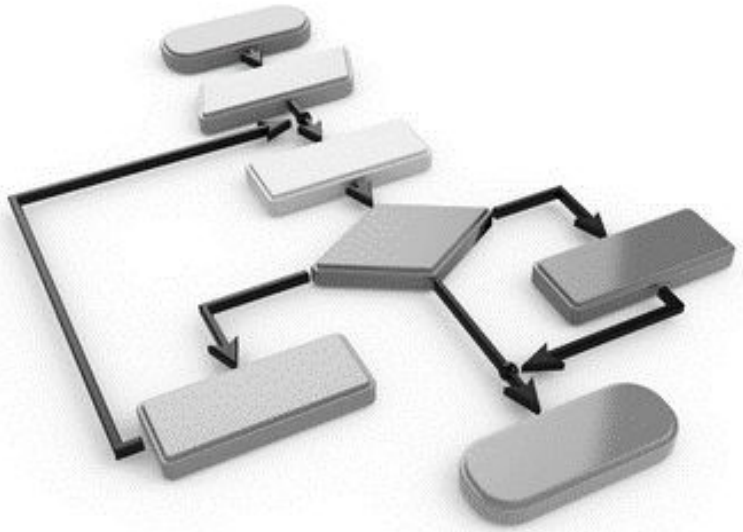


Dynamic Programming

CSE 301: Combinatorial Optimization

Fibonacci Numbers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

Computing the n^{th} Fibonacci number recursively:

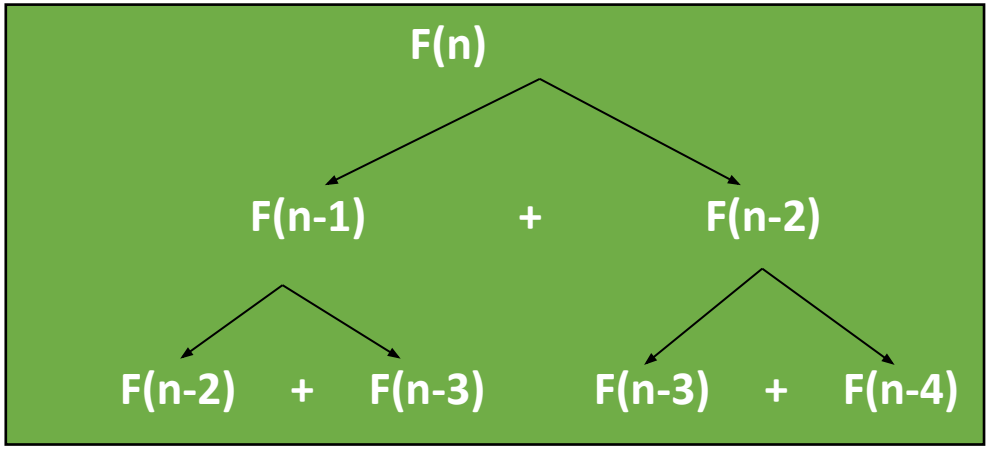
F(n) = F(n-1) + F(n-2), when n>1

F(0) = 0

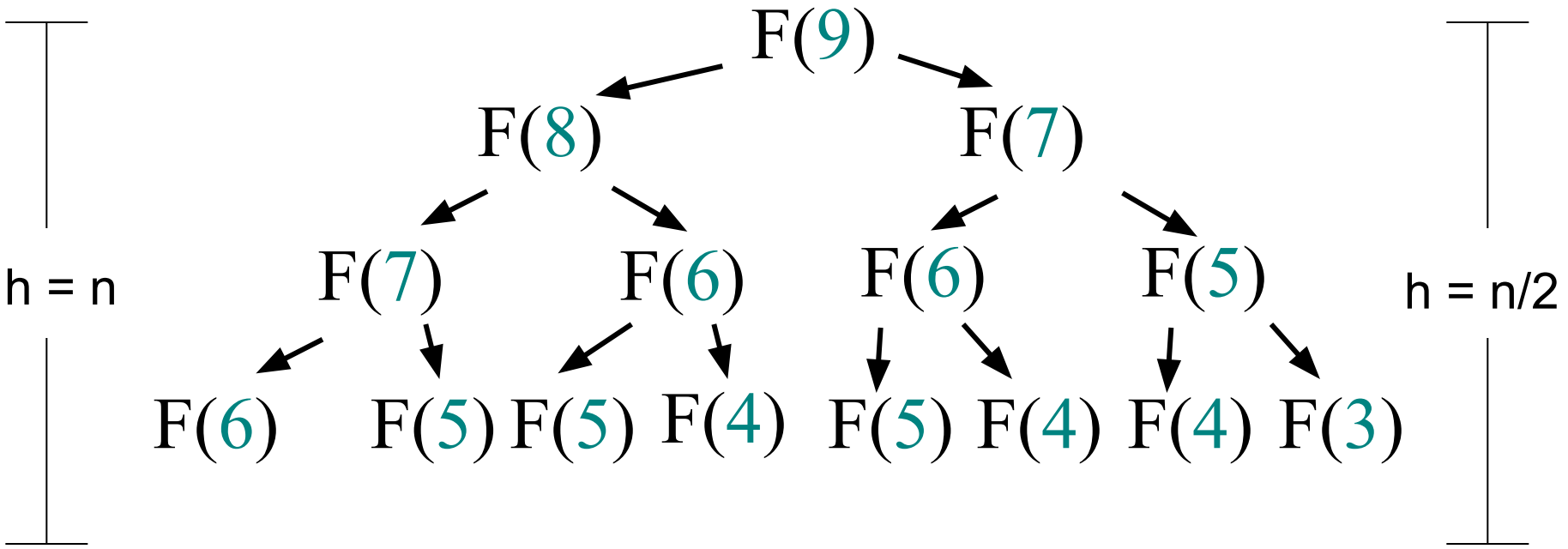
F(1) = 1

Top-down (recursive) algorithm

```
Algorithm Fibo(n):
    if n ≤ 1 then
        return n
    else
        return Fibo(n-1)+Fibo(n-2)
```



Time Complexity of Top-Down Algorithm

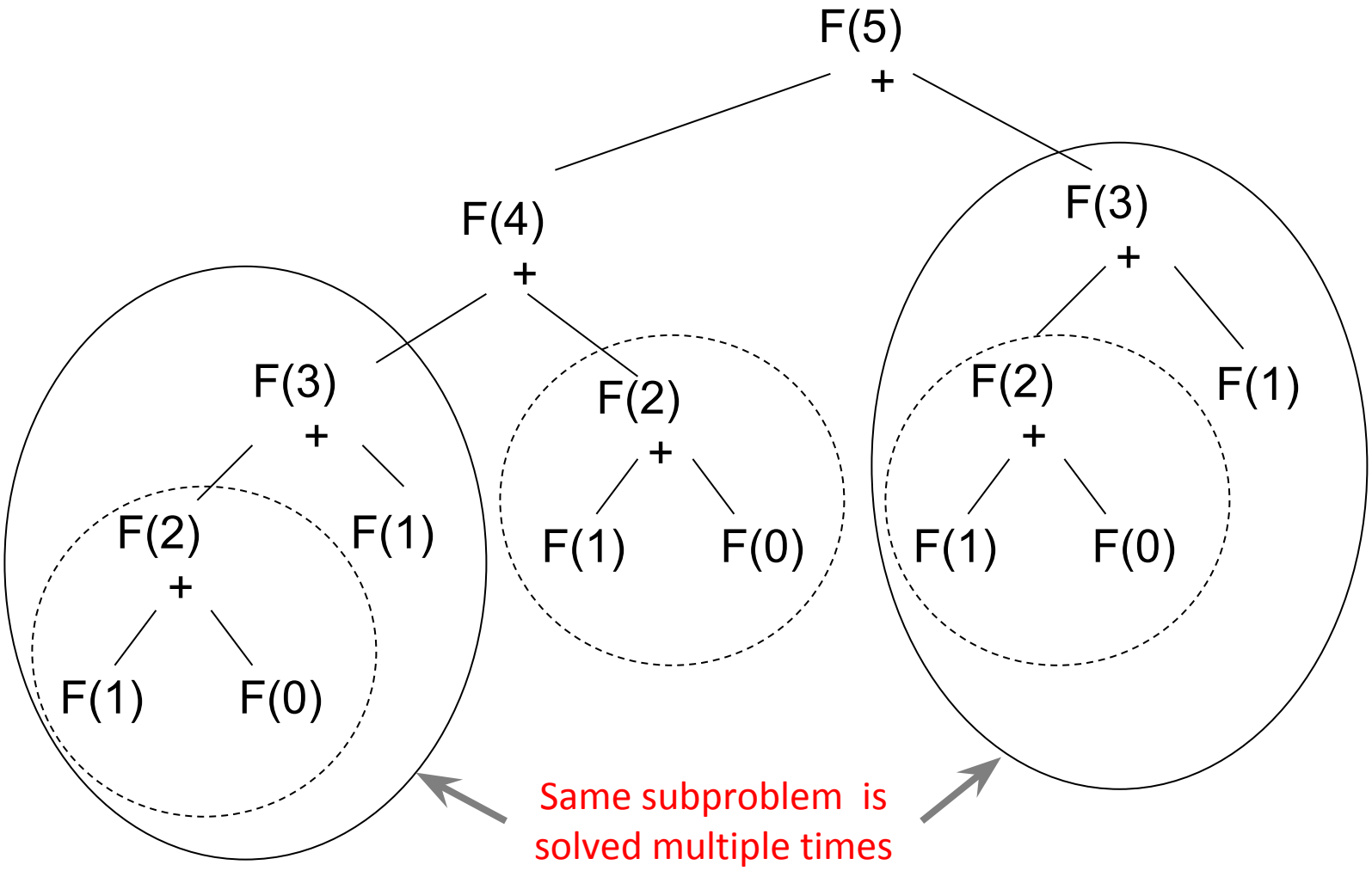


Time complexity between $2^{n/2}$ and 2^{n}

Recursion Tree of Top-Down Algorithm

Why is the top-down approach so inefficient?

Recomputes many sub-problems (*a.k.a. overlapping subproblem*).



A Top-down Algorithm that memorizes previous solutions (memoized top-down approach)

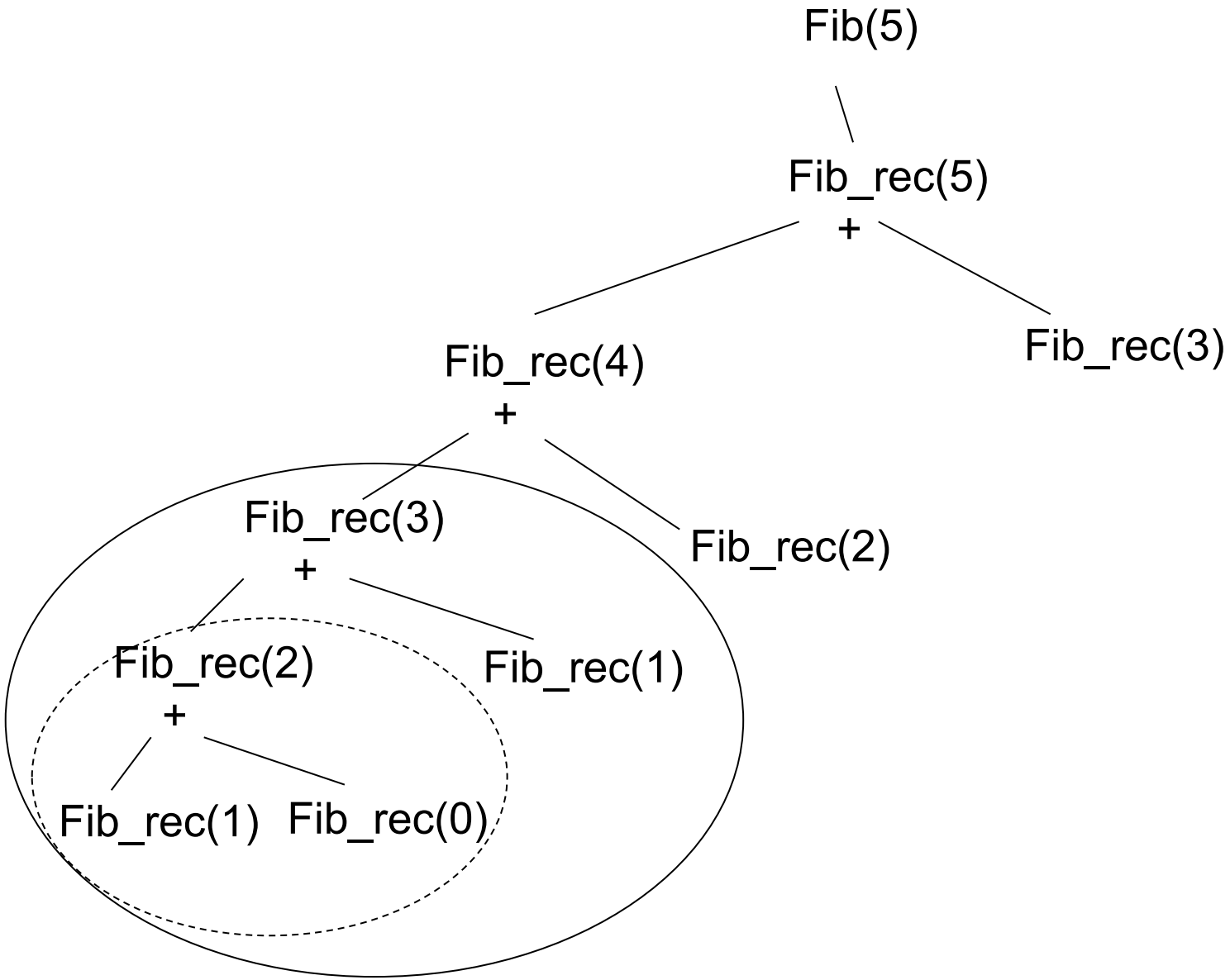
```
Algorithm Fib(n):
    for i = 0 to n
        F[i] = -∞
    Fib_rec(n)

Algorithm Fib_rec(n):
    if F[n]≥0 then
        return F[n]
    if n ≤ 1 then
        F[n] = n
    else
        F[n] = Fib_rec(n-1)+Fib_rec(n-2)
    return F[n]
```

```
//Non-memoized Top-Down Approach
Algorithm Fibo(n):
    if n ≤ 1 then
        return n
    else
        return Fibo(n-1)+Fibo(n-2)
```

Time complexity?

Each of the F[i] values (i={0, 1,…, n}) is computed only once. So time is *O(n).*

Recursion Tree of Memoized Algorithm



A Bottom-up (Iterative) Algorithm

```
Algorithm fib_bu(n):
    for i = 0 to n
        if i ≤ 1 then
        F[i] = i
       else
        F[i] = F[i-1] + F[i-2]
    return F[n]
```

Time complexity?

The for loop here runs O(n) times and all the other (first and last) statements take O(1) time. So total time is O(n).

Summery

Problem with the recursive Fib algorithm:

- Each subproblem was solved for many times!

Solution: avoid solving the same subproblem more than once

- (1) pre-compute all subproblems that may be needed later (bottom-up)
- (2) Compute on demand, but memorize the solution to avoid recomputing (memoized top-down)

Can you always speedup a recursive algorithm by memoization (memoized top-down) or pre-computation (bottom-up)?

- E.g., merge sort
- No. since there is no overlap between the two sub-problems

Dynamic Programming

- Dynamic Programming (DP) is an algorithm design technique for ***optimization problems***.

- Like divide and conquer (D&C), DP solves problems by combining solutions to sub-problems.

- Unlike D&C, sub-problems are not independent.
  - Sub-problems may share sub-sub-problems

Dynamic Programming

The term Dynamic Programming comes from Control Theory, not computer science. Programming refers to the use of tables (arrays) to construct a solution.

In dynamic programming we usually reduce time by increasing the amount of space

We solve the problem by solving sub-problems of increasing size and saving each optimal solution in a table (usually).

The table is then used for finding the optimal solution to larger problems.

Time is saved since each sub-problem is solved only once.

Designing a DP Algorithm

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom up fashion.
4. Construct an optimal solution from computed information.

Example: *NSU Number*

Almost like Fibonacci number; with a simple difference: we take the summation of last three NSU numbers, instead of last two.

Assuming A(n) represents the n-th NSU number,

A(n) = n , if n <= 2

= A(n-1) + A(n-2) +A(n-3) , otherwise

A(0) = 0

A(1) = 1

A(2) = 2

A(3) = 0+1+2 = 3

A(4) = 1+2+3 = 6

A(5) = 2+3+6 = 11

….

Design (i) a memoized top-down algorithm and (ii) a bottom-up algorithm to compute A(n).

Show that the problem of computing A(n) satisfies overlapping subproblem property.

Example: *n* choose *r* (combinations)

Given *n* things, how many different sets of size *r* can be chosen?

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}, 1 < r < n \, (recursive)$$

with base cases:

$$\binom{n}{1} = n \;\; and \;\; \binom{n}{n} = 1$$

Assuming C(n,r) represents the value of n choose r,

C(n,r) = n , if r =1

= 1 , if r = n or r = 0

= C(n-1,r-1) + C(n-1, r) , otherwise

Combinations: Top Down Algorithm

Algorithm Comb(n,r):

```
if(r == 1)  // base case 1
   return n
else if r = n or r = 0  // base case 2
   return 1
else //general case
   return Comb(n-1, r-1)+Comb(n-1, r)
```

Top Down Memoized Algorithm

```
Algorithm Comb(n,r):
   for i=0 to n
     for j=0 to min(i,r)
           C[i,j] = -∞
   return rec(n,r)

Algorithm rec(n,r):
   if C[n,r] != -∞ then
      return C[n,r]
      if r = 1 then  // base case 1
           C[n,r] = n
      else if r = n or r = 0 then  // base case 2
         C[n,r] = 1
      else //general case
      C[n,r] = rec(n-1, r-1)+rec(n-1, r)
   return C[n,r]
```

Bottom-up Algorithm

Algorithm Combination (n,r):

```
for i = 0 to n
  for j = 0 to min(i,r)
   if j = 1 then
          C[i,j] = i
     else if j = i or j = 0 then
        C[i,j] = 1
     else                    //general case
      C[i,j] = C[i-1, j-1]+C[i-1, j]

return C[n,r]
```

Time Complexity:

Θ (nr), for both memorized top-down & bottom-up algorithms

Example: Rod Cutting

You are given a rod of length $n \geq 0$ (n in inches)

A rod of length i inches will be sold for p_i dollars

Cutting is free (simplifying assumption)

Problem: given a table of prices p_i determine the maximum revenue r_n obtainable by cutting up the given rod (of length n) and selling the pieces.

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

Example: Rod Cutting

Problem: given a table of prices p_i determine the maximum revenue r_n obtainable by cutting the given rod (of length *n)* and selling the pieces.

Can we use a greedy algorithm (like fractional knapsack) that always takes the length with highest price/length value?

Length i	1	2	3	4
Price p_i	1	20	33	36
Value p_i/i	1	10	11	9

No, for e.g. by greedy algorithm, optimal revenue of a 4 inch rod should be 33+1=34 which we get by cutting a rod of length 3 first (which has the maximum value of 11) and then taking the remaining rod of length 1 (i.e., the cutting lengths here are in order: (3,1)).

However, the optimal cutting is: (2,2) which gives us optimal revenue of 20+20 = 40.

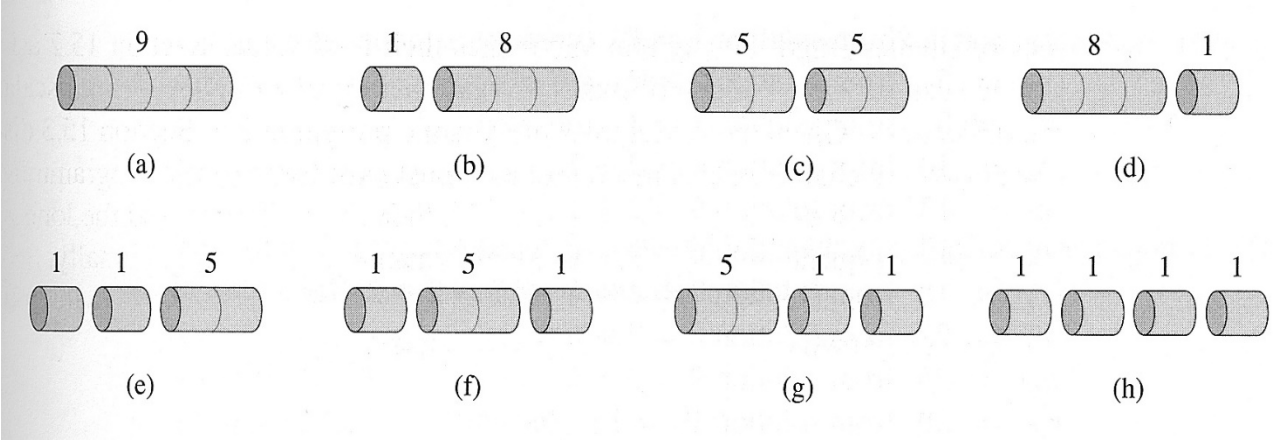
Example: Rod Cutting

Can we use a brute-force/exhaustive algorithm that tries all possible cuts of a rod of length n?

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

Question: in how many different ways can we cut a rod of length n?

For a rod of length 4:



$2^{4-1} = 2^3 = 8$

Yes but the cost would be too high!!! There are 2^{n-1} possible ways to cut a rod of length n (**Exponential**). We cannot try all possibilities for "large" n; the exhaustive approach isn't practically feasible.

Example: Rod Cutting

Let us find a way to solve the problem recursively:

Let r_n = the maximum revenue obtainable from a rod of length n

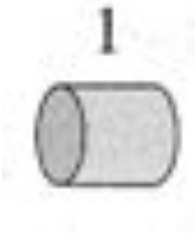
How can we construct a recurrence relation for r_n?

Advice: when you don't know what to do next, start with a simple example and hope that some idea will click…

Example: 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

Maximum revenue obtainable by cutting a rod of length n, r_n = ? (for n = 1)



$r_1 = p_1 = 1$

Example: 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

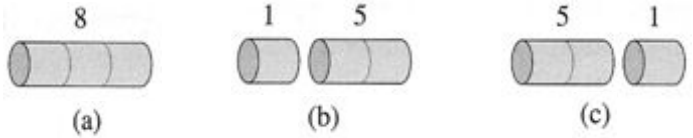
Maximum revenue obtainable by cutting a rod of length n, r_n = ? (for n = 2)

$r_2 = \max(p_2, p_1+r_1) = \max(5, 1+1) = 5$

Example: 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

Maximum revenue obtainable by cutting a rod of length n, r_n = ? (for n = 3)

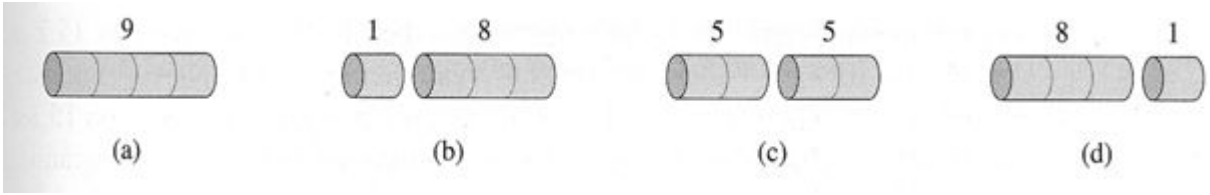


$r_3 = \max(p_3, p_1+r_2, p_2+r_1) = \max(8, 1+5, 5+1) = 8$

Example: 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

Maximum revenue obtainable by cutting a rod of length n, r_n = ? (for n =4)



$r_4 = \max(p_4, p_1+r_3, p_2+r_2, p_3+r_1) = \max(9, 1+8, 5+5, 8+1) = 10$

Example: Rod Cutting

In general, for any n:

$r_n = max(p_n, p_1 + r_{n-1}, p_2 + r_{n-2}, p_3 + r_{n-3}, \ldots, p_{n-1} + r_1)$

$\Rightarrow r_n = max(p_n + r_0, p_1 + r_{n-1}, p_2 + r_{n-2}, p_3 + r_{n-3}, \ldots, p_{n-1} + r_1)$ *:[Let, $r_0 = 0$]*

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

In other words,

- maximal revenue r_n is obtained by cutting the rod into smaller pieces of lengths: *i* and *(n-i)*, for some value of *i* (for which $\boldsymbol{p_i + r_{n-i}}$ is maximum) where $1 \le i \le n$ and
- $r_i = p_i$, *i.e.,* the piece of length *i* need no more cut (because cutting it into smaller pieces will not increase its revenue) whereas
- the piece of length *(n-i)* may need more cut; but we have already calculated the maximal revenue, r_{n-i} of a rod of length *(n-i)* before calculating r_n. We can use that value (r_{n-i}) to calculate r_n

Optimal Substructure Property of Rod Cutting Problem

- The recurrence relation: $r_n = max_{1 \le i \le n}(p_i + r_{n-i})$ shows that the Rod Cutting problem has optimal substructure property:

an optimal solution to the problem (here r_n) can be calculated using optimal solutions to its sub problems (here r_{n-i}).

This recurrence relation can be implemented as a simple top-down recursive procedure

Top Down algorithm for Rod Cutting

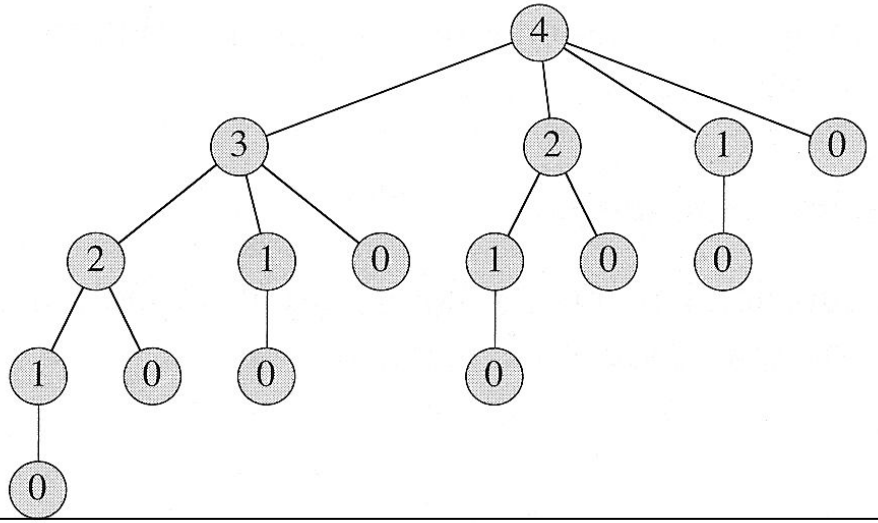
```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Computing the recursion leads to re-computing some numbers (**overlapping subproblems**) again and again – how many?

Overlapping subproblem property of Rod Cutting Problem

Let's call Cut-Rod(p, 4), to see the effects on a simple case:

```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = −∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n − i))
6  return q
```



T(n) = total # of calls made to CUT-ROD for an initial call of CUT-ROD(p,n)
= The # of nodes for a recursion tree corresponding to a rod of size n = ?

Solution: CUT-ROD(p,n) calls CUT-ROD(*p, n-i*) for *i = 1,2,3, …, n, i.e.,* CUT-ROD(p,n) calls CUT-ROD(*p, j*) for *j = 0,1,2, …, n-1*

$$T(0) = c, T(n) = c + \sum_{j=0}^{n-1} T(j) = O(2^n), n \geq 1.$$

How to decrease time?

We have a problem: "reasonable size" problems are not solvable in "reasonable time" (but, in this case, they are solvable in "reasonable space").

Specifically:

- Note that navigating the whole tree requires 2^n function calls
- However, no more than n different values need to be computed or used.

We exploit these observations to come up with two ways to decrease time:

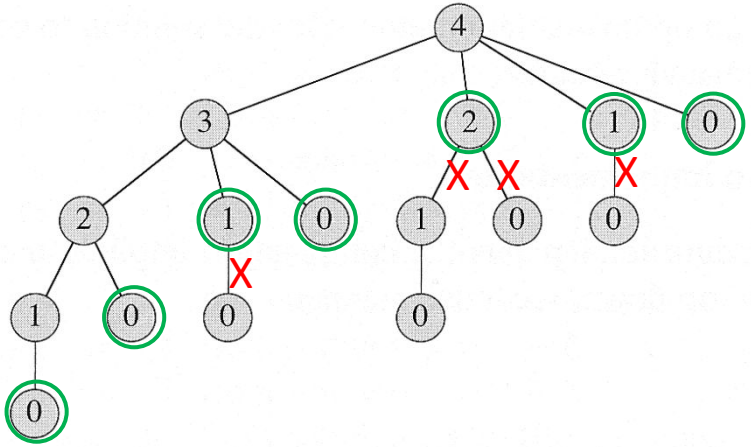
- **Memoized Top-down algorithm:** store the value of r_k (once computed by a call of CUT-ROD) in a table and reuse it in later calls of CUT-ROD as needed. This technique allows us to compute r_k only once.

*The technique of storing and reusing values computed by a previous call of a recursive function is called "**memoizing**" (i.e., writing yourself a memo).*

- **Bottom Up Algorithm**: Compute small subproblems first, then gradually solve larger and larger subproblems by using the pre-computed solutions of smaller subproblems. For e.g., for rod-cutting problem: compute $r_0, r_1, r_2, \ldots, r_n$

Memoized Top-down Algorithm for Rod-Cutting Problem

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```




```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

Memoized Top-down Algorithm for Rod-Cutting Problem

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

Time: $\Theta(n^2)$

Each r[i], for *i = 1,2,3, ..., n,* is computed only once and this computation takes $\Theta(n)$ time (due to the for loop in lines 6-7).

As we compute n values of r[i] (for i=1,2,…,n), total time is $\Theta(n^2)$

Bottom Up Algorithm

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4       q = -∞
5       for i = 1 to j
6            q = max(q, p[i] + r[j - i])
7       r[j] = q
8  return r[n]
```

Bottom-up approach also takes $\Theta(n^2)$ time.
Why??
The most heavily executed statement in bottom-up approach is line 6 which is executed j times for each value of $j \varepsilon \{1, 2, \ldots, n\}$. So it is executed $1+2+\ldots+n = \Theta(n^2)$ times.

Simulation: Rod Cutting

Length i	1	2	3	4	5	6	7
Price p_i	1	5	8	9	10	12	17

We begin by constructing (by hand) the optimal solutions for i = 0, 1, …, 10:

$r_0 = 0$

$r_1 = p_1 = 1$ (no cuts)

$r_2 = \max(\mathbf{p_2}, p_1+r_1) = \max(\mathbf{5},1+1) = \mathbf{5}$ (no cuts)

$r_3 = \max(\mathbf{p_3}, p_2+r_1, p_1+r_2) = \max(\mathbf{8},5+1,1+5) = \mathbf{8}$ (no cuts)

$r_4 = \max(p_4, p_3+r_1, \mathbf{p_2+r_2}, p_1+r_3) = \max(9, 8+1, \mathbf{5+5}, 1+8) = \mathbf{10}$

$r_5 = ?$

$r_6 = ?$

$r_7 = ?$

Simulation: Rod Cutting

Length i	1	2	3	4	5	6	7
Price p_i	1	5	8	9	10	12	17

We begin by constructing (by hand) the optimal solutions for i = 1, …, 10:

$r_0 = 0$

$r_1 = \mathbf{p_1 = 1}$ (no cuts)

$r_2 = \max(\mathbf{p_2}, p_1+r_1) = \max(\mathbf{5}, 1+1) = \mathbf{5}$ (no cuts)

$r_3 = \max(\mathbf{p_3}, p_2+r_1, p_1+r_2) = \max(\mathbf{8}, 5+1, 1+5) = \mathbf{8}$ (no cuts)

$r_4 = \max(p_4, p_3+r_1, \mathbf{p_2+r_2}, p_1+r_3) = \max(9, 8+1, \mathbf{5+5}, 1+8) = \mathbf{10}$

$r_5 = \max\{p_5, p_4+r_1, p_3+r_2, \mathbf{p_2+r_3}, p_1+r_4\}$

$= \max(10, 9+1, 8+5, \mathbf{5+8}, 1+10) = \mathbf{13}$

$r_6 = \max\{p_6, p_5+r_1, p_4+r_2, \mathbf{p_3+r_3}, p_2+r_4, p_1+r_5\}$

$= \max\{12, 11, 14, \mathbf{16}, 15, 14\} = 16$

$r_7 = \ldots = \mathbf{p2 + r5} = 5 + 13 = 18$

Now we know maximum revenue obtainable from a 7 inch rod;
but we don't know how to cut the rod to get that much revenue

Reconstructing a Solution

```
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 = -∞
    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
```

Save cutting points

Reconstructing a Solution

```
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]
```

Simulation: Rod Cutting

Length i	1	2	3	4	5	6	7
Price p_i	1	5	8	9	10	12	17
s_i	1	2	3	2	2	3	2

We begin by constructing (by hand) the optimal solutions for i = 1, …, 10:

$r_0 = 0$

$r_1 = \mathbf{p_1 = 1}$ (no cuts)

$r_2 = \max(\mathbf{p_2}, p_1+r_1) = \max(\mathbf{5}, 1+1) = \mathbf{5}$ (no cuts)

$r_3 = \max(\mathbf{p_3}, p_2+r_1, p_1+r_2) = \max(\mathbf{8}, 5+1, 1+5) = \mathbf{8}$ (no cuts)

$r_4 = \max(p_4, p_3+r_1, \mathbf{p_2+r_2}, p_1+r_3) = \max(9, 8+1, \mathbf{5+5}, 1+8) = \mathbf{10}$

$r_5 = \max\{p_5, p_4+r_1, p_3+r_2, \mathbf{p_2+r_3}, p_1+r_4\}$

$= \max(10, 9+1, 8+5, \mathbf{5+8}, 1+10) = \mathbf{13}$

$r_6 = \max\{p_6, p_5+r_1, p_4+r_2, \mathbf{p_3+r_3}, p_2+r_4, p_1+r_5\}$

$= \max\{12, 11, 14, \mathbf{16}, 15, 14\} = 16$

$r_7 = \mathbf{p2 + r5} = 5 + 13 = 18$

Simulation: Rod Cutting

Length i	1	2	3	4	5	6	7
Price p_i	1	5	8	9	10	12	17
s_i	1	2	3	2	2	3	2

Optimal Cutting lengths of a rod of length 7:

$s_7 = 2$

$\Rightarrow$ 2, optimal cutting of a rod of remaining length (7-2) is: $s_{7-2} = s_5$

$\Rightarrow$ $s_5 = 2$, optimal cutting of a rod of remaining length (5-2) is: $s_{5-2} = s_3$

$\Rightarrow$ $s_3 = 3$, remaining length = 3-3 = 0; so stop

Therefore optimal cutting lengths of a rod of length 7 is: 2,2,3

Practice: Find optimal cutting and max. revenue for this rod cutting problem

Length i	1	2	3	4	5	6	7	8
Price p_i	1	5	8	9	10	12	17	19
s_i								

r8 = max (p8, p7+r1, p6+r2, p5+r3, p4+r4, p3+r5, p2+r6, p1+r7)
= max(19, 17+1, 12+5, 10+8, 9+10, 8+13, 5+16, 1+18)
= max(19, 18, 17, 18, 19, 21, 21, 19) = 21

Multiple Rod Cutting

You are given m rods of different lengths

A rod of length *i* inches will be sold for p_i dollars

Problem: given a table of prices p_i determine the maximum revenue obtainable by cutting *m* rods and selling the pieces.

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

Idea: apply the DP algorithm for rod-cutting on the longest rod and then use its solution (r,s) to compute optimal cutting and optimal revenue of each rod.

Rod Cutting having cutting-cost

You are given a rod of length n

A rod of length *i* inches will be sold for p_i dollars

Each cut costs you *c* dollars.

Problem: given a table of prices p_i determine the maximum revenue obtainable by cutting a rod of length n and selling the pieces.

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

Idea: Add cutting cost in the recurrence relation of rod-cutting:

$$r_n = max_{1 \le i \le n}(p_i + r_{n-i} - c) \text{ for } n>0;\ r_0 = 0$$

Chocolate Cutting/Breaking

You are given a Mimi chocolate having m*n blocks

In each step, we can break one piece of chocolate into two rectangular pieces horizontally/vertically (so no L shaped piece ever appears). A chocolate of *i**j* blocks can be sold for $p_{i,j}$ dollars. Given $p_{i,j}$ determine the maximum revenue $r_{m,n}$ obtainable by breaking a chocolate with m*n blocks and then selling the pieces.

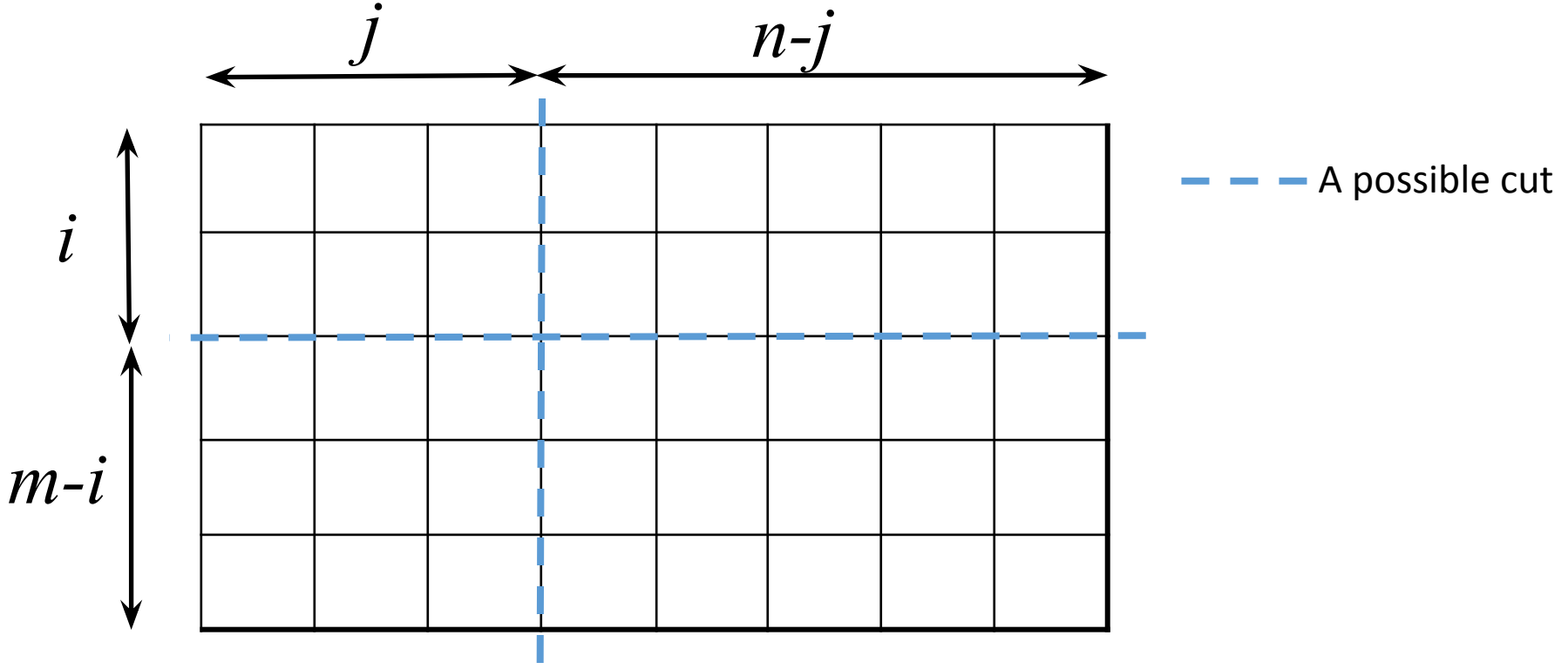
Price $p_{i,j}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	1	5	8	9	10	17	17	20	24	30
2	5	6	18	22	31	35	37	39	40	45
3	8	18	22	34	37	39	42	43	45	48
4	9	22	34	40	44	47	48	50	52	53

Chocolate Cutting

Recurrence relation:

$$r_{m,n} = max\left\{ \max_{1 \le i \le n}\{p_{i,n} + r_{m-i,n}\}, \max_{1 \le j \le n}\{p_{m,j} + r_{m,n-j}\} \right\}, \text{ when } n>0$$

What are the base cases?

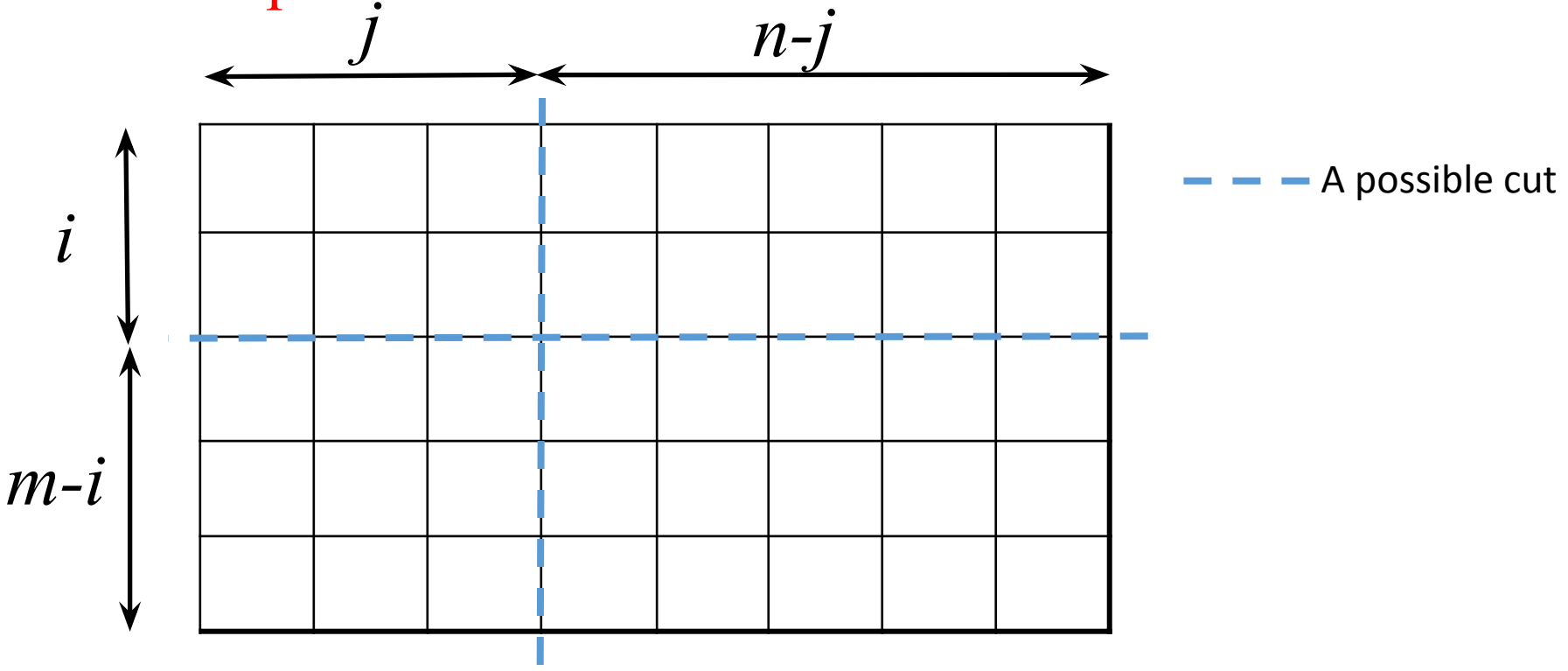


Chocolate Cutting

Recurrence relation:

$$r_{m,n} = max\left\{ \max_{1 \le i \le n}\{p_{i,n} + r_{m-i,n}\},\ \max_{1 \le j \le n}\{p_{m,j} + r_{m,n-j}\} \right\},\ when\ n>0$$

Can you write memoized top-down and bottom-up algorithms to solve this problem?



Properties of DP Problems

How can one tell if a DP algorithm will be able to solve an optimization problem i.e., whether a problem is a "DP problem"?

There are 2 properties exhibited by most DP Problems (problems solvable via a DP algorithm):

1. **Overlapping Subproblem Property**: To solve a large problem, the same subproblem is required to be solved again & again
   - It implies that a we can save time by solving each subproblem exactly once & saving that in a table, so that whenever that solution is required later, we can simply look up that table and reuse that value
2. **Optimal Sub-structure Property**: Optimal solution of a problem depends on optimal solution(s) of its subproblem(s)
   - It implies that the optimal solution(s) of subproblem(s) can be combined together to obtain the optimal solution of the problem itself