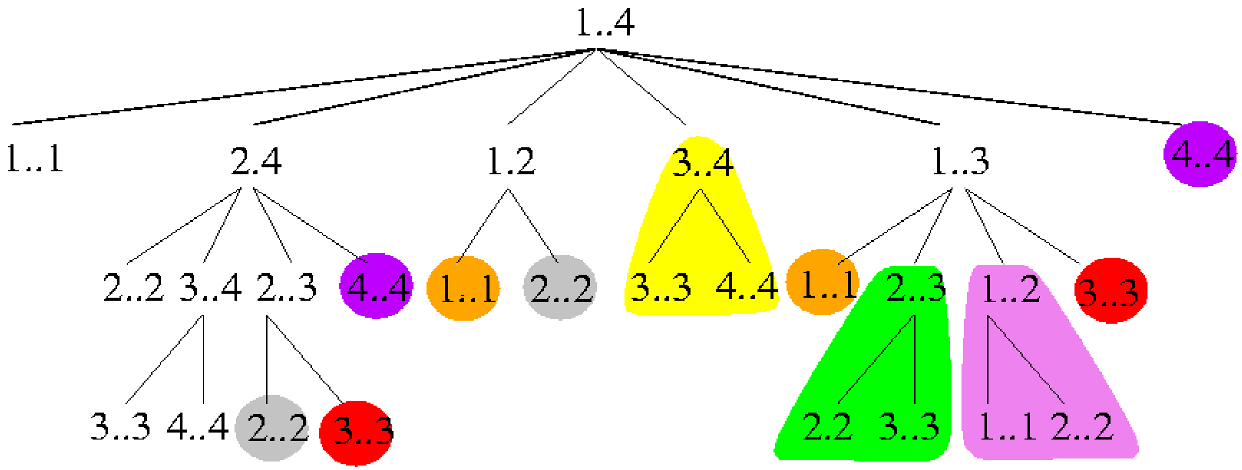




Top-Down, Recursive Matrix-Chain Order

- Time complexity: $\Omega(2^n)$ ($T(n) > \sum_{k=1}^{n-1}(T(k)+T(n-k)+1)$)

```
Recursive-Matrix-Chain($p$, $i$, $j$)
if $i == j$
    return 0
$m[i, j] = \infty$
for $k = i$ to $j - 1$
    $q$ = Recursive-Matrix-Chain($p$, $i$, $k$)
          + Recursive-Matrix-Chain($p$, $k+1$, $j$) + $p_{i-1}p_kp_j$
    if $q < m[i, j]$
        $m[i, j] = q$
return $m[i, j]$
```



Top-Down DP Matrix-Chain Order (Memoization)

- Complexity: $O(n^2)$ space for $m[]$ matrix and $O(n^3)$ time to fill in $O(n^2)$ entries (each takes $O(n)$ time)

```
Memoized-Matrix-Chain($p$)
n = $p.length$ - 1
let $m[1..n, 1..n]$ be a new table
for $i$ = 1 to $n$
    for $j$ = $i$ to $n$
        $m[i, j] = \infty$
return Lookup-Chain($m$, $p$, 1, $n$)
```

```
Lookup-Chain($m$, $p$, $i$, $j$)
if $m[i, j] < \infty$
    return $m[i, j]$
if $i == j$
    $m[i, j] = 0$
else for $k$ = $i$ to $j$ - 1
        $q$ = Lookup-Chain($m$, $p$, $i$, $k$) + Lookup-Chain($m$, $p$, $k$+1, $j$) + $p_{i-1}p_kp_j$
        if $q < m[i, j]$
            $m[i, j] = q$
return $m[i, j]$
```

Longest Common Subsequence

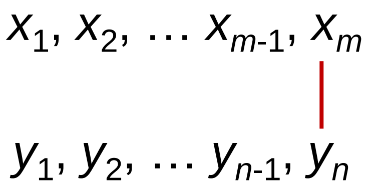
Longest Common Subsequence

- **Problem:** Given $X = <x_1, x_2, \ldots, x_m>$ and $Y = < y_1, y_2, \ldots, y_n>$, find the **longest common subsequence (LCS)** of X and Y.
- **Exp:** $X = <a, b, c, b, d, a, b>$ and $Y = <b, d, c, a, b, a>$
  LCS = $<b, c, b, a>$ (also, LCS = $<b, d, a, b>$).
- Exp: DNA sequencing: measure similarity
  - S1 = ACCGGTCGAGATGCAG;
    S2 = GTCGTTCGGAATGCAT;
    LCS S3 = CGTCGGATGCA
- Brute-force method:
  - Enumerate all subsequences of X and check if they appear in Y
  - Each subsequence of X corresponds to a subset of the indices $\{1, 2, \ldots, m\}$ of the elements of X
  - There are 2^m subsequences of X. Why?

Objects are linearly ordered (and cannot be rearranged)??

Optimal Substructure for LCS

- **First step: define subproblems!**
- **Theorem:** Let $X = <x_1, x_2, \ldots, x_m>$ and $Y = <y_1, y_2, \ldots, y_n>$ be sequences, and $Z = <z_1, z_2, \ldots, z_k>$ be LCS of X and Y
  1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
  2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies Z is an LCS of X_{m-1} and Y
  3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies Z is an LCS of X and Y_{n-1}
  - An LCS of two sequences contains within it an LCS of prefixes of the two sequences
- $c[i, j]$: length of the LCS of X_i and Y_j
- $c[m, n]$: length of LCS of X and Y
- Basis: $c[0, j] = 0$ and $c[i, 0] = 0$



$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases}$$

Top-Down DP for LCS

- $c[i, j]$: length of the LCS of X_i and Y_j, where $X_i = <x_1, x_2, \ldots, x_i>$ and $Y_j=<y_1, y_2, \ldots, y_j>$
- $c[m, n]$: LCS of X and Y
- Basis: $c[0, j] = 0$ and $c[i, 0] = 0$

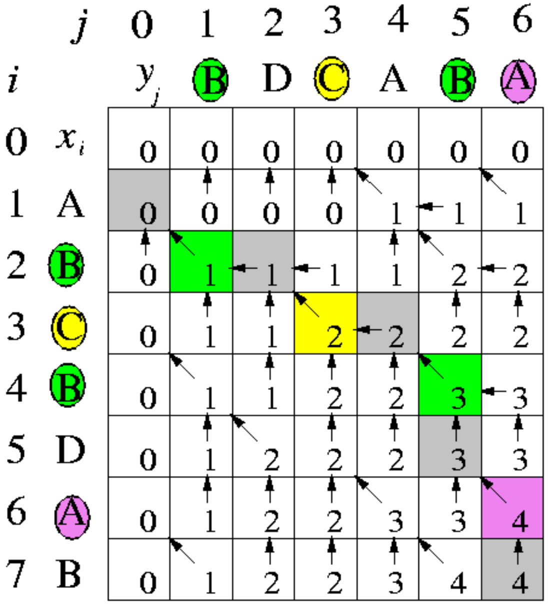
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

- Top-down DP: initialize $c[i, 0] = c[0, j] = 0$, $c[i, j] = \text{NIL}$

```
TD-LCS($i$, $j$)
if $c[i, j]$ == NIL                    // check memo
    if $x_i == y_j$
        $c[i, j]$ = TD-LCS($i$-1, $j$-1) + 1
    else $c[i, j]$ = max(TD-LCS($i$, $j$-1), TD-LCS($i$-1, $j$))
return $c[i, j]$
```

Bottom-Up DP for LCS

- **Find the right order to solve the subproblems**
- To compute $c[i, j]$, we need $c[i-1, j-1]$, $c[i-1, j]$, and $c[i, j-1]$
- $b[i, j]$: points to the table entry w.r.t. the optimal subproblem solution chosen when computing $c[i, j]$

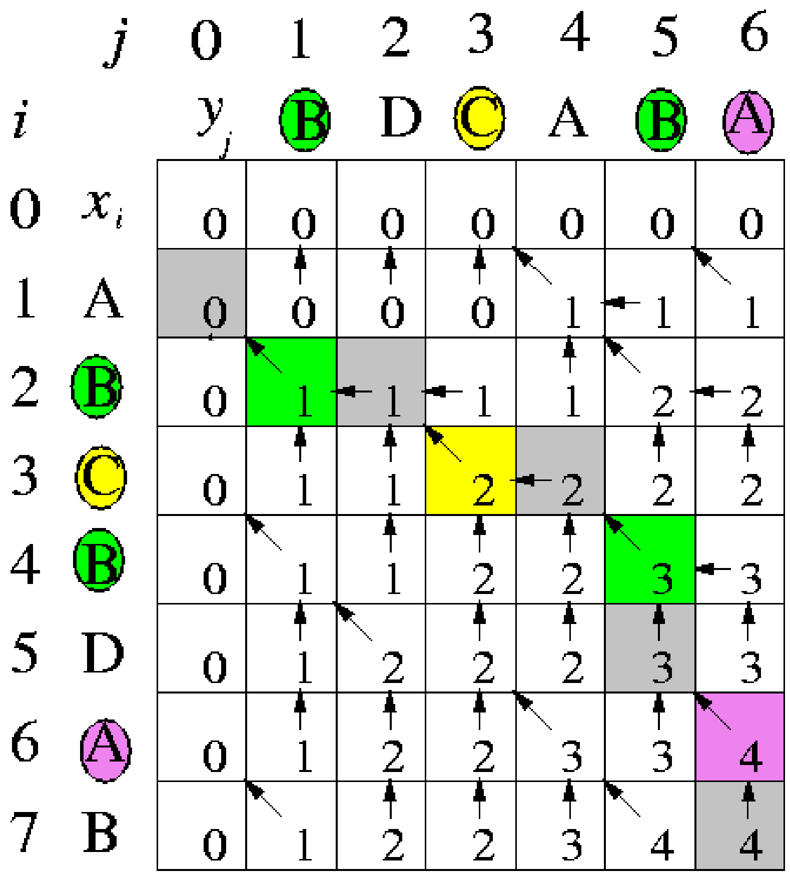


	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	↑ 0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

```
LCS-Length(X,Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n]
    be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i-1, j-1]+1
            b[i, j] = "↖"
        elseif c[i-1,j] ≥ c[i, j-1]
            c[i,j] = c[i-1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j-1]
            b[i, j] = "←"
return c and b
```

Example of LCS

- LCS time and space complexity: $\Theta(mn)$
- X = <A, B, C, B, D, A, B> and Y = <B, D, C, A, B, A> ⇒ LCS = <B, C, B, A>

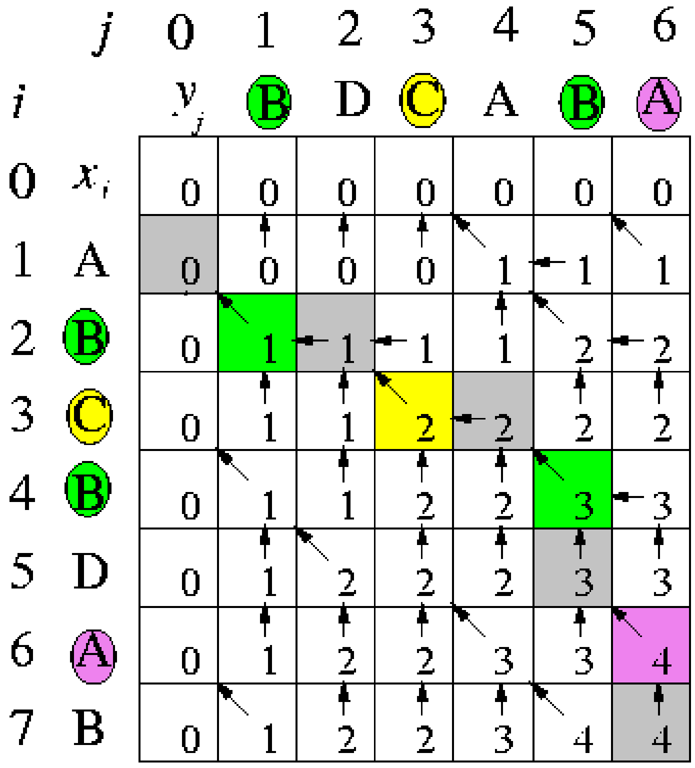


	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Constructing an LCS

- Trace back from $b[m, n]$ to $b[1, 1]$, following the arrows: $O(m+n)$ time

```
Print-LCS($b$, $X$, $i$, $j$)
if $i$ == 0 or $j$ == 0
    return
if $b[i, j]$ == "↖ "
    Print-LCS($b$, $X$, $i$-1, $j$-1)
    print $x_i$
elseif $b[i, j]$ == " ↑ "
    Print-LCS($b$, $X$, $i$ -1, $j$)
else Print-LCS($b$, $X$, $i$, $j$-1)
```

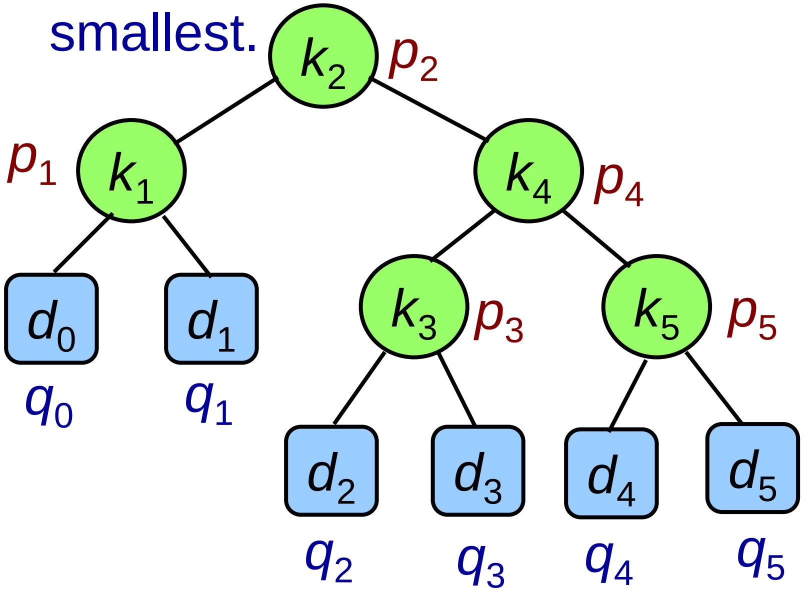


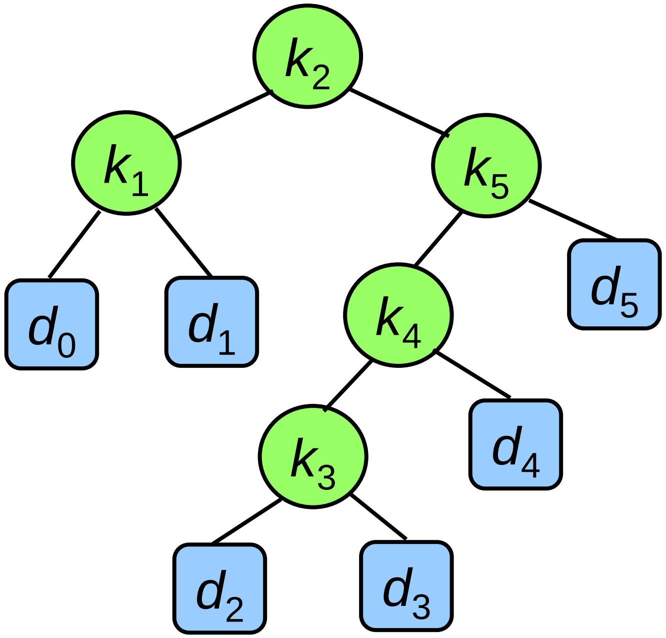
	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Optimal Binary Search Tree

Optimal Binary Search Tree

- Given
  - a sequence $K = <k_1, k_2, \ldots, k_n>$ of n distinct keys in sorted order ($k_1 < k_2 < \ldots < k_n$) a set of probabilities $P = <p_1, p_2, \ldots, p_n>$ for searching the keys in K
  - $Q = <q_0, q_1, q_2, \ldots, q_n>$ for unsuccessful searches (corresponding to $D = <d_0, d_1, d_2, \ldots, d_n>$ of $\boldsymbol{n+1}$ distinct dummy keys with d_i representing all values between k_i and k_{i+1})
- Goal
  - construct a **binary search tree** whose expected search cost is smallest.

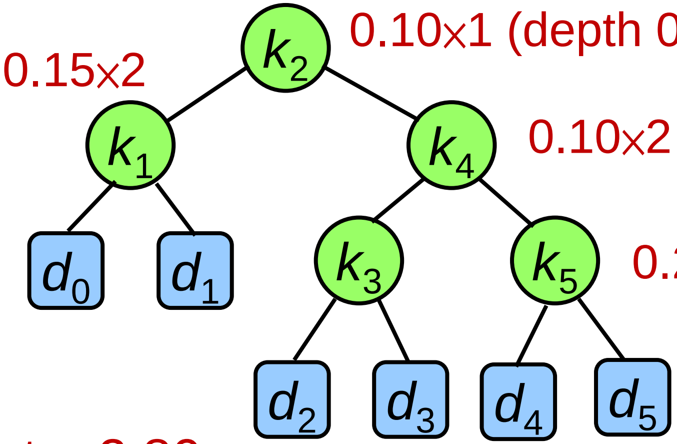




An Example

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$



0.10×1 (depth 0)

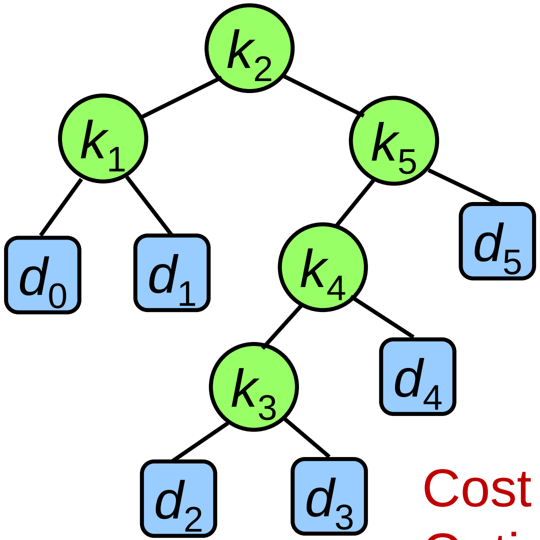
0.15×2

0.10×2

0.20×3

0.10×4

Cost = 2.80



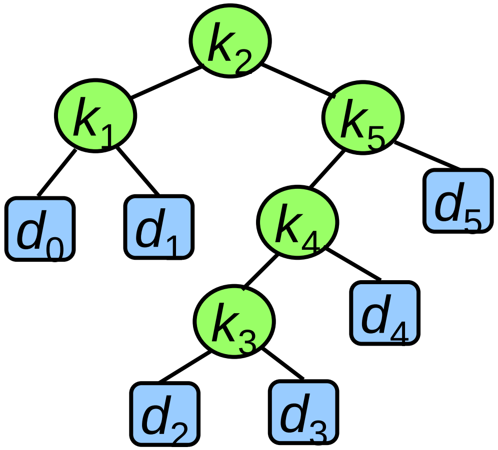
Cost = 2.75

Optimal!!

$$\begin{aligned} \mathrm{E}\left[\text{search cost in } T\right] &= \sum_{i=1}^{n} (\operatorname{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\operatorname{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \operatorname{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \operatorname{depth}_T(d_i) \cdot q_i \end{aligned}$$

Optimal Substructure

- If an optimal binary search tree T has a subtree T' containing keys $k_i, \ldots, k_j$, then this subtree T' must be optimal as well for the subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$
  - Given keys $k_i, \ldots, k_j$ with $\mathbf{k_r}$ ($i \leq r \leq j$) as the root, the left subtree contains the keys $k_i, \ldots, k_{r-1}$ (and dummy keys $d_{i-1}, \ldots, d_{r-1}$) and the right subtree contains the keys $k_{r+1}, \ldots, k_j$ (and dummy keys $d_r, \ldots, d_j$)
  - For the subtree with keys $k_i, \ldots, k_j$ with root k_i, the left subtree contains keys **k_i, .., k_{i-1} (no key)** and the dummy key d_{i-1}



Overlapping Subproblem: Recurrence

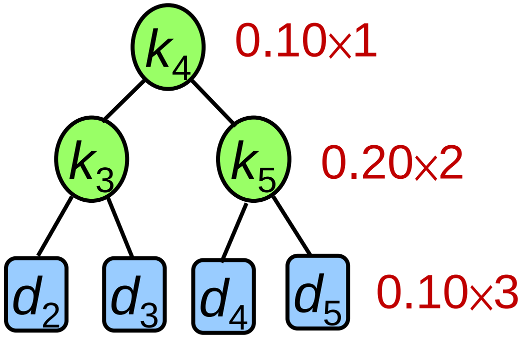
- $e[i, j]$: expected cost of searching an optimal binary search tree containing the keys $k_i, \ldots, k_j$
  - Want to find $e[1, n]$
  - $e[i, i-1] = q_{i-1}$ (only the dummy key d_{i-1})
- If k_r ($i \le r \le j$) is the root of an optimal subtree containing keys $k_i, \ldots, k_j$ and let $w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$ then

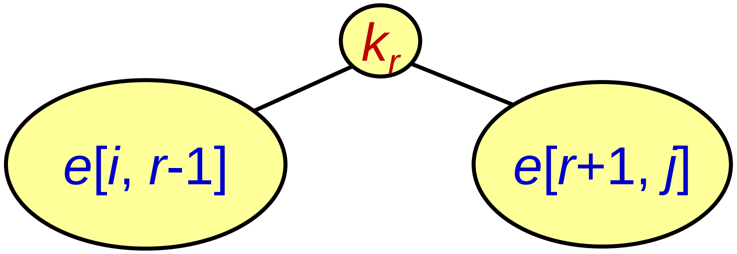
Node depths increase by 1 after merging two subtrees, and so do the costs

$$e[i, j] = \mathbf{p_r} + (e[i, r-1] + \mathbf{w(i, r-1)}) + (e[r+1, j] + \mathbf{w(r+1, j)})$$

$$= e[i, r-1] + e[r+1, j] + \mathbf{w(i, j)}$$

- Recurrence: $$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ \min\limits_{i \le r \le j}\{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \le j \end{cases}$$

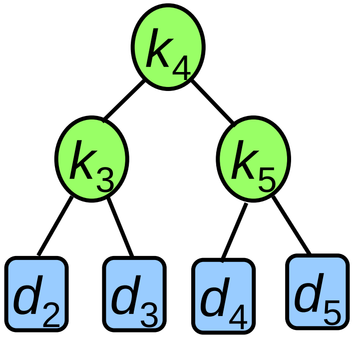




Computing the Optimal Cost

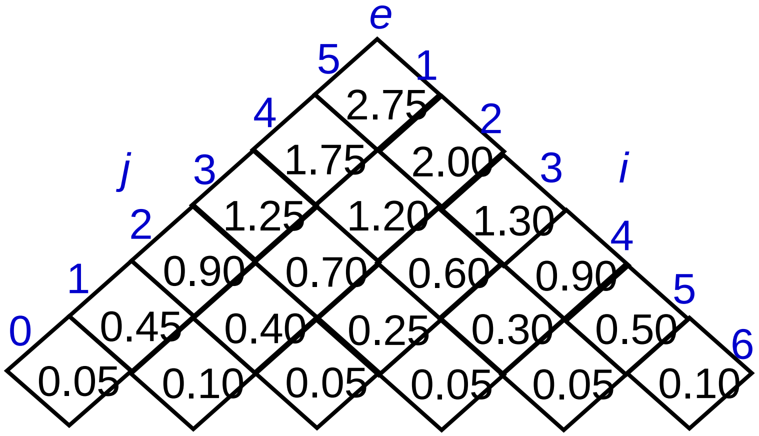
- Need a table $e[1..n+1, 0..n]$ for $e[i, j]$ (why $e[1, 0]$ and $e[n+1, n]$?)
- Apply the recurrence to compute $w(i, j)$ (why?)

$$w[i, j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ w[i, j-1] + p_j + q_j & \text{if } i \le j \end{cases}$$



```
Optimal-BST(p, q, n)
let e[1..n+1, 0..n], w[1..n+1, 0..n],
    and root[1..n, 1..n] be new tables
for i = 1 to n + 1
    e[i, i-1] = $q_{i-1}$
    w[i, i-1] = $q_{i-1}$
for l = 1 to n
    for i = 1 to n – l + 1
        j = i + l -1
        e[i, j] = ∞
        w[i, j] = w[i, j-1] + $p_j$ + $q_j$
        for r = i to j
            t = e[i, r-1] + e[r+1, j] + w[i, j]
            if t < e[i, j]
                e[i, j] = t
                root[i, j] = r
return e and root
```

- $root[i, j]$: index r for which k_r is the root of an optimal search tree containing keys $k_i, \ldots, k_j$



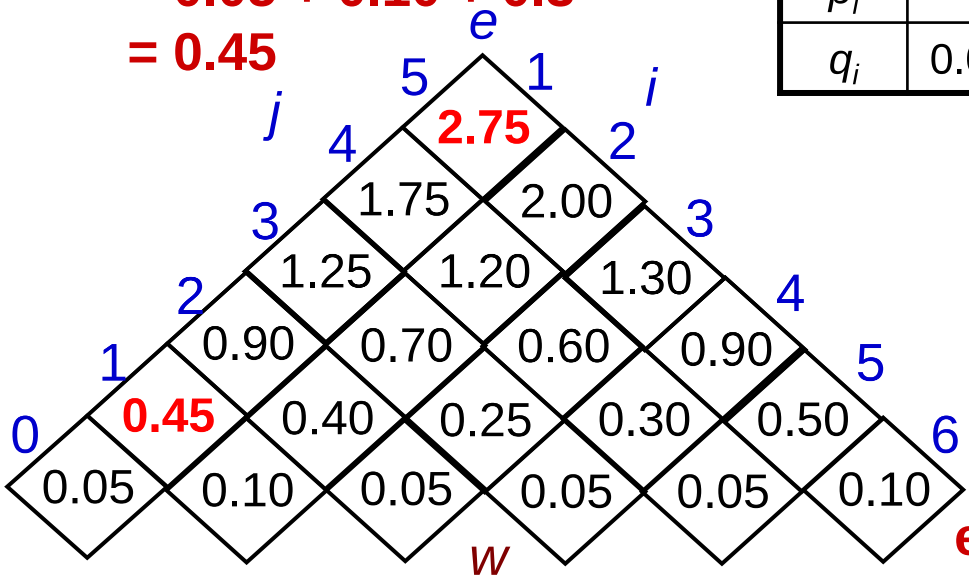
2.75					
1.75	2.00				
1.25	1.20	1.30			
0.90	0.70	0.60	0.90		
0.45	0.40	0.25	0.30	0.50	
0.05	0.10	0.05	0.05	0.05	0.10

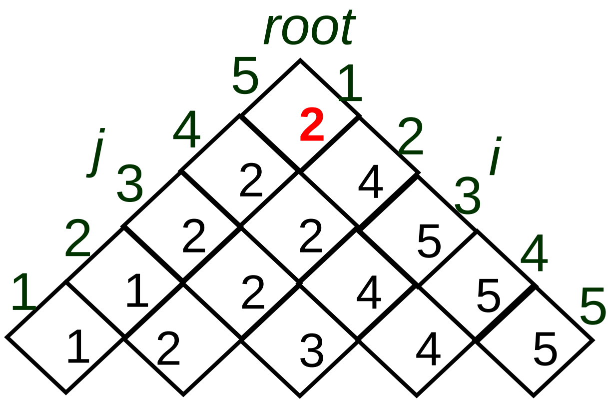
Example

$$e[i,j]=\begin{cases} q_{i-1} & \text{if } j=i-1 \\ \min\limits_{i\le r\le j}\{e[i,r-1]+e[r+1,j]+w(i,j)\} & \text{if } i\le j \end{cases}$$

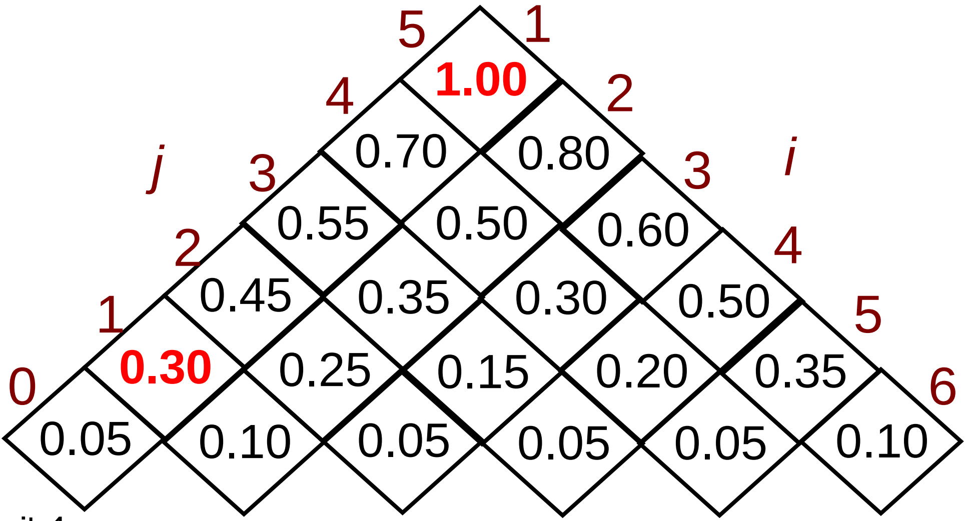
i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

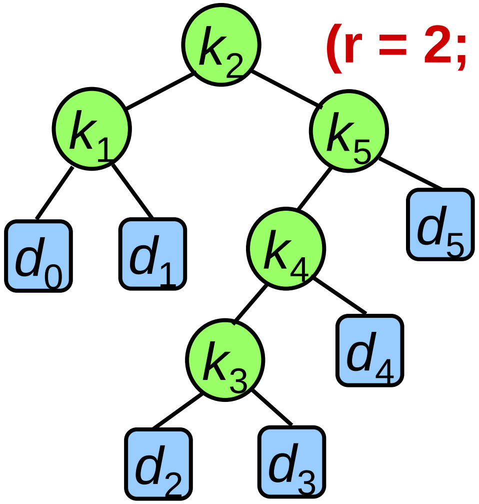
e[1, 1] = e[1, 0] + e[2, 1] + w(1,1)
= 0.05 + 0.10 + 0.3
= 0.45





e[1, 5] = e[1, 1] + e[3, 5] + w(1,5)
= 0.45 + 1.30 + 1.00 = 2.75
(r = 2; r=1, 3?)

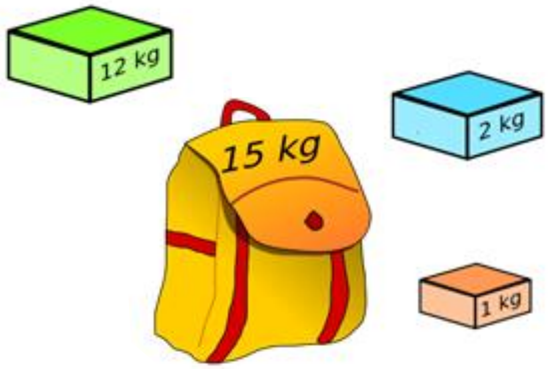




Subset Sums & Knapsacks

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

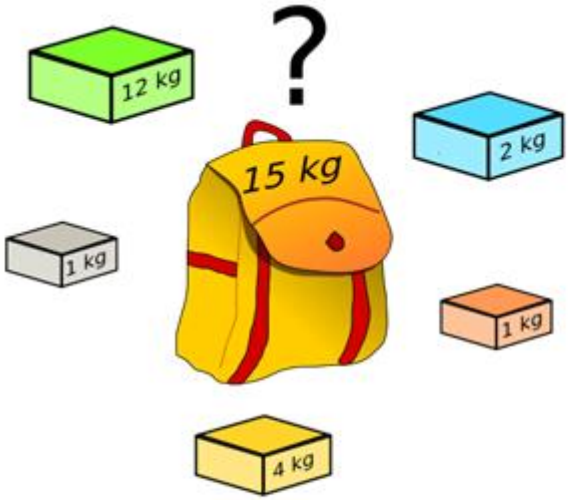
Karp's 21 NP-complete problems:
R. M. Karp, "Reducibility among combinatorial problems". *Complexity of Computer Computations*. pp. 85–103.

Dynamic Programming: False Start

- Optimization problem formulation
  - max $\Sigma_{i \in S} w_i$ ← objective function
    s.t. $\Sigma_{i \in S} w_i \leq W, S \subseteq \{1, \ldots, n\}$ ← constraints
- OPT(i) = the total weight of optimal solution for items 1,…, i
  - 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)
    - OPT(n) = OPT(n-1) (Optimal solution of {1, 2, …, n-1})
  - Case 2: $n \in O$ (OPT(n) counts w_n)
    - OPT(n) = w_n + OPT(n-1)

Q: What's wrong?

A: Accept item $n \Rightarrow$ For items {1, 2, …, n-1}, we have less capacity, $W - w_n$



Adding a New Variable

- Optimization problem formulation
  - $\max \Sigma_{i \in S} w_i$
    s.t. $\Sigma_{i \in S} w_i \leq W$, $S \subseteq \{1, \ldots, n\}$
- 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

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

```
Subset-sum($n$, $w_1,\ldots, w_n$, $W$)
for $w = 0, 1, \ldots, W$ do
    $M[0, w] = 0$
for $i = 0, 1, \ldots, n$ 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
            $M[i, w] = \max \{M[i-1, w], w_i + M[i-1, w-w_i]\}$
```

Example

```
Subset-sum($n, w_1,\dots, w_n, W$)
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:
$O(nW)$

$W + 1$ →

$n + 1$ ↓

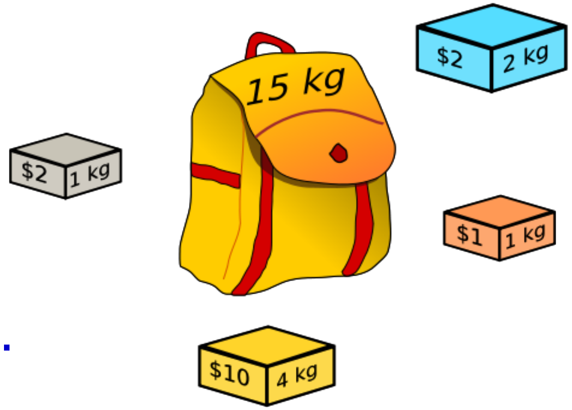
	0	1	2	3	4	5	6	7	8	9	10	11
∅	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	2	3	3	3	3	3	3	3	3	3
{ 1, 2, 3 }	0	1	2	3	3	5	6	7	8	8	8	8
{ 1, 2, 3, 4 }	0	1	2	3	3	5	6	7	8	9	9	11
{ 1, 2, 3, 4, 5 }	0	1	2	3	3	5	6	7	8	9	10	11

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. $\Sigma_{i \in S} w_i \leq W$, $S \subseteq \{1, \ldots, n\}$
- 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:

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

Traveling Salesman Problem

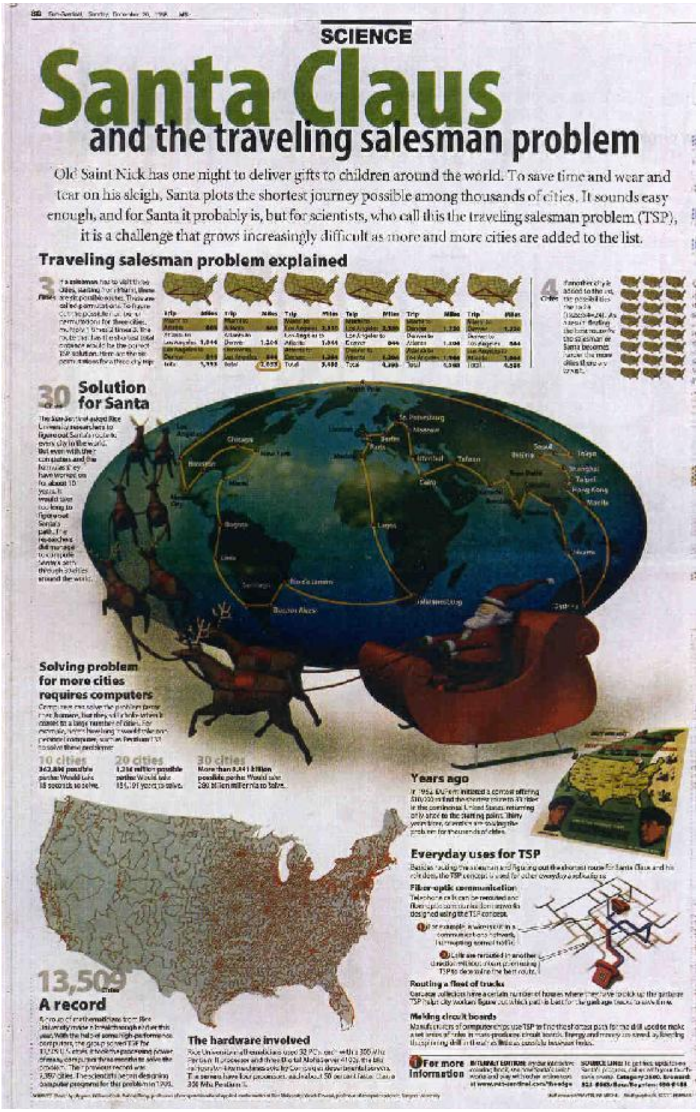
Richard E. Bellman, 1962

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

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

Traveling 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.math.uwaterloo.ca/tsp/
- 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

13,509 A record

Years ago

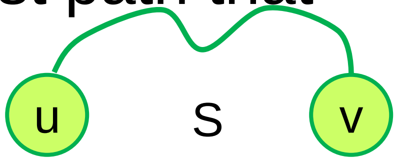
Everyday uses for TSP

The hardware involved

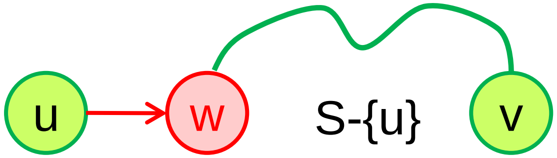
The Florida Sun-Sentinel, 20 Dec. 1998.

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) = $\min_{w \in S\text{-}\{u,v\}}\{d(u, w) + \text{OPT}(S\text{-}u, w, v)\}$



- 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!
  - Optimal substructure
  - Overlapping subproblems

Courtesy of Prof. Jeff Erickson @ UIUC

Summary: Algorithmic Paradigms

- Brute-force (Exhaustive search): 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
  - The first step of DP: define the subproblem!