Dynamic Programming

Fibonacci Numbers

$$fib(n) = fib(n-1) + fib(n-2)$$

- Write a program to find the nth fibonacci number.

```
def Fibonacci(n):
   if n == 0: return 0
   if n == 1: return 1
   return Fibonacci(n-1) + Fibonacci(n-2)
```

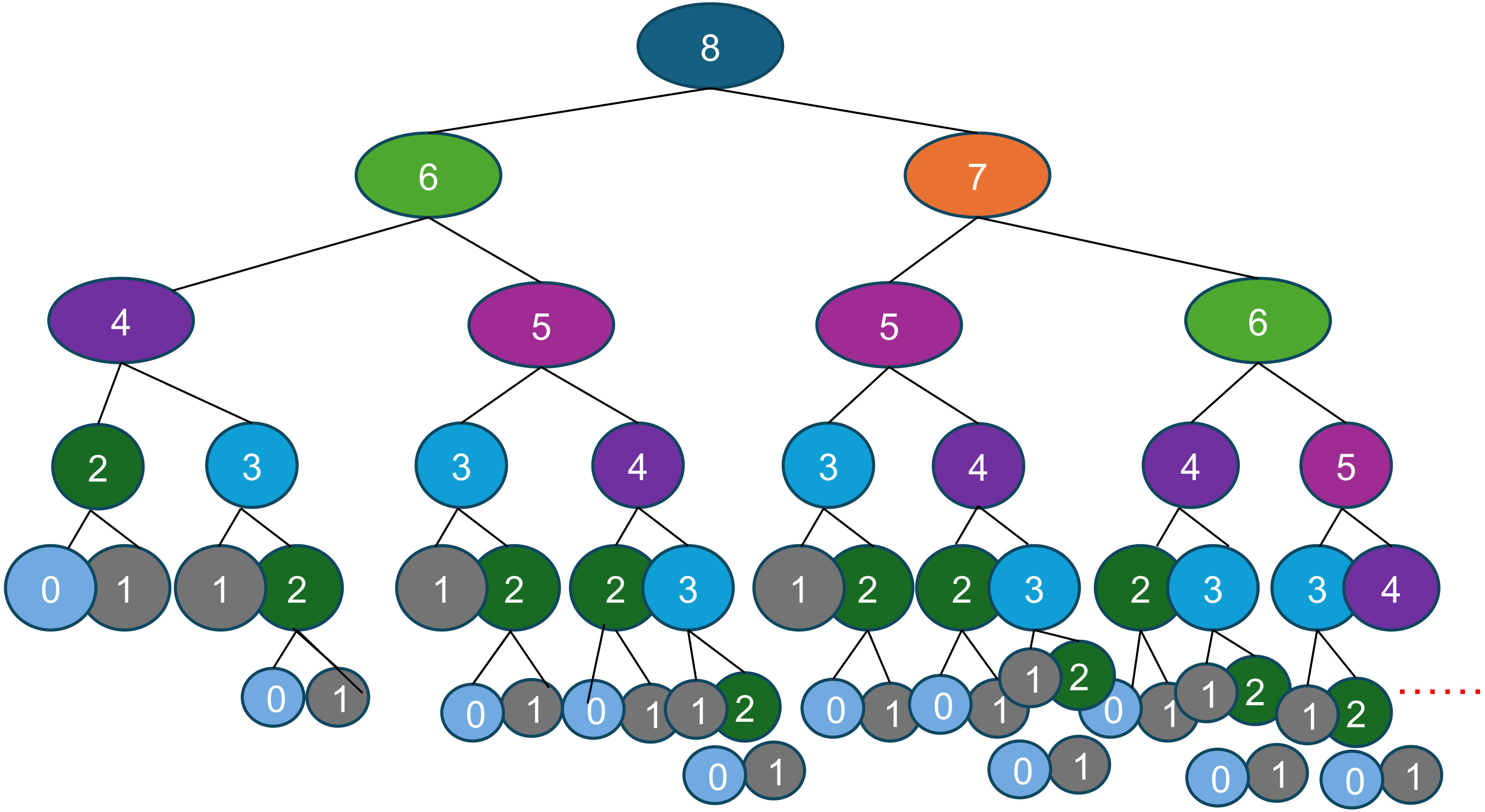
- Runtime analysis

$$T(n) = T(n-1) + T(n-2) + O(1)$$
$$T(n) = \Omega(2^{n/2})$$

What's going on?

Consider Fib(8)

That's a lot of repeated computation!



Fibonacci Numbers

- How to avoid repeated computations?
  - Store already computed results

```
global fib[n] initialized to -1/−∞/NIL

def Fibonacci(n):
   if n == 0: return 0
   if n == 1: return 1
   if fib[n] not computed:
      fib[n] = Fibonacci(n-1) + Fibonacci(n-2)
   return fib[n]
```

Fibonacci Numbers

- How to avoid repeated computations?
    - Start from smaller problems and proceed towards bigger ones.

```
def Fibonacci(n):
   local fib[n+1]
   fib[0] = 0
   fib[1] = 1
   for i -> 2 to n:
         fib[i] = fib[i-1] + fib[i-2]
   return fib[n]
```

What did we do?

Dynamic Programming!!!

Dynamic Programming

- It is an algorithm design paradigm
  - like divide-and-conquer is an algorithm design paradigm.
- Usually, it is for solving **optimization problems**
  - E.g., ***shortest*** path, **minimum/maximum** profit, **longest** sequences
  - (Fibonacci numbers aren’t an optimization problem, but they are a good example of DP anyway…)

Elements of dynamic programming

1. Optimal Sub-structure Property
    - Big problems break up into sub-problems.
    - The solution to a problem can be expressed in terms of solutions to smaller sub-problems.
        - Fibonacci:

$$fib(n) = fib(n-1) + fib(n-2)$$

Elements of dynamic programming

2. Overlapping Sub-Problem Property
    - The sub-problems overlap.
        - Fibonacci:
            - Both fib[i+1] and fib[i+2] directly use fib[i].
            - And lots of different fib[i+x] indirectly use fib[i].
    - This means that we can save time by solving a sub-problem just once and storing the answer.

Elements of dynamic programming

1. Optimal substructure.
   - Optimal solutions to sub-problems can be used to find the optimal solution of the original problem.
2. Overlapping subproblems.
   - The subproblems show up again and again

- Using these properties, we can design a **dynamic programming** algorithm:
  - Keep a table of solutions to the smaller problems.
  - Use the solutions in the table to solve bigger problems.
  - At the end we can use information we collected along the way to find the solution to the whole thing.

Two ways to think about and/or implement DP algorithms

- Top-down Approach with Memoization

```
global fib[n] initialized to -
1/—∞/NIL

def Fibonacci(n):
    if n == 0: return 0
    if n == 1: return 1
    if fib[n] not computed:
        fib[n] = Fibonacci(n-1) +
        Fibonacci(n-2)
    return fib[n]
```

- Bottom-up Approach

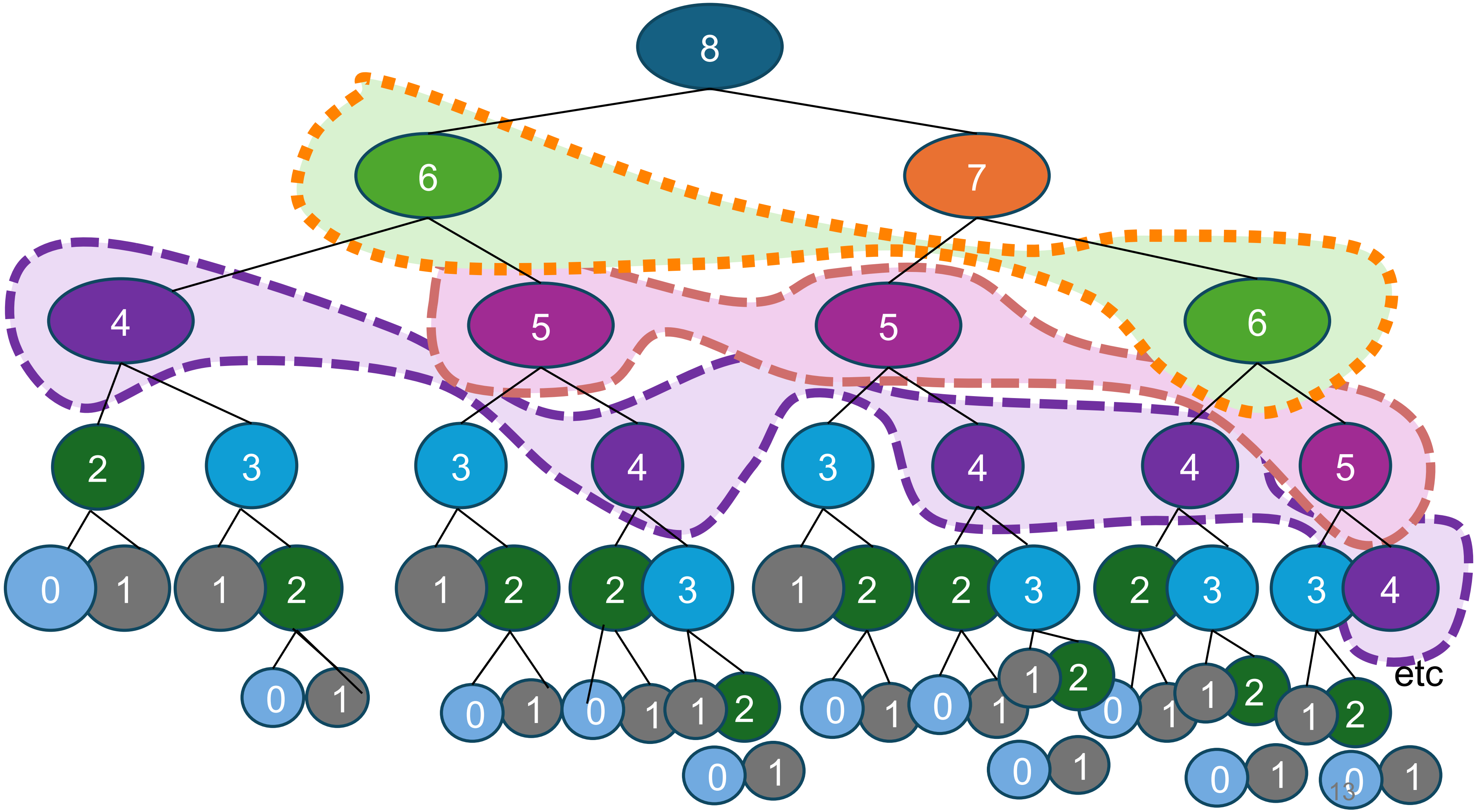
```
def Fibonacci(n):
    local fib[n+1]
    fib[0] = 0
    fib[1] = 1
    for i -> 2 to n:
          fib[i] = Fibonacci(i-1)
    +     Fibonacci(i-2)
    return fib[n]
```

Top-down Approach

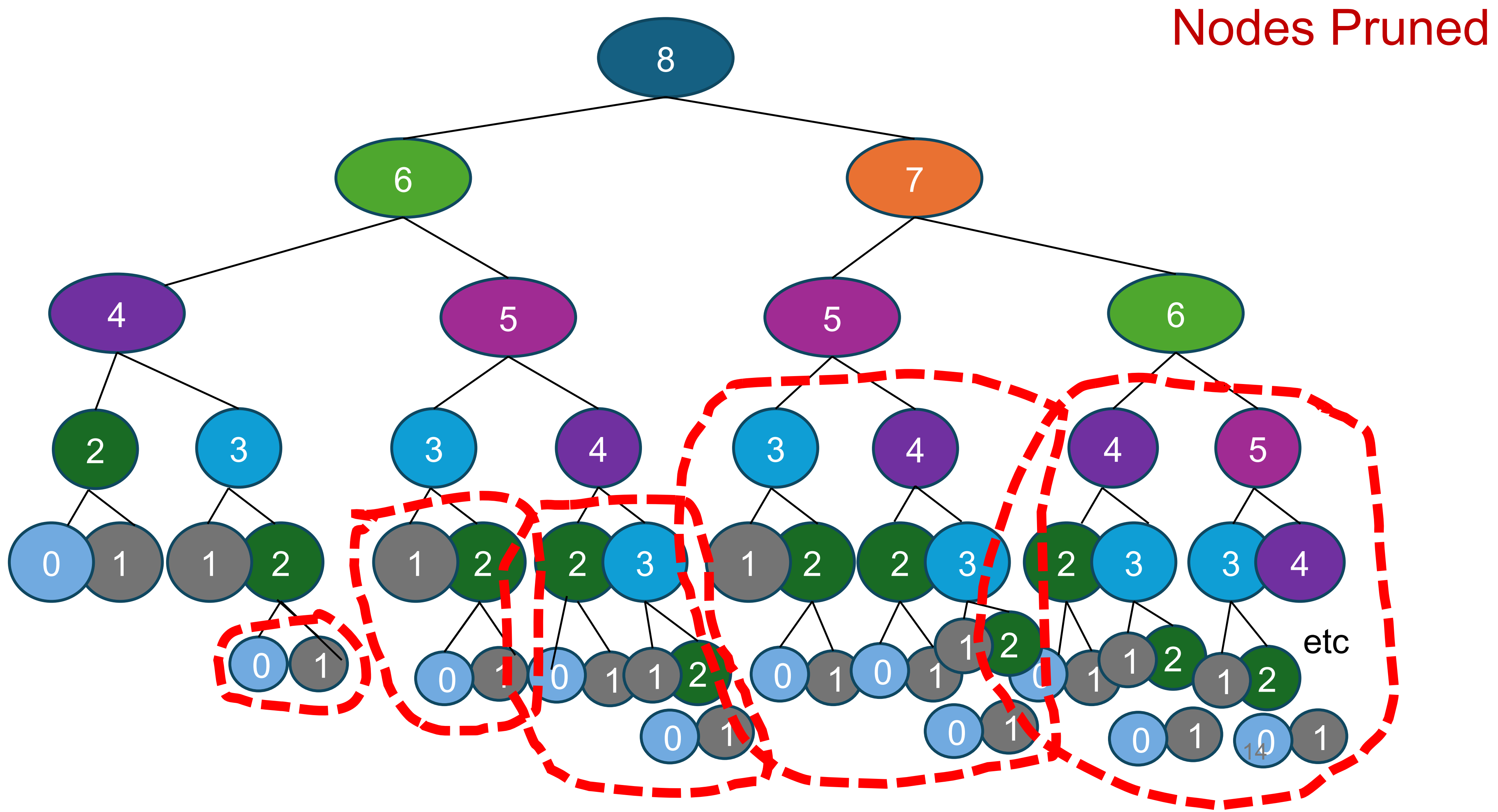
- Think of it like a recursive algorithm.
  - To solve the big problem:
    - Recurse to solve smaller problems
      - Those recurse to solve smaller problems
        - etc..

- The difference from divide and conquer:
  - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.
  - Aka, "**memo-ization**"

Top-down Approach

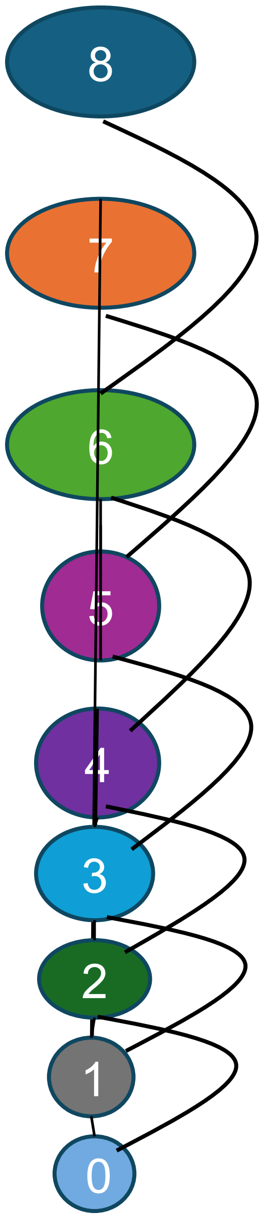


Top-down Approach



Bottom-up Approach

- Solve the small problems first
  - fill in fib[0],fib[1]
- Then bigger problems
  - fill in fib[2]
- …
- Then bigger problems
  - fill in fib[n-1]
- Then finally solve the real problem.
  - fill in fib[n]



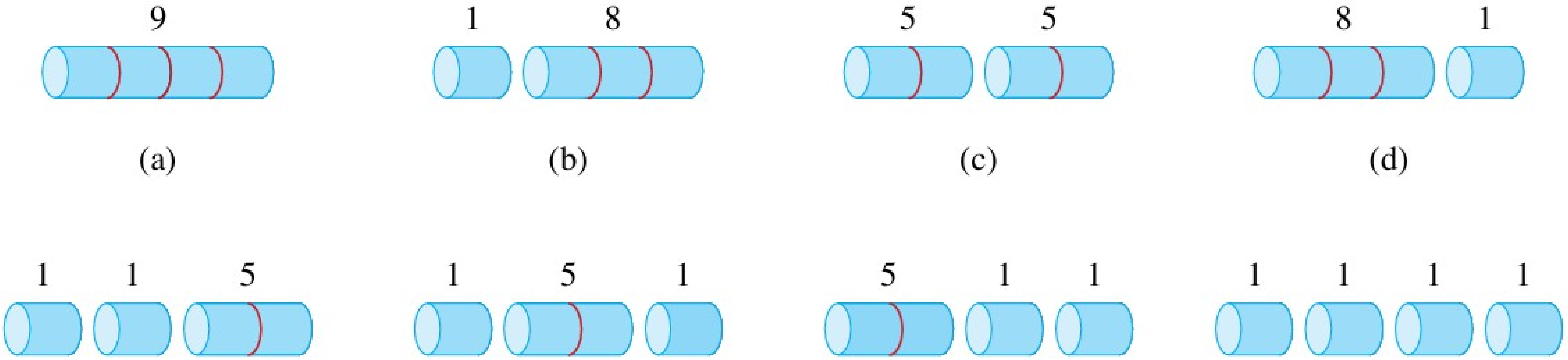
Rod-cutting Problem

- Given
  - A rod of length n
  - A table of prices p_i for $i = 1, 2, \dots, n$,
  - Determine the maximum revenue r_n obtainable by cutting up the rod and selling all the pieces.
  - For example,

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

Rod-cutting Problem

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



(a) (b) (c) (d)

(e) (f) (g) (h)

Rod-cutting Problem

- An optimal solution involving k pieces,
  - Each piece has length $i_1, i_2, i_3, \dots, i_k$
  - $n = i_1 + i_2 + i_3 + \cdots + i_k$
  - The optimal revenue, $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$

Rod-cutting Problem

- Once an initial cut is made,
  - The two resulting smaller pieces will be cut independently
  - Smaller instance of the rod-cutting problem
  - Optimal Sub-structure Property

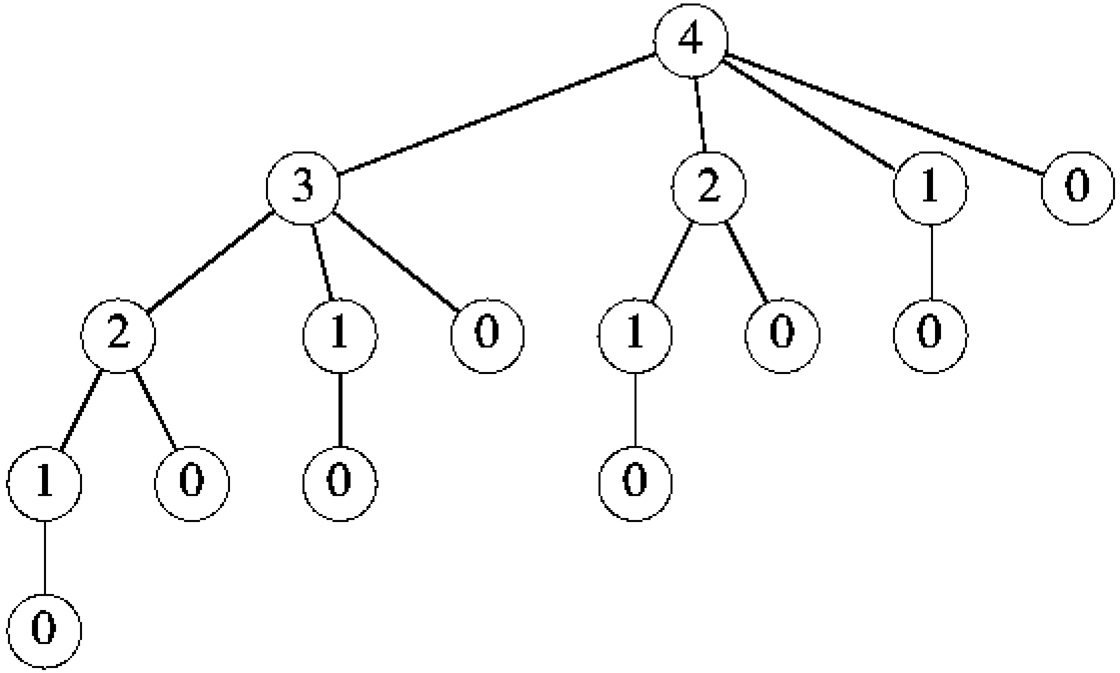
  - Different pieces can be cut into same length pieces (on not)
  - Overlapping Sub-structure Property

Rod-cutting Problem

- Assuming an initial cut is made,
  - $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots r_{n-1} + r_1)$
- Further assuming the initial cut is always the leftmost cut,
  - A first piece followed by some decomposition of the remainder
  - First piece is not further divided only the remaining is decomposed
  - $r_n = \max(p_i + r_{n-i} : 1 \leq i \leq n)$

Rod-cutting Problem

```
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
```



Runtime $O(2^{n-1})$

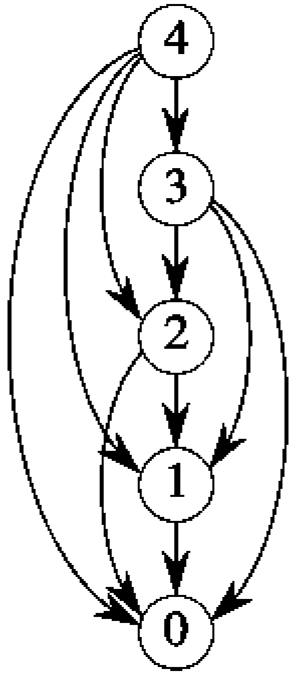
Rod-cutting Problem (Top-down Approach)

```
MEMOIZED-CUT-ROD(p, n)
let r[0 : n] be a new array          // will remember solution values in r
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0                 // already have a solution for length n?
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n          // i is the position of the first cut
        q = max {q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r)}
r[n] = q                    // remember the solution value for length n
return q
```

Rod-cutting Problem (Bottom-up Approach)

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0 : n] be a new array          // will remember solution values in r
r[0] = 0
for j = 1 to n                       // for increasing rod length j
    q = −∞
    for i = 1 to j                   // i is the position of the first cut
        q = max {q, p[i] + r[j − i]}
    r[j] = q                         // remember the solution value for length j
return r[n]
```



Runtime $O(n^2)$

Rod-cutting Problem (Bottom-up Approach)

- Reconstruct the choices that led to the optimal solution

- Store the choices that lead to the optimal solution
  - Store the optimal size `i` of the first piece to cut off when solving a subproblem of size `j`

Rod-cutting Problem (Bottom-up Approach)

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0 : n] and s[1 : n] be new arrays
r[0] = 0
for j = 1 to n                      // for increasing rod length j
    q = −∞
    for i = 1 to j                  // i is the position of the first cut
        if q < p[i] + r[j − i]
            q = p[i] + r[j − i]
            s[j] = i                // best cut location so far for length j
    r[j] = q                        // remember the solution value for length j
return r and s
```

Rod-cutting Problem (Bottom-up Approach)

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]          // cut location for length n
    n = n - s[n]        // length of the remainder of the rod
```

Matrix Chain Multiplication

- Given a chain of n matrices, $A_1, A_2, \ldots, A_n$
- Compute the product with standard multiplication algorithm
- Goal: Minimize the number of scalar multiplications

- Example,
  - A_1, A_2, A_3 with dimensions 10 * 100, 100 * 5, 5 * 50
  - $((A_1, A_2), A_3)$ perfoms a total of 7500 scalar multiplication
  - $(A_1, (A_2, A_3))$ perfoms a total of 75000 scalar multiplication

Matrix Chain Multiplication

- Parenthesizing resolves ambiguity in multiplication order
- Fully parenthesized chain of matrices
  - Either a single matrix
  - Or the product of two fully parenthesized matrix products, surrounded by parentheses
- Example,
  - $< A_1, A_2, A_3, A_4 >$ can be parenthesized in 5 distinct ways.
  - $(A_1, (A_2, (A_3, A_4)))$, $(A_1, ((A_2, A_3), A_4))$, $((A_1, A_2), (A_3, A_4))$, $((A_1, (A_2, A_3)), A_4)$, $(((A_1, A_2), A_3), A_4)$

Matrix Chain Multiplication

- Given a chain of n matrices, $A_1, A_2, \ldots, A_n$
- Matrix A_i has dimensions $p_{i-1} * p_i$
- Goal: Fully parenthesize the product to minimize the number of scalar multiplications

- Note: determine the order of multiplication not the product itself.

Matrix Chain Multiplication

- Fully parenthesized product
  - Split the product into fully parenthesized subproducts

- Let, the first split occurs between `kth` and `(k+1)st` matrices

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

- How many possible ways? $\Omega(2^n)$

Matrix Chain Multiplication

- Let, $m[i, j] \ = $ The minimum number of scalar multiplications needed to compute the matrix $A_{i:j}$

- We need to find $m[1, n]$

Matrix Chain Multiplication

- The optimal parenthesization
  - Split the product $A_{i:j}$ between A_k and A_{k+1} for some value of $i \leq k < j$

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} * p_k * p_j$$

  - We need to consider all such splits, i.e., all values of k

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

Matrix Chain Multiplication

- The optimal parenthesization
  - Split the product $A_{i:j}$ between A_k and A_{k+1} for some value of $i \leq k < j$

$$m[i,j] = m[i,k] + m[k,+1\ j] + p_{i-1} * p_k * p_j$$

  - We need to consider all such splits, i.e., all values of k

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

  - How many such $A_{i:j}$ subproblems? $\Theta(n^2)$

Matrix Chain Multiplication

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
if i == j
    return 0
m[i, j] = ∞
for k = i to j − 1
    q = RECURSIVE-MATRIX-CHAIN(p, i, k)
        + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
        + p_{i−1} p_k p_j
    if q < m[i, j]
        m[i, j] = q
return m[i, j]
```

Matrix Chain Multiplication

```
MATRIX-CHAIN-ORDER(p, n)
let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
for i = 1 to n                              // chain length 1
    m[i, i] = 0
for l = 2 to n                              // l is the chain length
    for i = 1 to n − l + 1                  // chain begins at A_i
        j = i + l − 1                       // chain ends at A_j
        m[i, j] = ∞
        for k = i to j − 1                  // try A_{i:k} A_{k+1:j}
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q                 // remember this cost
                s[i, j] = k                 // remember this index
return m and s
```

Matrix Chain Multiplication

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

j\i	1	2	3	4	5	6
1	0					
2	15750	0				
3	7875	2625	0			
4		4375	750	0		
5			2500	1000	0	
6				3500	5000	0

m[1, 3] = min(
- m[1,1] + m[2, 3] + 30*35*5 = 7875,
- m[1,2] + m[3, 3] + 30*15*5 = 18000,

)

Matrix Chain Multiplication

<table>
<tr><td>A_1</td><td>A_2</td><td>A_3</td><td>A_4</td><td>A_5</td><td>A_6</td><td>A_1</td><td>A_2</td><td>A_3</td><td>A_4</td><td>A_5</td><td>A_6</td><td>A_1</td><td>A_2</td><td>A_3</td><td>A_4</td><td>A_5</td><td>A_6</td><td>A_1</td><td>A_2</td><td>A_3</td><td>A_4</td><td>A_5</td><td>A_6</td><td>A_1</td><td>A_2</td><td>A_3</td><td>A_4</td><td>A_5</td><td>A_6</td><td>A_1</td><td>A_2</td><td>A_3</td><td>A_4</td><td>A_5</td><td>A_6</td></tr>
<tr><td>30*35</td><td>35*15</td><td>15*5</td><td>5*10</td><td>10*20</td><td>20*25</td><td>30*35</td><td>35*15</td><td>15*5</td><td>5*10</td><td>10*20</td><td>20*25</td><td>30*35</td><td>35*15</td><td>15*5</td><td>5*10</td><td>10*20</td><td>20*25</td><td>30*35</td><td>35*15</td><td>15*5</td><td>5*10</td><td>10*20</td><td>20*25</td><td>30*35</td><td>35*15</td><td>15*5</td><td>5*10</td><td>10*20</td><td>20*25</td><td>30*35</td><td>35*15</td><td>15*5</td><td>5*10</td><td>10*20</td><td>20*25</td></tr>
<tr><td colspan="6">30 * 35</td><td colspan="6">35 * 15</td><td colspan="6">15 * 5</td><td colspan="6">5 *10</td><td colspan="6">10 * 20</td><td colspan="6">20 * 25</td></tr>
</table>

j\i	1	2	3	4	5	6
1	0					
2	15750	0				
3	7875	2625	0			
4	9375	4375	750	0		
5	11875	7125	2500	1000	0	
6	15125	10500	5375	3500	5000	0

m[2, 5] = min(

- m[2,2] + m[3, 5] + 35*15*20 = 13000,
- m[2,3] + m[4, 5] + 35*5*20 = 7125,
- m[2,4] + m[5, 5] + 35*10*20 = 11375

)

Elements of dynamic programming (Revisit)

1. Optimal substructure.
    - Optimal solutions to sub-problems can be used to find the optimal solution of the original problem.

2. Overlapping subproblems.
    - The subproblems show up again and again

Optimal Substructure Property

- Solution to sub-problems are included in the optimal solution

- Rod-cutting
  - Solution to smaller pieces are also part of the solution to the entire rod

- Matrix Chain Multiplication
  - Solution to $A_{i:k}$ and $A_{k+1:j}$ is exactly include in the solution to $A_{i:j}$

- How to prove this?
  - Cut-and-paste

Longest Common Subsequence

- A strand of DNA consists of a string of molecules called bases
  - Adenine, Cytosine, Guanine, and Thymine
  - ACGT

- `S1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA`
- `S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA`

Longest Common Subsequence

- A strand of DNA consists of a string of molecules called bases
  - Adenine, Cytosine, Guanine, and Thymine
  - ACGT

- Given a sequence $X = < x_1, x_2, \dots, x_n >$
  - Another sequence $Z = < z_1, z_2, \dots, z_n >$ is a subsequence of X
  - If there exists a strictly increasing sequence $< i_1, i_2, \dots, i_n >$ indices of X such that for all $j = 1, 2, \dots, k,\ x_{i_j} = z_j$

Longest Common Subsequence

- A strand of DNA consists of a string of molecules called bases
  - Adenine, Cytosine, Guanine, and Thymine
  - ACGT

- Given two sequences X and Y
  - A sequence Z is a common sub-sequence if Z is a subsequence of both X and Y

- Goal: find a **maximum-length** common subsequence of X and Y.

Longest Common Subsequence

- Need to consider all subsequences

- How many subsequences of X?
  - 2^n

Longest Common Subsequence

- Given a sequence $X = < x_1, x_2, \dots, x_n >$
  - The ith prefix of X is $X_i = < x_1, x_2, \dots, x_i >$

Theorem 14.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any 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 $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is an LCS of X and Y_{n-1}.

Longest Common Subsequence

- Let $c[i, j]$ = the length of an LCS of the sequences X_i and Y_j

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

Longest Common Subsequence

```
LCS-LENGTH(X, Y, m, n)
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            // compute table entries in row-major order
    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
```

Longest Common Subsequence

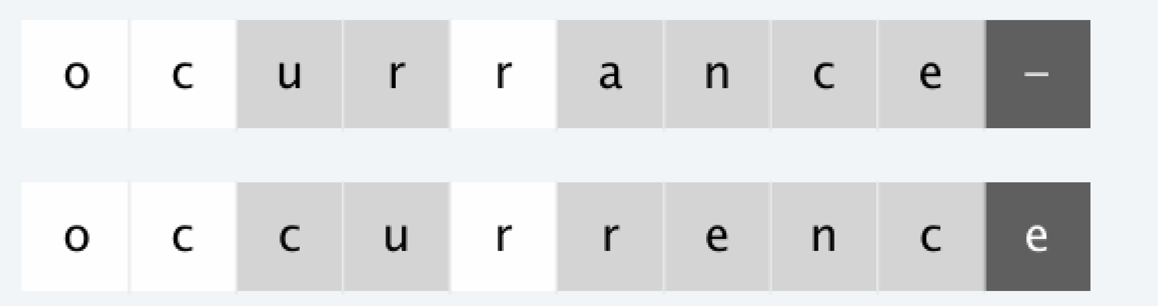
	j	0	1	2	3	4	5	6
i	x_i \ y_j		B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0	0	0	1		
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Longest Common Subsequence

	j	0	1	2	3	4	5	6
i	x_i \ y_j		B	D	C	A	B	A
0		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	$\boxed{0}$	1	$\boxed{2}$	2	$\boxed{3}$	3	4
7	B	0	$\boxed{1}$	2	2	3	4	4

Sequence Alignment

- How similar are two strings?
  - ocurrance vs occurrence



6 mismatches, 1 gap

1 mismatch, 1 gap

0 mismatches, 3 gaps

Sequence Alignment

- Edit distance
  - Gap penalty and mismatch penalty
  - Cost is sum of all penalties

Sequence Alignment

- Given two sequences
  - $x_1 x_2 \dots x_n$ and $y_1 y_2 \dots y_n$
  - An alignment is a set of ordered pairs (x_i, y_j) such that each character appears in at most one pair and there are no crossings.
    - Crossing: (x_i, y_j) and $(x_{i'}, y_{j'})$ cross if $i < i'$ but $j > j'$

o	c	u	r	r	a	n	c	e	–
o	c	c	u	r	r	e	n	c	e

6 mismatches, 1 gap

Sequence Alignment

- Given two sequences
  - $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_n$

- The cost of an alignment M is,

$$cost(M) = \sum_{(x_i, y_j) \in M} cost_{mismatch} + \sum_{\substack{x_i \\ unmatched}} cost_{gap} + \sum_{\substack{y_j \\ unmatched}} cost_{gap}$$

Sequence Alignment

- Given two sequences
  - $x_1 x_2 \dots x_n$ and $y_1 y_2 \dots y_n$

  - Goal: Find minimum cost alignment of the two sequences

Sequence Alignment

- Given two sequences
  - $x_1 x_2 \ldots x_n$ and $y_1 y_2 \ldots y_n$

  - $OPT(i, j) =$ minimum cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$

Sequence Alignment

- $OPT(i,j) =$ minimum cost of aligning $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_j$
  - Case 1: $OPT(i,j)$ matches (x_i, y_j)
    - Pay mismatch for (x_i, y_j) + min cost of aligning $x_1 x_2 \dots x_{i-1}$ and $y_1 y_2 \dots y_{j-1}$
    - $cost_{mismatch}$+ $OPT(i-1,\ j-1)$

6 mismatches, 1 gap

Sequence Alignment

- $OPT(i, j) =$ minimum cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$
  - Case 2: $OPT(i, j)$ leaves x_i unmatched
    - Pay gap for x_i + min cost of aligning $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_j$
    - $cost_{gap} + OPT(i-1, \ j)$

o	c	–	u	r	r	–	a	n	c	e
o	c	c	u	r	r	e	–	n	c	e

0 mismatches, 3 gaps

Sequence Alignment

- $OPT(i, j) =$ minimum cost of aligning $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_j$
  - Case 3: $OPT(i, j)$ leaves y_j unmatched
    - Pay gap for y_j + min cost of aligning $x_1 x_2 \dots x_i$ and $y_1 y_2 \dots y_{j-1}$
    - $cost_{mismatch}$+ $OPT(i, \; j-1)$

Sequence Alignment

- $OPT(i, j) =$ minimum cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$
  - Case 1: $OPT(i, j)$ matches (x_i, y_j)
    - $cost_{mismatch} + OPT(i-1,\ j-1)$
  - Case 2: $OPT(i, j)$ leaves x_i unmatched
    - $cost_{mismatch} + OPT(i-1,\ j)$
  - Case 3: $OPT(i, j)$ leaves y_j unmatched
    - $cost_{mismatch} + OPT(i,\ \ j-1)$

Sequence Alignment

- $OPT(i,j)$ = minimum cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_j$

$$OPT(i,j) = \begin{cases} j\delta & \text{if } i = 0 \\ i\delta & \text{if } j = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + OPT(i-1, j-1) \\ \delta + OPT(i-1, j) \\ \delta + OPT(i, j-1) \end{cases} & \text{otherwise} \end{cases}$$

Sequence Alignment

SEQUENCE-ALIGNMENT(m, n, x_1, …, x_m, y_1, …, y_n, δ, α)

FOR $i = 0$ TO m

  $M[i, 0] \leftarrow i\,\delta$.

FOR $j = 0$ TO n

  $M[0, j] \leftarrow j\,\delta$.

FOR $\mathrm{i} = 1$ TO m

  FOR $\mathrm{j} = 1$ TO n

$$M[i, j] \leftarrow \min \{\, \alpha_{x_i y_j} + M[i-1, j-1],\; \delta + M[i-1, j],\; \delta + M[i, j-1] \,\}.$$

already computed

RETURN $M[m, n]$.

0-1 Knapsack Problem

- Given,
  - n items where item i has value $v_i > 0$ and weighs $w_i > 0$
  - Value of a subset of items = sum of values of individual items.
  - Knapsack has weight limit of W
- Goal. Pack knapsack so as to maximize total value of items taken.
  - Example, {1, 2, 5} yields 35$ while {3, 4} yields 40$.

i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

knapsack instance
(weight limit W = 11)

0-1 Knapsack Problem

- $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$, subject to weight limit w
- Goal: Find $OPT(n, W)$

0-1 Knapsack Problem

- $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$, subject to weight limit w

- Case 1: Don't pick item i
  - $OPT(i, w)$ selects best of $\{1, 2, \ldots, i - 1\}$ subject to weight limit w

- Case 2: Pick item i
  - $OPT(i, w)$ selects best of $\{1, 2, \ldots, i - 1\}$ subject to weight limit $w - w_i$
  - Collect value v_i

0-1 Knapsack Problem

- $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$, subject to weight limit w

- Case 1: Don't pick item i
  - $OPT(i, w) = OPT(i-1, w)$

- Case 2: Pick item i
  - $OPT(i, w) = v_i + OPT(i-1, w)$

0-1 Knapsack Problem

- $OPT(i, w)$ = optimal value of knapsack problem with items $1, \ldots, i$, subject to weight limit w

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

0-1 Knapsack Problem

```
KNAPSACK(n, W, w1, ..., wn, v1, ..., vn )
_________________________________________

FOR w = 0 TO W
   M[0, w] ← 0.
                                              previously computed values
FOR i = 1 TO n
   FOR w = 0 TO W
      IF (wi > w)  M[i, w] ← M[i–1, w].
      ELSE         M[i, w] ← max { M[i–1, w],  vi + M[i–1, w – wi] }.

RETURN M[n, W].
_________________________________________
```

0-1 Knapsack Problem

weight limit w

subset of items 1, …, i		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	6	7	7	7	7	7	7	7	7	7
	{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
	{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
	{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = optimal value of knapsack problem with items 1, …, i, subject to weight limit w

Reference

- Dynamic Programming
  - CLRS 4th Ed. Chapter 14 (14.1 – 14.4)
  - KT Sections 6.4 (The Knapsack Problem), 6.6