

Dynamic Programming

Divide & Conquer + Lookup Table

Overview

- The key idea of Divide & Conquer is to break a problem into smaller sub-problems and combine the result of those subproblems
- Some Problem can be divided into subproblems that is overlapping, i.e., same subproblem that happens more than once
 - If we use general D&C, each copies of the same subproblem will be solved repeatedly, wasting time
 - Dynamic Programming is a technique that use a look up table to store result of each sub-problem and immediately use it if any subproblem is required multiple times

Fibonacci Number

Fibonacci Number

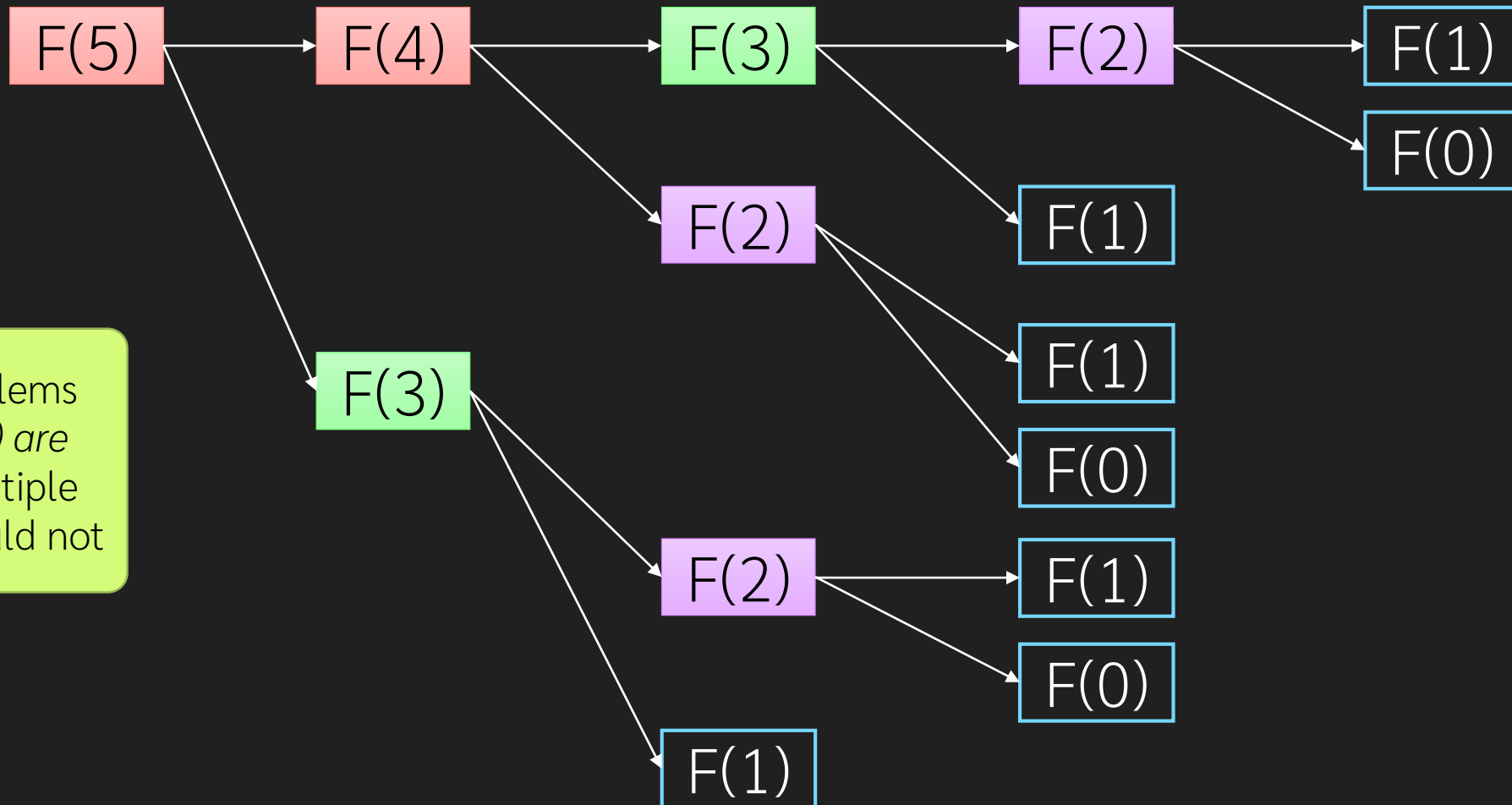
- **Problem:** compute $F(N)$, the Fibonacci function of N
- **Input:**
 - An integer $N \geq 0$
- **Output:**
 - $F(N)$, according to

$$F(N) = \begin{cases} F(N-1) + F(N-2) & ; n > 1 \\ 1 & ; n = 1 \\ 0 & ; n = 0 \end{cases}$$

Can be solve directly using Divide & Conquer

Recursion Tree

```
int fibo(int n) {  
    if (n == 1 || n == 0)  
        return n;  
    if (n >= 2)  
        return fibo(n-1) + fibo(n-2);  
}
```



Some subproblems
($F(3)$ and $F(2)$) are
computed multiple
times, they should not

Memoization: Simplest form of Dynamic Programming

- Top-Down approach
- Remember what have been done, if the subproblem is needed again, use the remembered result

```
ResultType DC(Problem p) {  
    if (p is trivial) {  
        solve p directly  
        return the result  
    } else {  
  
        divide p into  $p_1, p_2, \dots, p_n$   
        for (i = 1 to n)  
             $r_i = DC(p_i)$   
        combine  $r_1, r_2, \dots, r_n$  into r  
  
        return r  
    }  
}
```

```
ResultType DP(Problem p) {  
    if (p is trivial) {  
        solve p directly  
        return the result  
    } else {  
        if p is solved  
            return table.lookup(p); use  
        divide p into  $p_1, p_2, \dots, p_n$   
        for (i = 1 to n)  
             $r_i = DP(p_i)$   
        combine  $r_1, r_2, \dots, r_n$  into r  
        table.save(p, r); remember  
        return r  
    }  
}
```

Fibonacci: Top-Down DP

- `table` is an array[1..n] initialized by 0

```
int fibo_memo(int n) {  
    if (n == 1 || n == 0)  
        return n;  
  
    if (n >= 2) {  
        if (table[n] > 0) {  
            return table[n];  
        }  
        int value = fibo_memo(n-1) + fibo_memo(n-2);  
        table[n] = value;  
        return value;  
    }  
}
```

Exercise

- Draw recursion tree when we call fibo_memo(7)

```
//table is a global variable
int fibo_memo(int n) {
    if (n == 1 || n == 0)
        return n;

    if (n >= 2) {
        if (table[n] > 0) {
            return table[n];
        }
        int value = fibo_memo(n-1) + fibo_memo(n-2);
        table[n] = value;
        return value;
    }
}
```


Bottom-up dynamic programming

- Instead of relying on recursion to discover repetition of subproblems, we analyze the recursion directly and **build table constructively** from smaller subproblems
 - The **initial subproblems** are the ones from **trivial case** of Divide & Conquer recurrent relation
- **Benefit**: no-recursion, better runtime performance, (usually) easier to analyze
- **Drawback**: sometime, we build unnecessary sub-problem

Fibonacci: Bottom-Up DP

- From the definition of $F(N)$, we know that
 - $F(n)$ **needs** to know $F(n-1)$ and $F(n-2)$
 - In other words, if we know $F(n-1)$ and $F(n-2)$, then **we can construct $F(N)$**
- Initial Condition:
 - $F(0) = 0, F(1) = 1$
 - i.e., `table[0] = 0; table[1] = 1;`
- From the recurrent
 - `table[3] = table[2] + table[1]`
 - `table[4] = table[3] + table[2]`
 - ...

Fibonacci: Bottom Up

```
int fibo_bottom_up(int n) {  
    value[0] = 0;  
    value[1] = 1;  
    for (int i = 2; i <= n; ++i) {  
        value[i] = value[i-1] + value[i-2];  
    }  
    return value[n];  
}
```

Step 1

0	1				
---	---	--	--	--	--



Step 2

0	1	1							
---	---	---	--	--	--	--	--	--	--



Step 3

0	1	1	2						
---	---	---	---	--	--	--	--	--	--



Step 4

0	1	1	2	3					
---	---	---	---	---	--	--	--	--	--

Optimized version of Bottom-Up Fibo

- From bottom up approach, we know that we only need two prior Fibonacci numbers ($F(n-1)$ and $F(n-2)$) to compute the current Fibonacci number ($F(n)$)
 - There is no need to lookup for $F(n-3)$, $F(n-4)$, ... if we know $F(n-1)$, and $F(n-2)$
 - Hence, no need to use entire table
 - Just remember two previous Fibonacci number

```
def fibo(n)
    if (n == 0 || n == 1)
        return n
    f2 = 0
    f1 = 1
    for i from 2 to n
        #calculate current
        f = f2 + f1
        #prepare f1 and f2 for next round
        f2 = f1
        f1 = f
    end
    return f
end
```

Binomial Coefficient

choose r things from n things

Example 2: Binomial Coefficient

- $C_{n,r}$ = how to choose r things from n things
 - We have a closed form solution
 - $C_{n,r} = n! / (r!(n-r)!)$
 - We also have recurrence relation of $C_{n,r}$
 - $C_{n,r} = C_{n-1,r} + C_{n-1,r-1}$
 - $= 1$; $r = 0$
 - $= 1$; $r = n$
- What is the subproblem?
- Do we have overlapping subproblem?
- Input:
 - Two integer r and n ($0 \leq r \leq n$)
- Output:
 - $C_{n,r}$

Binomial Coefficient

- Each subproblem is represented by 2 numbers, r and n
 - Hence, the table should be 2D

```
int bino_naive(int n,int r) {  
    if (r == n) return 1;  
    if (r == 0) return 1;  
  
    int result = bino_naive(n-1,r) + bino_naive(n-1,r-1);  
    return result;  
}
```

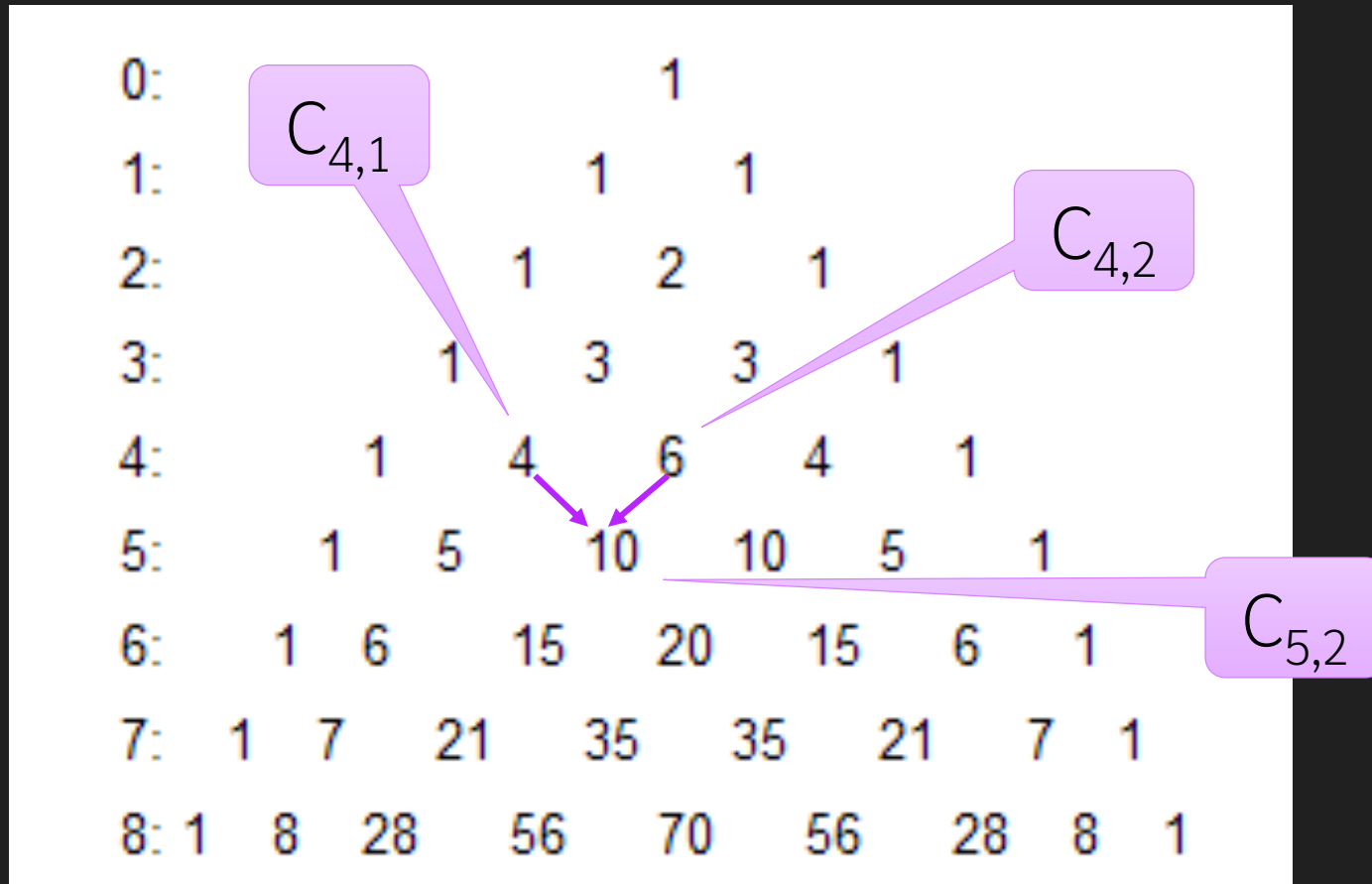
Binomial Coefficient: Top-Down (Memoization)

- `table[0..n][0..n]` is initialized by `-1`

```
int bino_memoize(int n,int r) {  
    if (r == n) return 1;  
    if (r == 0) return 1;  
  
    if (table[n][r] != -1)  
        return table[n][r];  
  
    int result = bino_memoize(n-1,r) + bino_memoize(n-1,r-1);  
    table[n][r] = result;  
  
    return result;  
}
```


Binomial Coefficient: Bottom Up

- Pascal triangle is a by-hand bottom-up DP of Binomial Coeff.



Binomial Coefficient: Bottom Up

	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1		1				
3	1			1			
4	1				1		
5	1					1	
6	1						1

```
int bino_DP(int n,int r) {  
    for (int i = 0;i <= n;i++) {  
        table[i][0] = 1;  
        table[i][i] = 1;  
    }  
    for (int i = 1;i <= n;i++) {  
        for (int j = 1;j < i;j++) {  
            table[i][j] = table[i-1][j] + table[i-1][j-1];  
        }  
    }  
    return table[n][r];  
}
```

Question

- Is it possible to fill the table in different order?
- Does previous code solve subproblem that we does not need?
 - If yes, how to avoid?

Maximum Subarray Sum

Revisiting

The problem

- Given array $A[1..n]$ of numbers, may contain negative number
 - Find a non-empty subarray $A[p..q]$ such that the summation of the values in the subarray is maximum
- Input:
 - $A[1..n]$
- Output:
 - p and q , where $1 \leq p \leq q \leq n$ and summation of $A[p..q]$ is maximum
- Example:
 - $A = [1, 4, 2, 3]$ output: 1 and 4
 - $A = [-2, -1, -3, -5]$ output: 2 and 2
 - $A = [2, 3, -6, 4, -2, 3, -5, -4, 3]$ output: 4 and 6

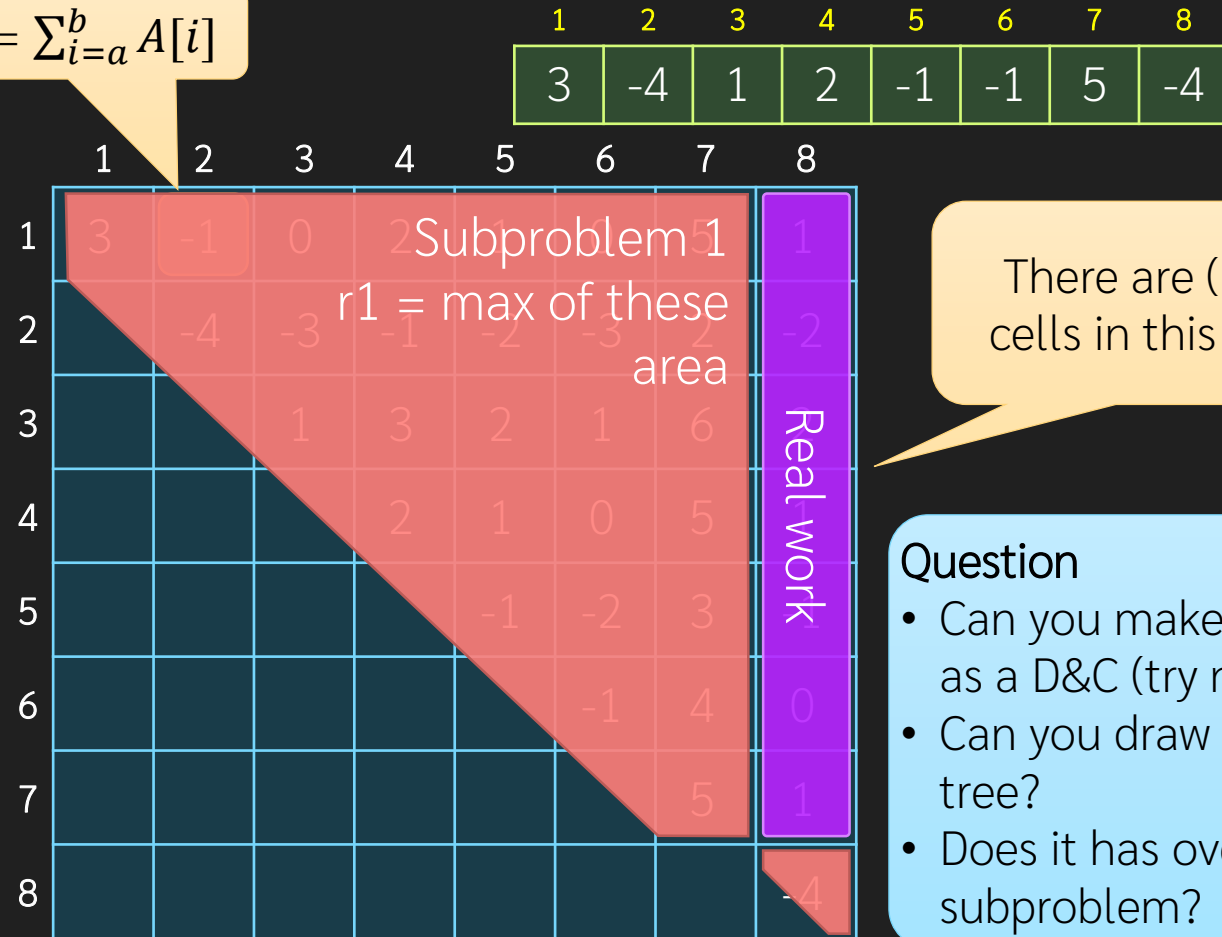
D&C by n-1

- Instead of dividing n into 2 of $n/2$ as previously done, we divide by $n-1$ and 1
 - The real work is solved by another D&C

```
def mss(A, stop)
  if (stop == 1)
    return A[1]
  r1 = mss(A, stop-1)
  r2 = A[stop]
  r3 = max_suffix(A, stop-1) + A[stop]
  return max(r1, r2, r3)
end
```

$$\text{max_suffix}(A, m) = \max_{1 \leq k \leq m} \sum_{i=k}^m A[i]$$

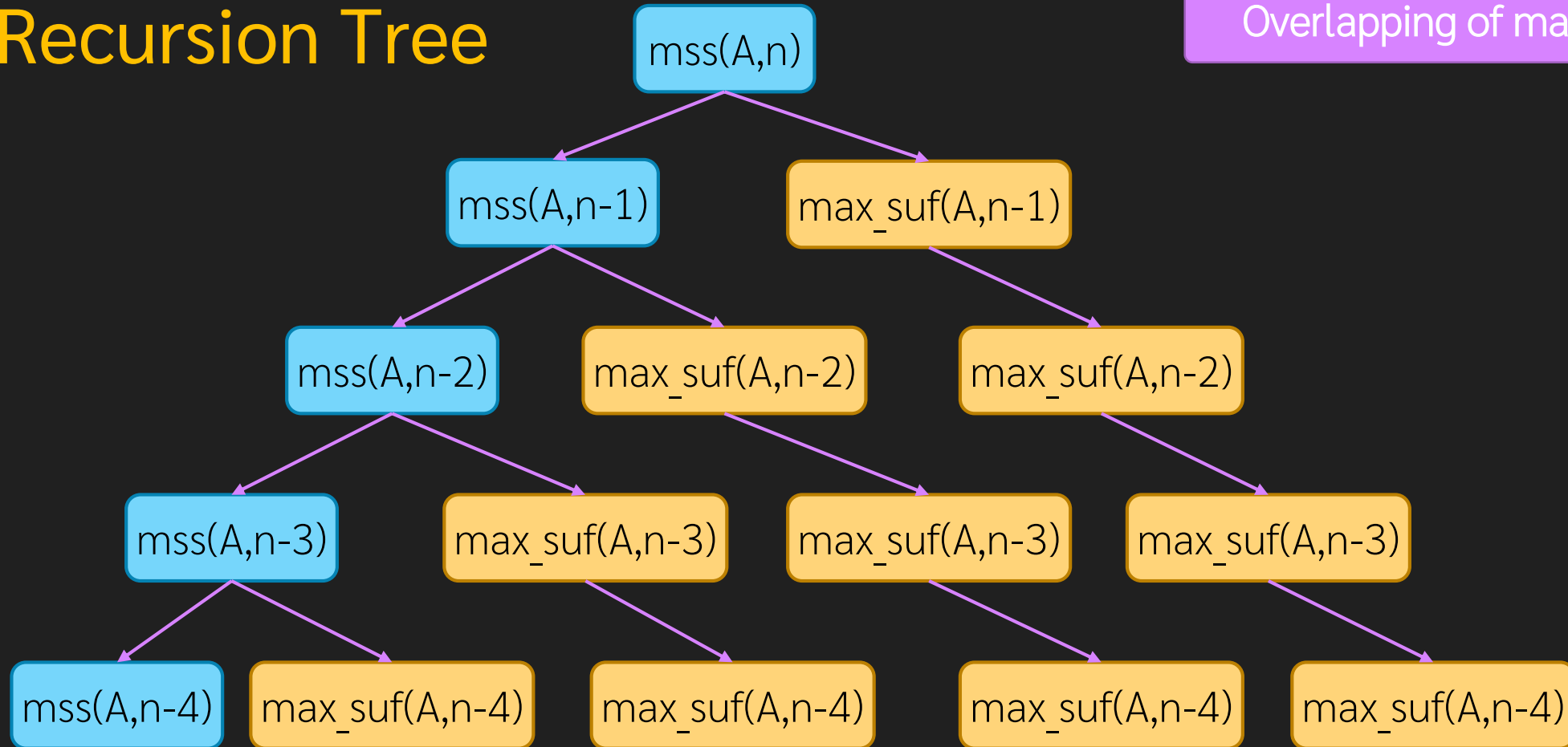
$$B[a][b] = \sum_{i=a}^b A[i]$$



```
def max_suffix(A, stop)
  if stop == 1
    return A[1]
  return max(A[stop],
             A[stop] + max_suffix(A, stop-1))
end
```

Recursion Tree

Overlapping of max_suf



MSS with Dynamic Programming

```
def mss(A, stop)
  if (stop == 1)
    return A[1]
  r1 = mss(A, stop-1)
  r2 = A[stop]
  r3 = max_suffix(A, stop-1) + A[stop]
  return max(r1, r2, r3)
end
```

```
def max_suffix(A[1..n], stop, table[1..n], done[1..n])
  if stop == 1
    return A[1]
  if (done[stop])
    return table[stop]

  table[stop] =
    max(A[stop],
        A[stop] + max_suffix(A, stop-1))
  done[stop] = true
  return table[stop]
end
```

- Memoization (top-down) approach
- Since the value of `max_suffix` can be negative, we need another table to determine whether this subproblem is already solved
 - `done[1..n]` is initialized as `false`

Bottom-Up approach

- Direct version
 - Build `max_suf` first
 - Calculate `mss` from 1 to `n`
- Optimized version (Kadane's Algorithm)
 - See that we need only one `max_suf`

```
def mss_bottom_up(A[1..n])
  max_suf is array [1..n]
  max_suf[1] = A[1]
  for i from 2 to n
    max_suf[i] = max(max_suf[i-1]+A[i],A[i])

  mss = A[1]
  for i from 2 to n
    mss = max(mss,
              max(A[i],
                  max_suf[i-1]))

  return mss
end
```


Kadane's Algorithm

```
def kadane(A[1..n])  
    suf = A[1]  
    mss = A[1]  
    for i from 2 to n  
        suf = max(A[i], suf+A[i])  
        mss = max(mss, suf)  
    return mss  
end
```

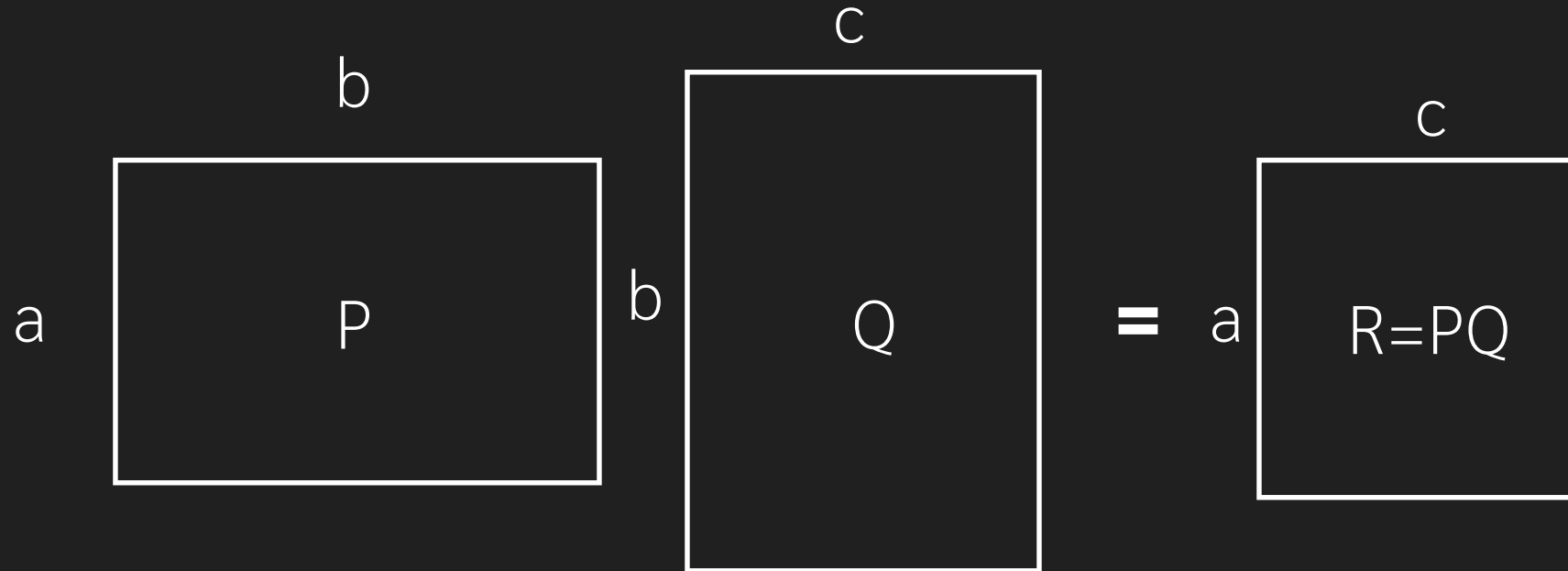


- Calculate both **mss** and **suf** on the fly
- Original problem was proposed by **Ulf Grenander** in 1977
 - Originally 2D problem, convert to 1D to gain insight
- **$O(n \log n)$** D&C proposed by **Michael Shamos**
- **Joseph Born Kadane** heard the problem in a seminar and propose **$O(n)$**

Matrix Chain Multiplication

Non-trivial bottom-up

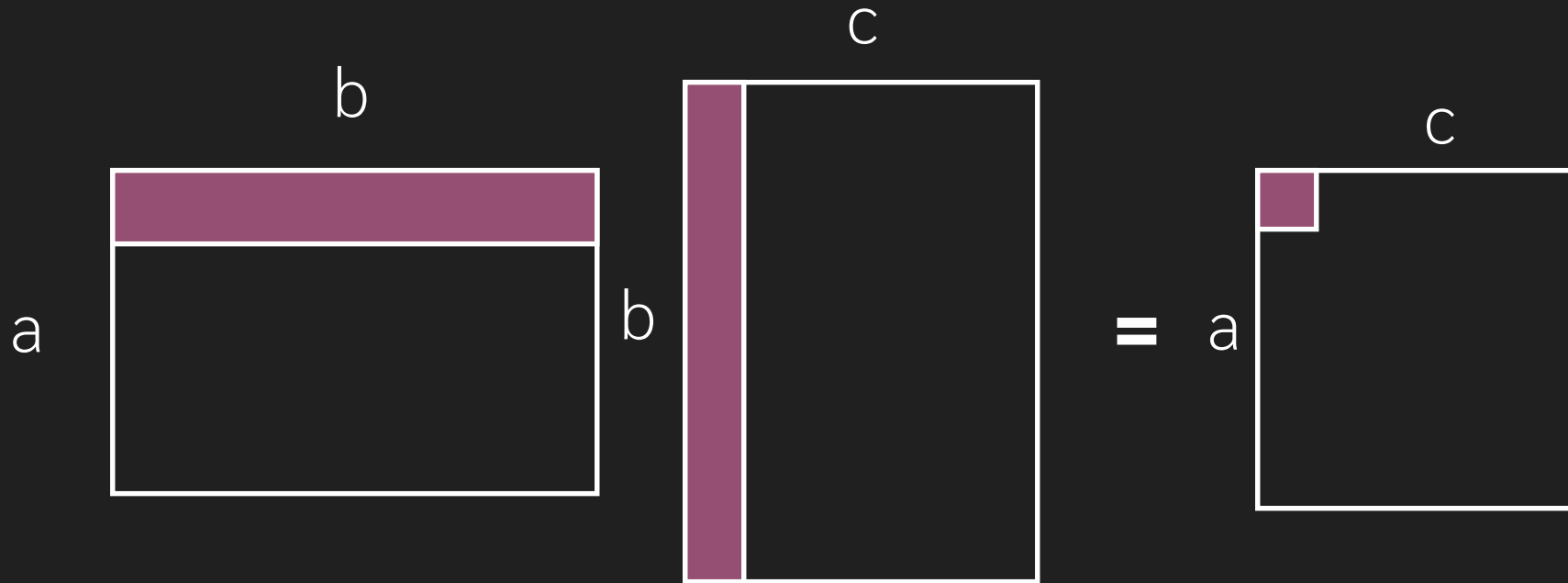
Matrix Multiplication



P = matrix with a rows and b columns

Q = matrix with b rows and c columns

Multiplying the Matrix



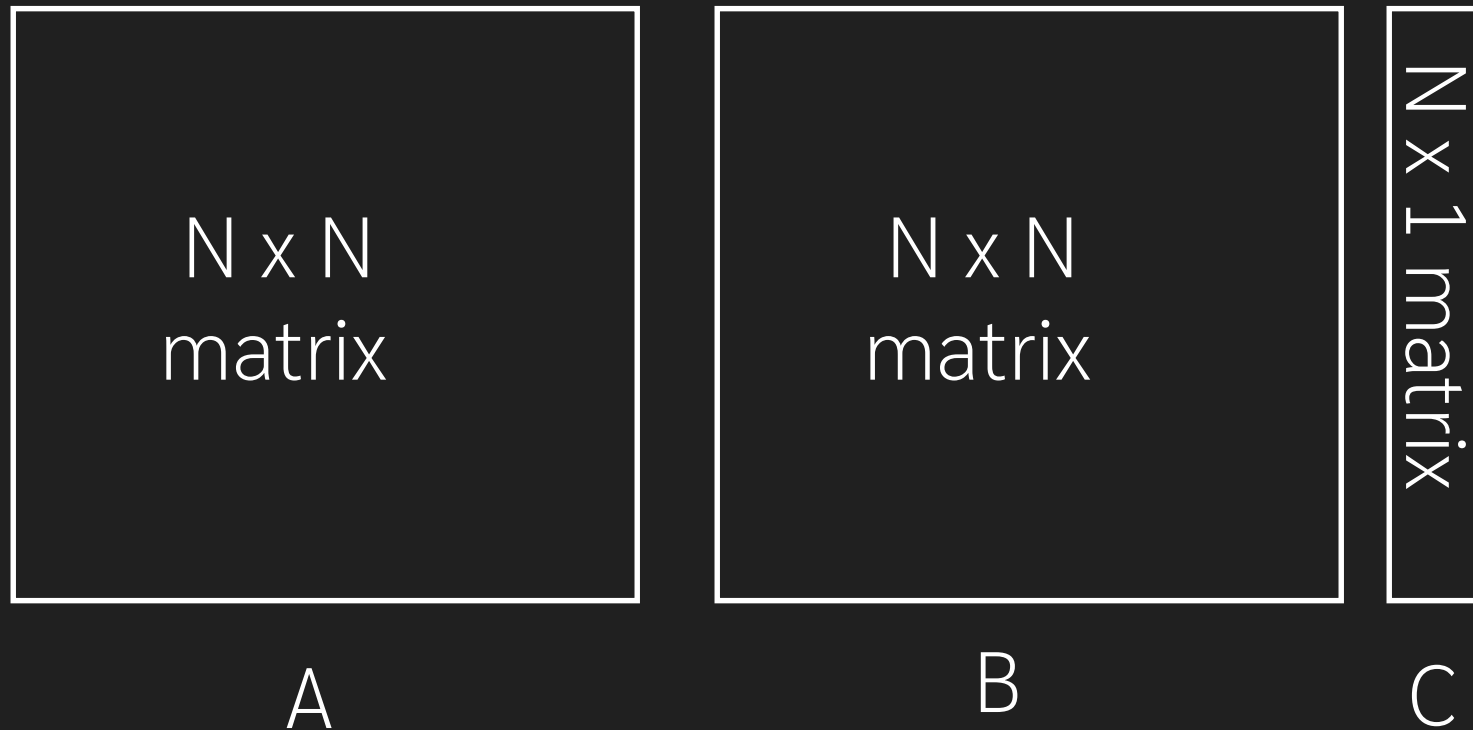
Time used = $\Theta(abc)$

Naïve Method

```
for (i = 1; i <= a; i++) {  
    for (j = 1; j <= c; j++) {  
        sum = 0;  
        for (k = 1; k <= b; k++) {  
            sum += P[i][k] * Q[k][j];  
        }  
        R[i][j] = sum;  
    }  
}
```

$O(abc)$

Matrix Chain Multiplication

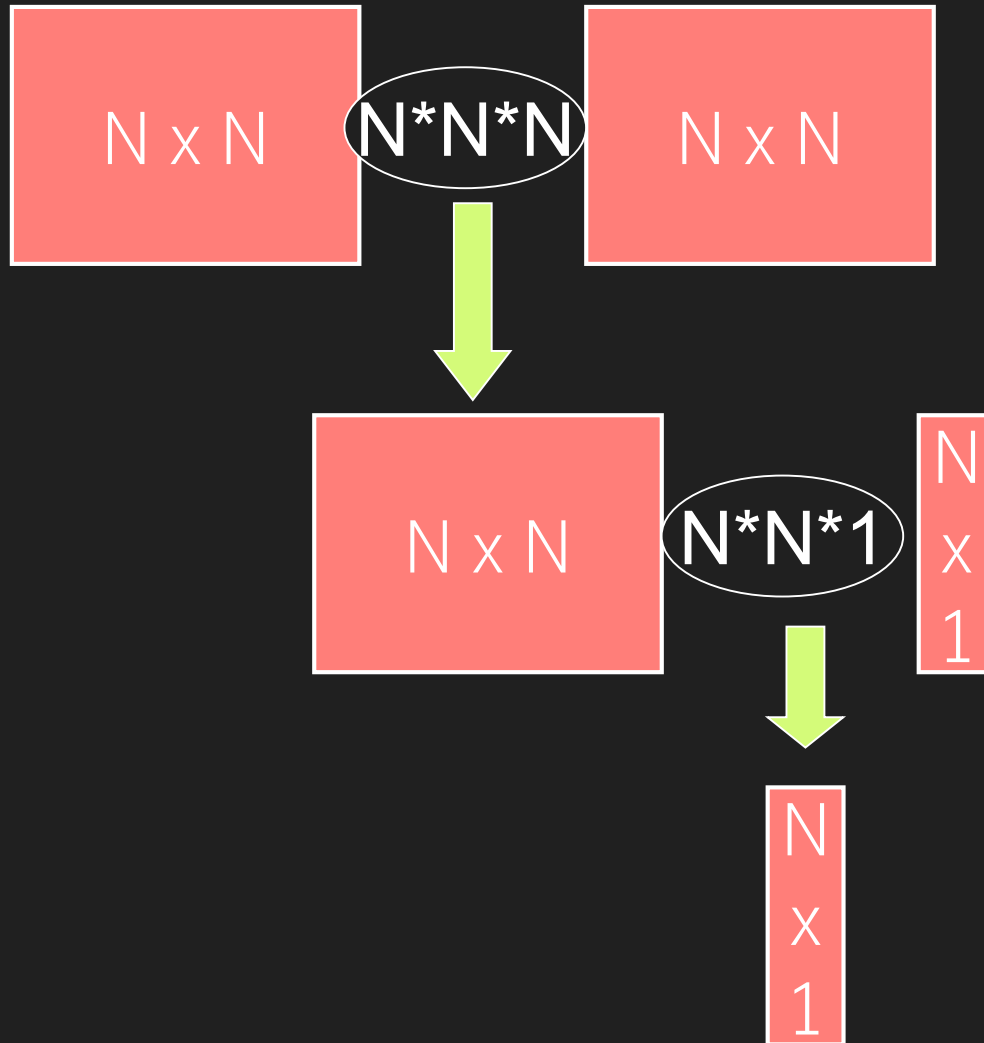


How to compute ABC ?

Matrix Multiplication

- $ABC = (AB)C = A(BC)$
- $(AB)C$ differs from $A(BC)$?
 - Same result, different efficiency
- What is the cost of $(AB)C$?
- What is the cost of $A(BC)$?

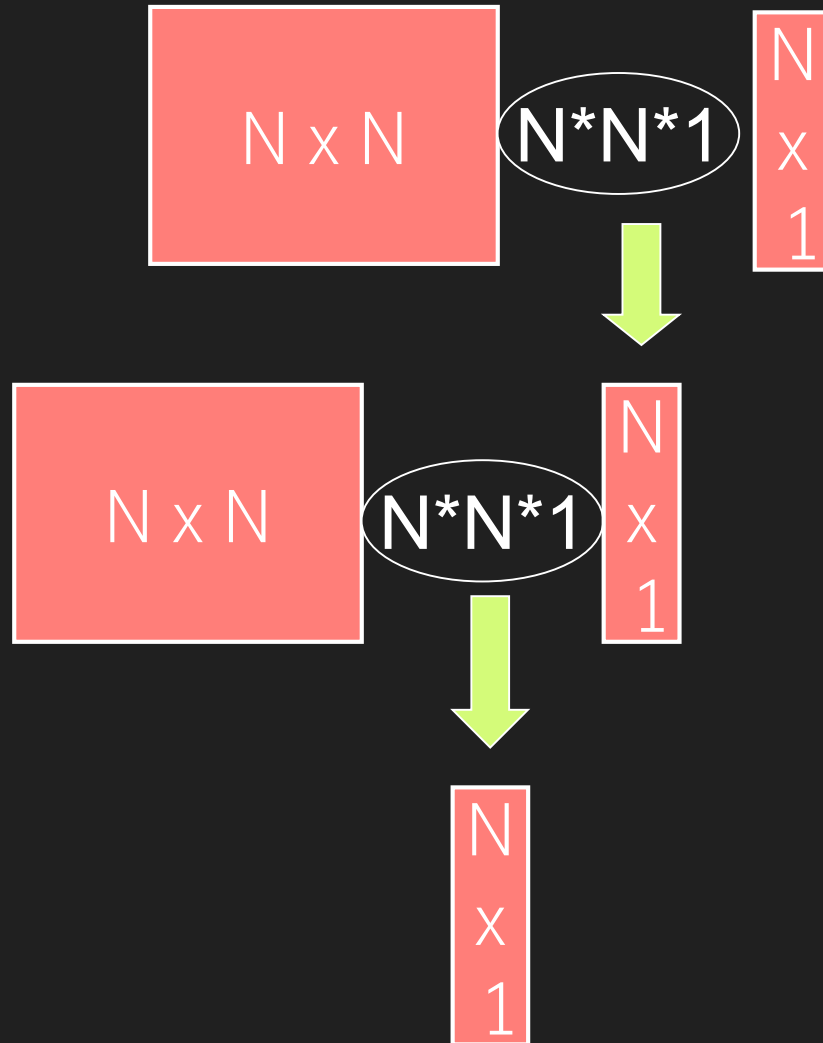
(AB)C



- Total = $N^3 + N^2$

A(BC)

- Total = $2N^2$



The Problem

- Input:

- $a_1, a_2, a_3, \dots, a_n$

- Output:

- The order of multiplication
 - How to parenthesize the chain
- How many multiplication is needed

- Example Instance:

- Input: 10 10 10 1

Output: $(B_1(B_2B_3))$ 200

These represents the size of the $n-1$ matrices $B_1 \dots B_{n-1}$

$a_1 \times a_2$	B_1
$a_2 \times a_3$	B_2
$a_3 \times a_4$	B_3
....	
$a_{n-1} \times a_n$	B_{n-1}

More Example

INPUT

- $a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6$
- $10 \times 5 \times 1 \times 5 \times 10 \times 2$
 $B_1 \quad B_2 \quad B_3 \quad B_4 \quad B_5$

Possible Output

$$((B_1 B_2)(B_3 B_4))B_5$$

$$(B_1 B_2)((B_3 B_4)B_5)$$

$$(B_1((B_2 B_3)B_4))B_5$$

And much more...

Consider the Output

What do

$$(B_1B_2)((B_3B_4)B_5)$$

$$(B_1B_2)(B_3(B_4B_5))$$

have in
common?

What do

$$((B_1B_2)(B_3B_4))B_5$$

$$(((B_1B_2)B_3)B_4))B_5$$

have in
common?

Solving $B_1 B_2 B_3 B_4 \dots B_{n-1}$

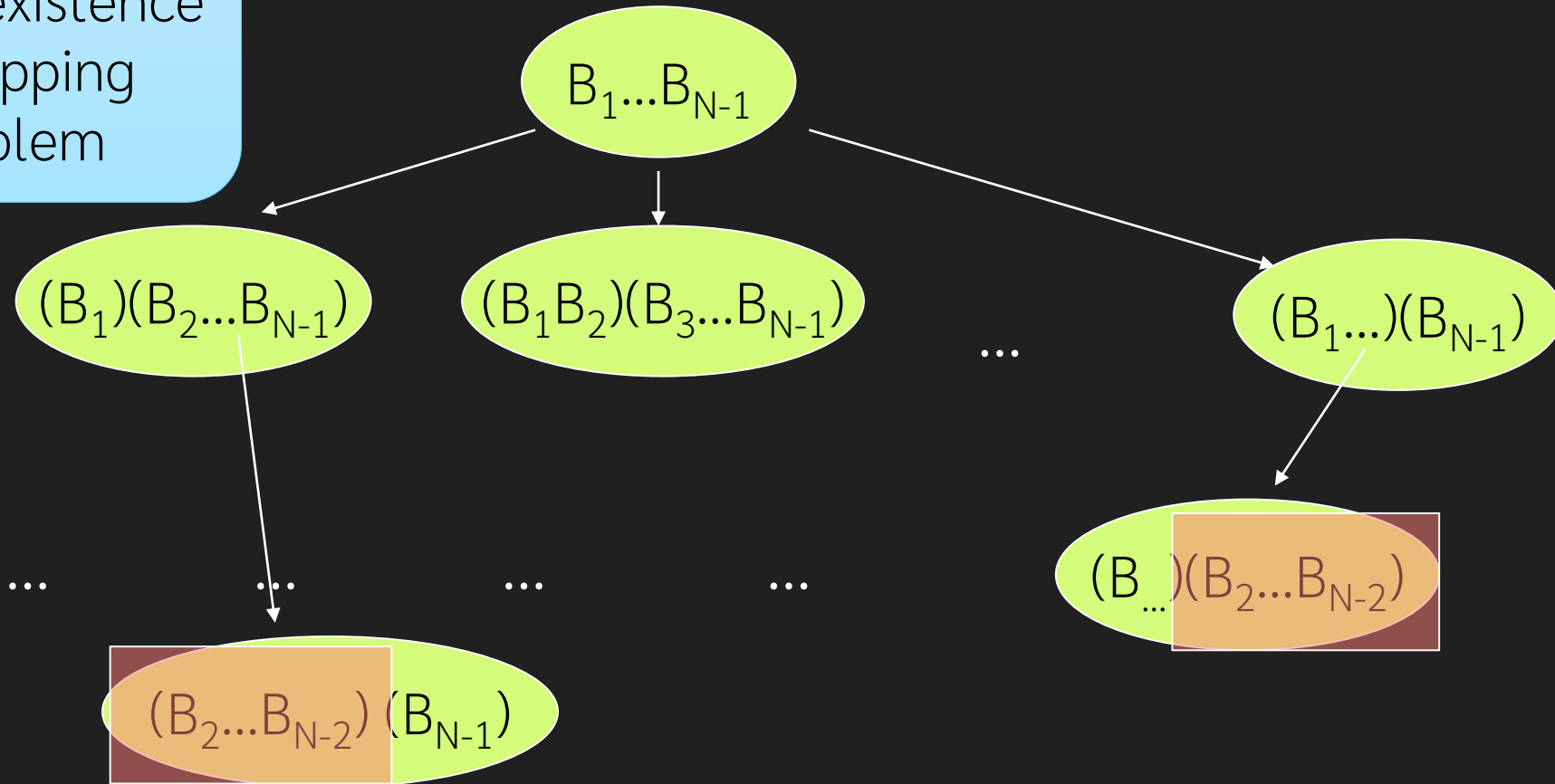
Min cost of

(1) B_1 $(B_2 B_3 \dots B_{n-1})$
 (2) $(B_1 B_2)$ $(B_3 B_4 \dots B_{n-1})$
 (3) $(B_1 B_2 B_3)$ $(B_4 B_5 \dots B_{n-1})$
 ...
 (n-2) $(B_1 B_2 B_3 B_4 \dots)$ B_{n-1}

- Each options ((1)..(n-2)) has 1 or 2 subproblems
- Sub problem is described by indices of left and right matrix
 - Needs 2 integers to describe a subproblem
- No overlapping subproblem (yet)

Overlapping Subproblem

Have to dig deeper
to identify existence
of overlapping
subproblem



Deriving the Recurrence Relation for D&C

- $mcm(l,r)$
 - The least cost to multiply $B_l \dots B_r$
- The solution is $mcm(1,n-1)$
- Initial Case, when $(r - l) \leq 1$ (one or two matrices)
 - $mcm(x,x) = 0$
 - $mcm(x,x+1) = a[x] * a[x+1] * a[x+2]$

The Recurrence Relation

- Recursion Case

$$\begin{array}{lcl}
 \text{mcm}(l,r) = \min \text{ of} & \begin{array}{c} \text{Subproblems} \\ \hline \text{min cost of } B_l \text{ (} B_{l+1} \text{ mcm}(l+1,r) B_r \text{)} \\ \text{min cost of } \text{mcm}(l, l+1) \text{ (} B_{l+2} \text{ mcm}(l+2,r) B_r \text{)} \\ \text{min cost of } \text{mcm}(l, l+2) \text{ (} B_{l+3} \text{ mcm}(l+3,r) B_r \text{)} \\ \dots \\ \text{min cost of } \text{mcm}(l, r-1) B_r \end{array} & \begin{array}{c} \text{Final multiplication} \\ \hline + a_l \cdot a_{l+1} \cdot a_{r+1} \\ + a_l \cdot a_{l+2} \cdot a_{r+1} \\ + a_l \cdot a_{l+3} \cdot a_{r+1} \\ \dots \\ + a_l \cdot a_r \cdot a_{r+1} \end{array}
 \end{array}$$

Divide & Conquer

```
int mcm(int l,int r) {  
    if (l < r) {  
        minCost = MAX_INT;  
        for (int i = l;i < r;i++) {  
            my_cost = mcm(l,i) + mcm(i+1,r) + (a[l] * a[i+1] * a[r+1]);  
            minCost = min(my_cost,minCost);  
        }  
        return minCost;  
    } else {  
        return 0;  
    }  
}
```

Using bottom-up DP

- Design the table
 - $M[i][j]$ = the best solution (min cost) for multiplying $B_i \dots B_j$
 - $M[i][j]$ stores $mcm(i,j)$
 - The solution is at $M[1][n-1]$
- Trivial Case
 - What is $M[x][x]$?
 - No multiplication, $M[x][x] = 0$
- Simple case
 - What is $M[x][x+1]$?
 - $B_x B_{x+1}$
 - Only one solution = $a_x * a_{x+1} * a_{x+2}$

What is $M[i,j]$?

- General case
 - What is $M[x][x+k]$?
 - $B_x B_{x+1} B_{x+2} \dots B_{x+k}$

min of

$$\left\{ \begin{array}{ll} M[x][x] + M[x+1][x+k] & + a_x \cdot a_{x+1} \cdot a_{x+k+1} \\ M[x][x+1] + M[x+2][x+k] & + a_x \cdot a_{x+2} \cdot a_{x+k+1} \\ M[x][x+2] + M[x+3][x+k] & + a_x \cdot a_{x+3} \cdot a_{x+k+1} \\ \dots & \\ M[x][x+k-1] + M[x+k][x+k] & + a_x \cdot a_{x+k} \cdot a_{x+k+1} \end{array} \right.$$

Filling the Table

The diagram shows a 6x6 grid representing a table. The first column and the first row are highlighted in a darker shade of purple. A green box labeled $M[1,1]$ has an arrow pointing to the cell at row 1, column 1. Another green box labeled $M[1,6]$ (our solution) has an arrow pointing to the cell at row 1, column 6. The grid is as follows:

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

Filling the Table




Arbitrary
case

	1	2	3	4	5	6
1	0					
2		0				
3			0			
4				0		
5					0	
6						0

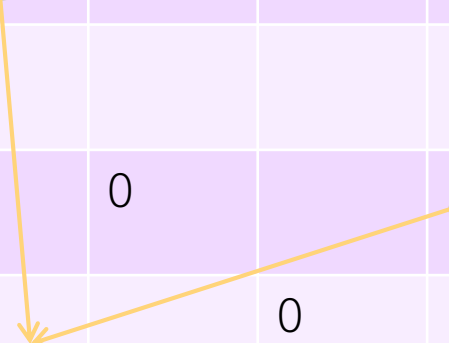
Plus $a_1 \cdot a_2 \cdot a_6$

Filling the Table

Arbitrary
case

	1	2	3	4	5	6
1	0					
2		0				
3			0			
4				0		
5					0	
6						0

Plus $a_1 \cdot a_3 \cdot a_6$



Filling the Table

Arbitrary
case

	1	2	3	4	5	6
1	0					
2		0				
3			0			
4				0		
5					0	
6						0

The diagram illustrates the process of filling a 6x6 table. A green box highlights the expression $\text{Plus } a_1 \cdot a_4 \cdot a_6$ at the intersection of row 4 and column 3. Two orange ovals are placed at the intersections of (row 1, column 3) and (row 4, column 5). An orange arrow points from the oval at (1, 3) to the green box. Another orange arrow points from the oval at (4, 5) to the green box. A red rectangle is located at the intersection of row 1 and column 5. The table contains zeros at (1,1), (2,2), (3,3), (4,4), (5,5), and (6,6). The header row and column are highlighted in light blue.

Filling the Table

Arbitrary
case

	1	2	3	4	5	6
1	0					
2		0				
3			0			
4				0		
5					0	
6						0

The diagram illustrates the process of filling a 6x6 table. A green box containing the text "Plus $a_1 \cdot a_5 \cdot a_6$ " is positioned over the cell at row 5, column 4. Two orange arrows originate from this box: one points to the cell at row 1, column 4, and the other points to the cell at row 5, column 5. Both of these target cells are highlighted with orange ovals. Additionally, a red rectangle is present in the cell at row 1, column 5.

Filling the Table

	1	2	3	4	5	6
1	0					
2		0				
3			0			
4				0		
5					0	
6						0

Filling the Table

	1	2	3	4	5	6
1	0					5
2		0			3	4
3			0	1	2	
4				0		
5					0	
6						0

The diagram illustrates the process of filling a 6x6 table. The main diagonal elements are all 0. The arrows show the sequence of calculations for the upper triangular part of the table:

- Arrow 1: From (3,3) to (4,4)
- Arrow 2: From (3,3) to (5,5)
- Arrow 3: From (2,2) to (5,5)
- Arrow 4: From (2,2) to (6,6)
- Arrow 5: From (1,1) to (6,6)

Example

- $a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6$

- $10 \times 5 \times 1 \times 5 \times 10 \times 2$

$B_1 \quad B_2 \quad B_3 \quad B_4 \quad B_5$



Example

a_1 a_2 a_3 a_4 a_5 a_6
10 x 5 x 1 x 5 x 10 x 2

	1	2	3	4	5
1	0				
2		0			
3			0		
4				0	
5					0

Example

a_1 a_2 a_3 a_4 a_5 a_6
 $10 \times 5 \times 1 \times 5 \times 10 \times 2$

	1	2	3	4	5
1	0	50			
2		0			
3			0		
4				0	
5					0

Example

a_1 a_2 a_3 a_4 a_5 a_6
10 x 5 x 1 x 5 x 10 x 2

	1	2	3	4	5
1	0	50			
2		0	25		
3			0		
4				0	
5					0

Example

a_1 a_2 a_3 a_4 a_5 a_6
10 x 5 x 1 x 5 x 10 x 2

	1	2	3	4	5
1	0	50			
2		0	25		
3			0	50	
4				0	
5					0

Example

a_1 a_2 a_3 a_4 a_5 a_6
10 x 5 x 1 x 5 x 10 x 2

	1	2	3	4	5
1	0	50			
2		0	25		
3			0	50	
4				0	100
5					0

Example

a_1 a_2 a_3 a_4 a_5 a_6
10 x 5 x 1 x 5 x 10 x 2

	1	2	3	4	5
1	0	50			
2		0	25		
3			0	50	
4				0	100
5					0

Example

$$\text{Option 1} = 0 + 25 + 10 \times 5 \times 5 = 275$$

$$\text{Option 2} = 50 + 0 + 10 \times 1 \times 5 = 100$$

$$a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6$$
$$10 \times 5 \times 1 \times 5 \times 10 \times 2$$

	1	2	3	4	5
1	0	50			
2		0	25		
3			0	50	
4				0	100
5					0

Example

a_1 a_2 a_3 a_4 a_5 a_6
10 x 5 x 1 x 5 x 10 x 2

	1	2	3	4	5
1	0	50	100 (2)		
2		0	25		
3			0	50	
4				0	100
5					0

(2) means that
the minimal
solution is by
dividing at B_2

Example

$$\text{Option 1} = 0 + 50 + 5 \times 1 \times 10 = 100$$

$$\text{Option 2} = 25 + 0 + 5 \times 5 \times 10 = 275$$

$$a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6$$
$$10 \times 5 \times 1 \times 5 \times 10 \times 2$$

	1	2	3	4	5
1	0	50	100 (2)		
2		0	25		
3			0	50	
4				0	100
5					0

Example

a_1 a_2 a_3 a_4 a_5 a_6
10 x 5 x 1 x 5 x 10 x 2

	1	2	3	4	5
1	0	50	100 (2)		
2		0	25	100 (2)	
3			0	50	70 (4)
4				0	100
5					0

Example

$$\text{Option 1} = 0 + 100 + 10 \times 5 \times 10 = 600$$

$$\text{Option 2} = 50 + 50 + 10 \times 1 \times 10 = 200$$

$$\text{Option 2} = 100 + 0 + 10 \times 5 \times 10 = 600$$

$$a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \quad a_6$$
$$10 \times 5 \times 1 \times 5 \times 10 \times 2$$

	1	2	3	4	5
1	0	50	100 (2)		
2		0	25	100 (2)	
3			0	50	70 (4)
4				0	100
5					0

Example

a_1 a_2 a_3 a_4 a_5 a_6
10 x 5 x 1 x 5 x 10 x 2

	1	2	3	4	5
1	0	50	100 (2)	200 (2)	
2		0	25	100 (2)	
3			0	50	70 (4)
4				0	100
5					0

Example

a_1 a_2 a_3 a_4 a_5 a_6
10 x 5 x 1 x 5 x 10 x 2

	1	2	3	4	5
1	0	50	100 (2)	200 (2)	
2		0	25	100 (2)	80 (2)
3			0	50	70 (4)
4				0	100
5					0

Example

a_1 a_2 a_3 a_4 a_5 a_6
10 x 5 x 1 x 5 x 10 x 2

	1	2	3	4	5
1	0	50	100 (2)	200 (2)	140 (2)
2		0	25	100 (2)	80 (2)
3			0	50	70 (4)
4				0	100
5					0

Analysis

- There is $O(n^2)$ cell to be filled
 - Each cell has $O(n)$ options
- This totals to $O(n^3)$

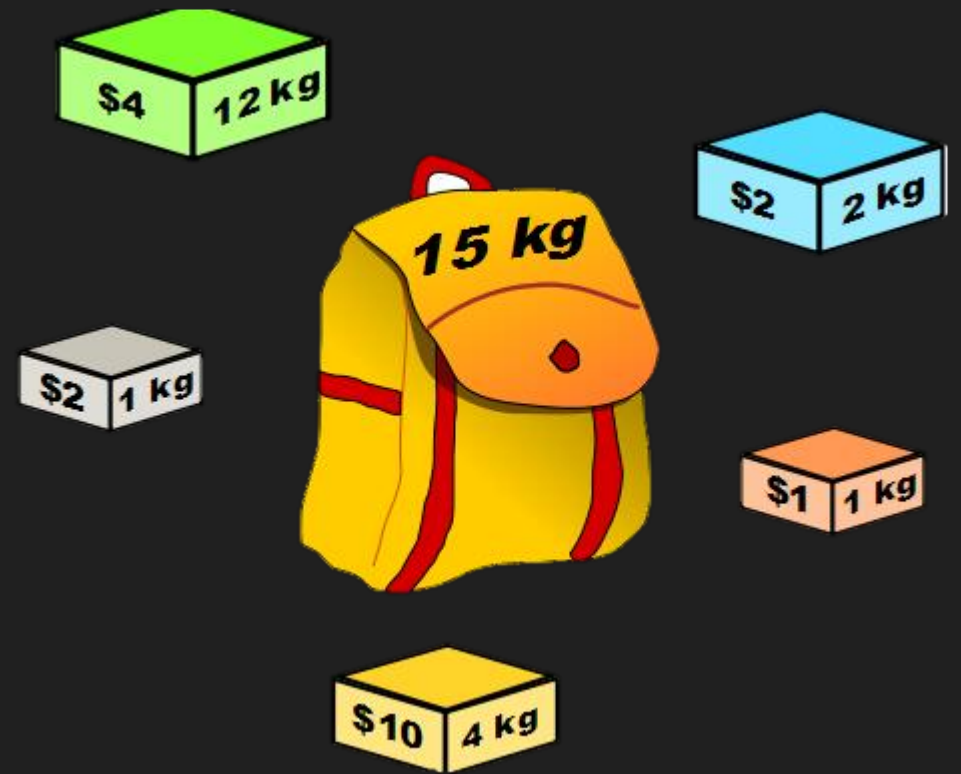
Can you write a code for
Bottom-up DP of Matrix
Chain Multiplication
Problem?

Also can your code build
the actual solution (the
parenthesis of B_i , not
just the minimum cost)

0-1 Knapsack Problem

Knapsack Problem

- Given a sack, able to hold W kg
- Given a list of objects
 - Each has a **value** and a **weight**
- Try to pack the object in the sack so that the total value is maximized



Variation

- Rational Knapsack

- Object is like a gold bar, we can cut it into pieces, each has the same value/weight ratio

- 0-1 Knapsack

- Object cannot be broken, we have to choose to take (1) or leave (0) the object
 - $W = 50$
 - Objects = (60, 10) (100, 20) (120, 30)
 - Best solution = second and third

The Problem

- Input:

- A number W , the capacity of the sack
- n values of weight and price
 - w_i = weight of the i^{th} items
 - p_i = price of the i^{th} item

- Output:

- A subset S of $\{1,2,3,\dots,n\}$ such that
 - $\sum_{i \in S} p_i$ is maximum
 - $\sum_{i \in S} w_i \leq W$

- Example Instance

- $W = 50$
- $P_i = 60, 100, 120$
- $w_i = 10, 20, 30$
- Best solution = second and third

Naïve approach

```
def knapsack(W,w[1..n],p[1..n],idx,pick[1..n])
  if (idx == 0)
    sum_price = 0
    sum_weight = 0
    for i from 1 to n
      if pick[i]
        sum_price += p[i]
        sum_weight += w[i]
      if (sum_weight <= W && sum_price > max)
        max = sum_price

    pick[idx] = false
    knapsack(W,w,p,idx-1,pick)
    pick[idx] = true
    knapsack(W,w,p,idx-1,pick)
end
```

- Try every possible combination of {1,2,3,...n}
- Test whether a combination satisfies the weight constraint
 - If so, remember the best one
 - Start with knapsack(W,w,p,n,[1..n])
 - `max` is global var
 - $\theta(2^n * n)$

Another Naïve approach

- Keep track of remaining weight, sum the total price along the way
- What is the benefit of this approach?

```
def knapsack(W,w[1..n],p[1..n],idx,remain)
  if (idx == 0)
    return 0
  if (remain >= w[idx])
    #r1 is that we don't pick item #idx
    r1 = knapsack(W,w,p,idx-1,remain)
    #r2 is that we pick item #idx
    r2 = knapsack(W,w,p,idx-1,remain - w[idx]) + p[idx]
    return max(r1,r2)
  else
    return knapsack(W,w,p,idx-1,remain)
end
```


The Recurrence Relation

- $K(a,b)$ = the best total price when and only item number 1 to number a is considered and the knapsack is of size b
- $K(a,b) = 0$ when $a = 0$ or $b = 0$
- $K(a,b) = K(a-1,b)$ when $w_a > b$
- $K(a,b) = \max(K(a-1, b - w_a) + p_a, K(a-1,b))$
- The solution is at $K(n,W)$

The Failed Attempt #1

- Let $K(a)$ be the best total value when we consider only item number 1 to number a and the weight limit is W
 - The answer is at $K(n)$
 - By definition, $K(n)$ and $K(n-1)$ and $K(n-2)$... all consider the same weight limit
- Let's say that the answer contains item number n
 - Also by definition, it means that $K(n) = K(n-1) + p_n$
 - However, $K(n-1)$ will consider the problem thinking that the weight limit is the same (not reduced by weight of item number n)
 - It is wrong to say that $K(a) = \max(K(a-1) + p_a, K(a-1))$
 - It is not possible to have a recurrence relation that does not consider W

The Failed Attempt #2

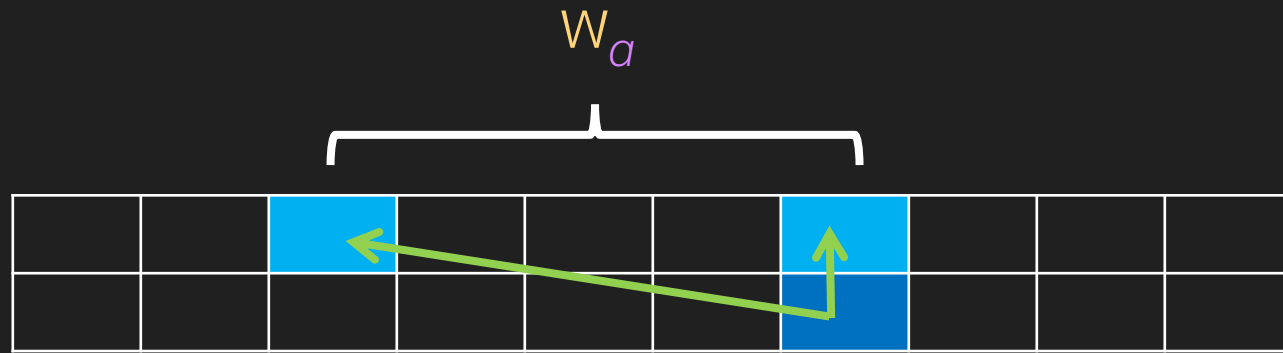
- Let $K(b)$ be the best total value when the weight limit of the sack is b
 - The answer is at $K(W)$
- If the i^{th} item is in the best solution
 - $K(W) = K(W - w_i) + p_i$
- But, we don't really know that the i^{th} item is in the optimal solution
 - So, we try everything
 - $K(W) = \max_{1 \leq i \leq n} (K(W - w_i) + p_i)$
- Is this our algorithm?
 - Yes, if and only if we allow each item to be selected multiple times (that is not true for this problem)

Exercise: Top-Down approach

- Write a top down dynamic programming approach using this recurrence relation
 - $K(a,b) = 0$ when $a = 0$ or $b = 0$
 - $K(a,b) = K(a-1,b)$ when $w_a > b$
 - $K(a,b) = \max(K(a-1,b - w_a) + p_a, K(a-1,b))$
- Which data structure should we use to store result?
 - Should we use 2D array?
 - Should we use associative data structure such as `std::map` or `std::unordered_map`?

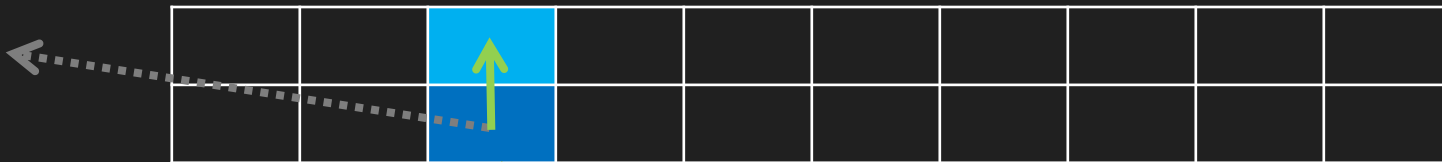
The Table for Bottom-Up

- Row = item id (a)
- Col = weight (b)



Normal case ($w_a \leq b$)

$K(a, b)$



$K(a, b)$

Too much weight ($w_a > b$)

Example

$p = \{4, 2, 2, 1, 10\}$

$w = \{12, 2, 1, 1, 4\}$ $W = 15$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0															
2	0															
3	0															
4	0															
5	0															

$K(a,b) = 0$

when $a = 0$ or $b = 0$

Example

$$p = \{4, 2, 2, 1, 10\}$$

$w = \{12, 2, 1, 1, 4\}$ $W = 15$

Fill row 1 ($p_1=4$ $w_1=12$)

[illegible]

Example

$p = \{4, 2, 2, 1, 10\}$

$w = \{12, 2, 1, 1, 4\}$ $W = 15$

Fill row 1 ($p_1=4$ $w_1=12$)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0				
2	0															
3	0															
4	0															
5	0															

$$K(a,b) = K(a-1,b)$$

when $w_a > b$

Example

$p = \{4, 2, 2, 1, 10\}$

$w = \{12, 2, 1, 1, 4\}$ $W = 15$

Fill row 1 ($p_1=4$ $w_1=12$)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0															
3	0															
4	0															
5	0															

$$K(a,b) = \max(K(a-1, b - w_a) + p_a, K(a-1, b))$$

Example

$p = \{4, 2, 2, 1, 10\}$

$w = \{12, 2, 1, 1, 4\}$ $W = 15$

Fill row 2 ($p_2=2$ $w_2=2$)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	0														
3	0															
4	0															
5	0															

$$K(a,b) = K(a,b-1) \quad \text{when } w_b > a$$

Example

$p = \{4, 2, 2, 1, 10\}$

$w = \{12, 2, 1, 1, 4\}$ $W = 15$

Fill row 2 ($p_2=2$ $w_2=2$)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	0														
3	0															
4	0															
5	0															

$$K(a,b) = K(a-1,b)$$

when $w_a > b$

Example

$p = \{4, 2, 2, 1, 10\}$

$w = \{12, 2, 1, 1, 4\}$ $W = 15$

Fill row 2 ($p_2=2$ $w_2=2$)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	0	2	2	2	2	2	2	2	2	2	2	4	4	6	6
3	0															
4	0															
5	0															

$$K(a,b) = \max(K(a-1, b - w_a) + p_a, K(a-1, b))$$

Example

$p = \{4, 2, 2, 1, 10\}$

$w = \{12, 2, 1, 1, 4\}$ $W = 15$

Fill row 2 ($p_2=2$ $w_2=2$)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	0	2	2	2	2	2	2	2	2	2	2	4	4	6	6
3	0															
4	0															
5	0															

$$K(a,b) = \max(K(a-1, b - w_a) + p_a, K(a-1, b))$$

Example

$p = \{4, 2, 2, 1, 10\}$

$w = \{12, 2, 1, 1, 4\}$ $W = 15$

Fill row 3 ($p_3=2$ $w_3=1$)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	0	2	2	2	2	2	2	2	2	2	2	4	4	6	6
3	0	2	2	4	4	4	4	4	4	4	4	4	4	6	6	8
4	0															
5	0															

$$K(a,b) = \max(K(a-1,b - w_a) + p_a, \quad K(a-1,b))$$

Example

$$p = \{4, 2, 2, 1, 10\}$$
$$w = \{12, 2, 1, 1, 4\} \quad W = 15$$

Fill row 3 ($p_3=2$ $w_3=1$)

[illegible]

Example

$$p = \{4, 2, 2, 1, 10\}$$

$w = \{12, 2, 1, 1, 4\}$ $W = 15$

Fill row 4 ($p_4=1$ $w_4=1$)

[illegible]

Example

$$p = \{4, 2, 2, 1, 10\}$$
$$w = \{12, 2, 1, 1, 4\} \quad W = 15$$

Fill row 4 ($p_4=1$ $w_4=1$)

[illegible]

Example

$p = \{4, 2, 2, 1, 10\}$

$w = \{12, 2, 1, 1, 4\}$ $W = 15$

Fill row 5 ($p_5=10$ $w_5=4$)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	0	2	2	2	2	2	2	2	2	2	2	4	4	6	6
3	0	2	2	4	4	4	4	4	4	4	4	4	4	6	6	8
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

Example

$p = \{4, 2, 2, 1, 10\}$

$w = \{12, 2, 1, 1, 4\}$ $W = 15$

Fill row 5 ($p_5=10$ $w_5=4$)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	0	2	2	2	2	2	2	2	2	2	2	4	4	6	6
3	0	2	2	4	4	4	4	4	4	4	4	4	4	6	6	8
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

Example

$$p = \{4, 2, 2, 1, 10\}$$

$$w = \{12, 2, 1, 1, 4\} \quad W = 15$$

Trace the solution backward to get the actual item number
We have item number 5,4,3,2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
2	0	0	2	2	2	2	2	2	2	2	2	2	4	4	6	6
3	0	2	2	4	4	4	4	4	4	4	4	4	4	6	6	8
4	0	2	3	4	5	5	5	5	5	5	5	5	5	6	7	8
5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

Bottom-Up Code

```
set all K[0][*] = 0 and all K[*][0] = 0
for a = 1 to n
  for b = 1 to W
    if (w[a] > b)
      K[a][b] = K[a - 1][b];
    else
      K[a][b] = max( K[a - 1][b - w[a]] + p[a] ,
                    K[a - 1][b] )
return K[n][W];
```

Can you write a code that generate the list of actual item that we take?

- Does this code generate too much subproblem?
- Does it generates one that we does not need?
- Is it better to use Top-Down approach?
 - Can you show some instance that Top-Down is better than Bottom-up (this code)

Analysis

- From Bottom-Up, it is clear that this is $O(Wn)$
- Original generate-all-solution method is $O(2^n)$
- Which one is better
 - In what case that $O(Wn)$ Dynamic Programming will benefit greatly (because there are several overlapping subproblems)