Algorithms – I (CS29003/203)

Autumn 2022, IIT Kharagpur

Dynamic Programing

Dynamic Programing

"*Life can only be understood going backwards, but it must be lived going forwards.*"
- S. Kierkegaard, Danish Philosopher.

The first line of the famous book by Dimitri P Bertsekas.

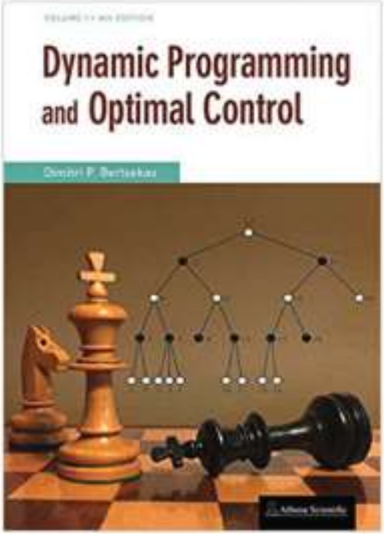


Image taken from: *amazon.com*

Dynamic Programing

- Dynamic Programing (DP), like Divide-and-conquer, solves problems by combining solutions to subproblems
- It was invented as a general method for optimizing multistage decision processes in the 1950s by prominent US Mathematician Richard Bellman

Richard Bellman

- Why is it named **programing**?
  - 'Programing' here refers to a tabular method for solving an optimization problem as opposed to writing computer code
  - Much like linear programing or quadratic programing

Dynamic Programing

- It has similarity with a divide and conquer approach. However, unlike divide and conquer, DP applies when the subproblems overlap – i.e., when **sub**problems share **subsub**problems
- Divide and conquer may work more than necessary, repeatedly solving the same subsubproblems
- A DP algorithm solves each subsubproblem only once and caches the answer to be reused later, thereby avoiding recomputation
- We shall start by revisiting the beautiful Fibonacci number and sequence
- The Fibonacci numbers are the elements of the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34$$

Fibonacci Number Recursive Algorithm

- The Fibonacci numbers are the elements of the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34$$

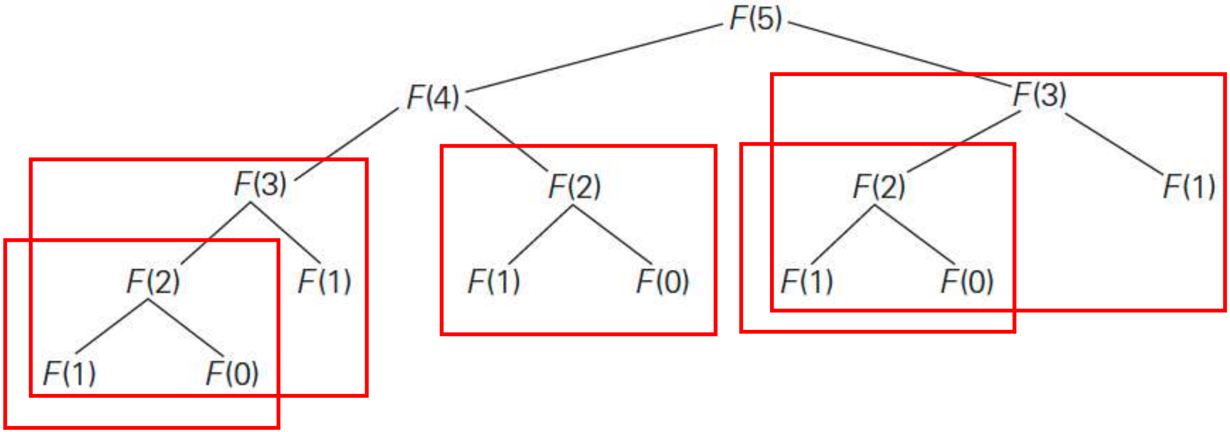
- This can be defined by the following simple recurrence

$$F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)$$

- A recursive algorithm to compute the Fibonacci number can be

```
//Input (n): A nonnegative integer
//Output: The nth Fibonacci number
Fib(n)
   if ((n==0) or (n==1))
       return n
   return Fib(n-1) + Fib(n-2)
```

Fibonacci Number Recursive Algorithm



- Runtime complexity?
- Its exponential
- Intuitive explanation: Every node needs a sum and number of nodes is roughly 2^n

Fibonacci Number – Efficient Computation

- It does a lot of duplicate computations
- How can we avoid that?
- We compute each subproblem only once and reuse them
- Lets take a bottom-up approach rather than a top-down approach
- That is – instead of starting from top of the tree why don’t we start small (the base cases, n=0 or 1) and use the solutions of the smaller problems to get the solution to the bigger problems

```
Fib(n)
    Create an array A of size n+1 // 0-based index
   A[0] = 0
   A[1] = 1
   for i=2 to n:
       A[i] = A[i-1] + A[i-2]
   return A[n]
```

Fibonacci Number – Efficient Computation

- What makes it efficient is we are solving the smaller problems first and storing them in a table (array)
- Whenever we need the solution of a smaller subproblem, instead of recomputing, we are looking up the solution in the table and reusing it
- Runtime?
- $\Theta(n)$

```
Fib(n)
    Create an array A of size n+1 // 0-based index
    A[0] = 0
    A[1] = 1
    for i=2 to n:
        A[i] = A[i-1] + A[i-2]
    return A[n]
```

Unlimited Number Knapsack

- Overall weight limit: 8 lb, we can take an unlimited number of each item
- Item a: 5 lb, $150
- Item b: 4 lb, $100
- Item c: 2 lb, $10

- Expensive first: Item a + Item c, value: $160
- Lightest first: Item c, 4 times, value: $40
- Optimal: Item b, 2 times, value: $200

- Greedy strategy does not provide the optimal solution
- A naive solution? Try all possibilities!

Source: UC Riverside, CS141 course, Fall 2021

A Naïve Algorithm

- Item a: 5 lb, $150
- Item b: 4 lb, $100
- Item c: 2 lb, $10

suitcase(8)

- suitcase(3)+150 //a fits
  - suitcase(1)+10 //Only c fits
    - 0 //Nothing fits
  - max(0,0+10)=10
- max(0,10+150)=160

```
suitcase(leftWeight)
  curBest = 0
  foreach item of (weight, val)
    if (leftWeight >= weight)
      curBest = max(curBest, suitcase(leftWeight-weight)+val)
  return curBest
```

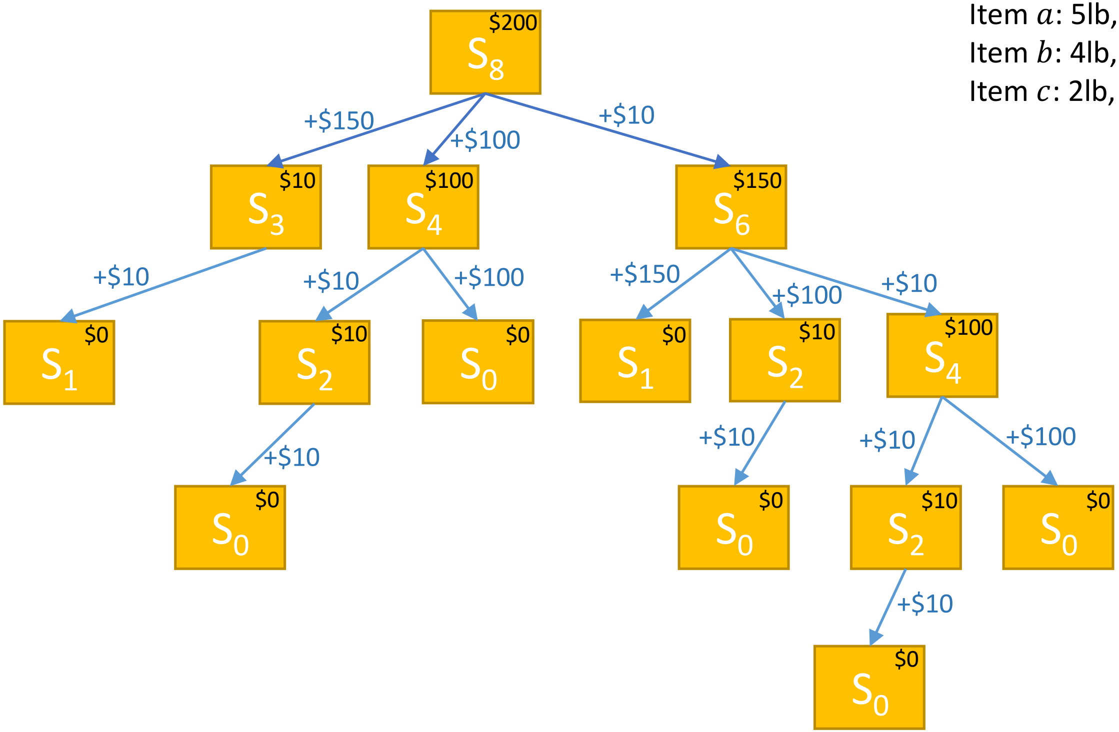
Recursive call

```
// Calling the function
answer = suitcase(8)
```

Source: UC Riverside, CS141 course, Fall 2021

Execution Recurrence Tree

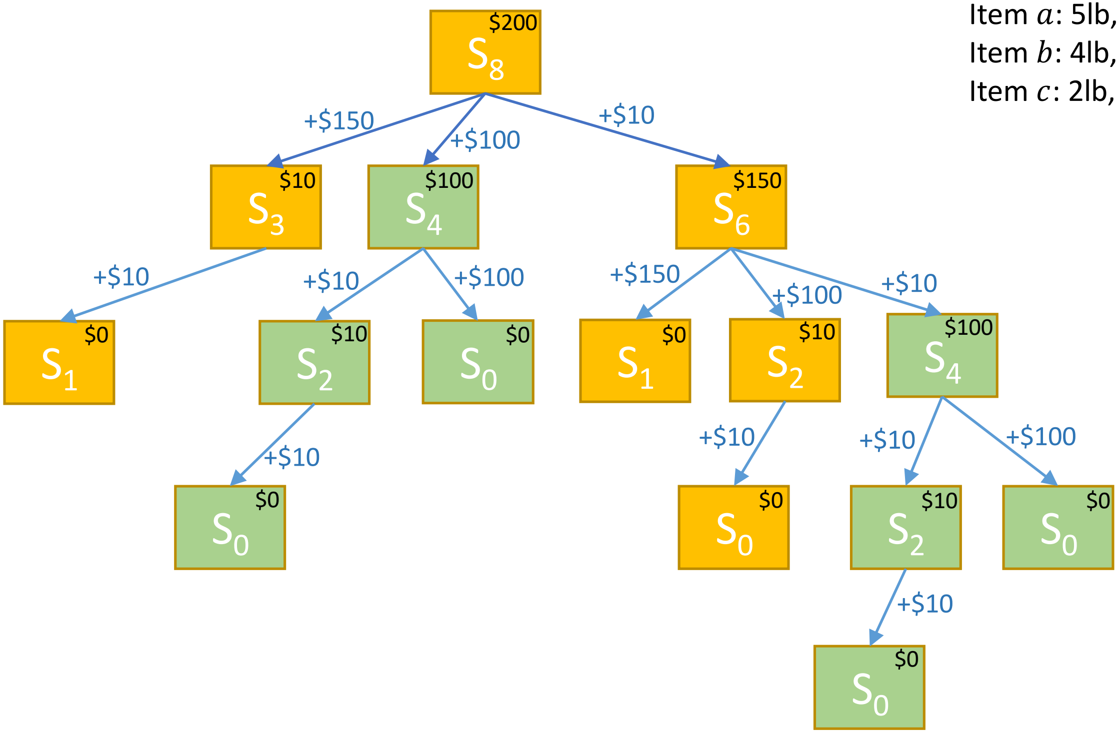
Item a: 5lb, \$150
Item b: 4lb, \$100
Item c: 2lb, \$10



Source: UC Riverside, CS141 course, Fall 2021

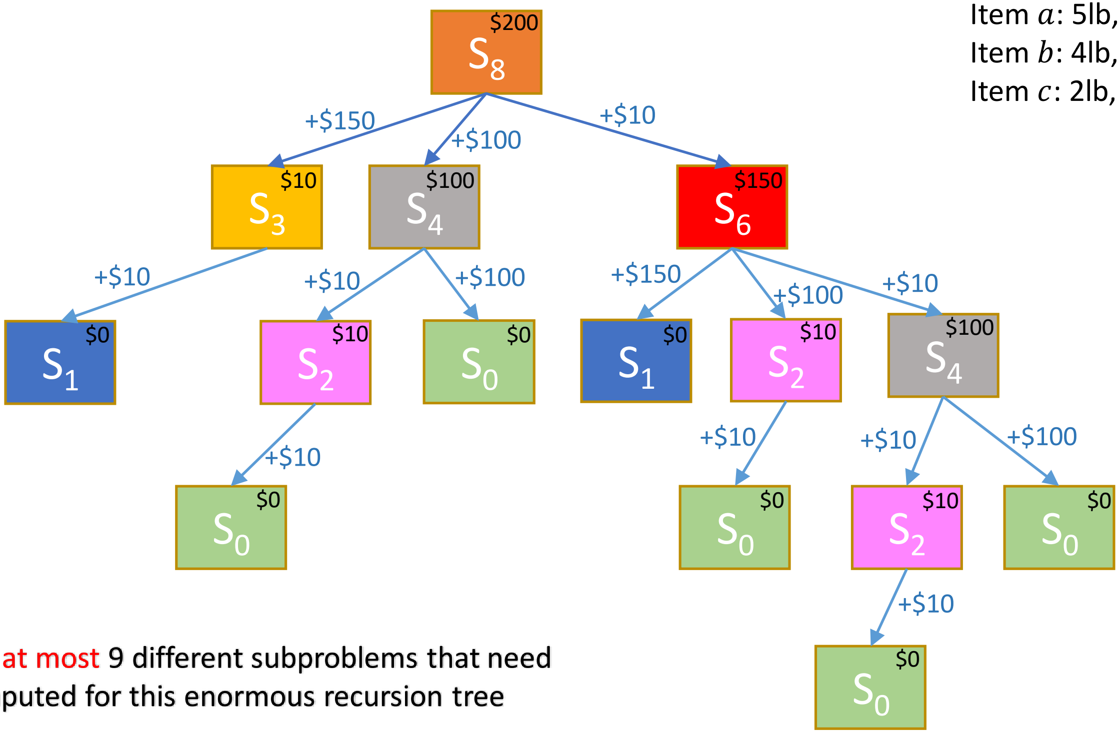
Execution Recurrence Tree

Item a: 5lb, \$150
Item b: 4lb, \$100
Item c: 2lb, \$10



Source: UC Riverside, CS141 course, Fall 2021

Execution Recurrence Tree



Item a: 5lb, $150
Item b: 4lb, $100
Item c: 2lb, $10

There are at most 9 different subproblems that need to be computed for this enormous recursion tree

Source: UC Riverside, CS141 course, Fall 2021

An Efficient Algorithm

Item a: 5 lb, \$150, Item b: 4 lb, \$100, Item c: 2 lb, \$10

```
suitcase(leftWeight)
  curBest = 0
  foreach item of (weight, val)
    if (leftWeight >= weight)
      curBest = max(curBest, suitcase(leftWeight-weight)+val)
  return curBest

// Calling the function
answer = suitcase(8)
```

Previous naïve version

Source: UC Riverside, CS141 course, Fall 2021

An Efficient Algorithm

Item a: 5 lb, $150, Item b: 4 lb, $100, Item c: 2 lb, $10

```
suitcase(leftWeight)
  if (ans[leftWeight] != -1)
    return ans[leftWeight]
  curBest = 0
  foreach item of (weight, val)
    if (leftWeight >= weight)
      curBest = max(curBest, suitcase(leftWeight-weight)+val)
  ans[leftWeight] = curBest
  return curBest

// Calling the function
ans[0,..,8] = {-1,..,-1}
answer = suitcase(8)
```

Source: UC Riverside, CS141 course, Fall 2021

An Efficient Algorithm

Item a: 5 lb, $150, Item b: 4 lb, $100, Item c: 2 lb, $10

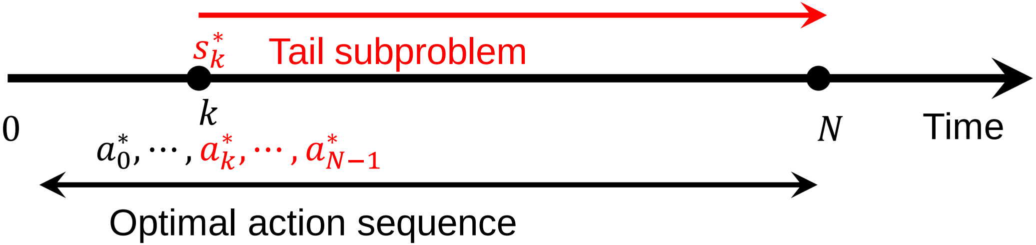
```
suitcase(leftWeight)
  if (ans[leftWeight] != -1)
    return ans[leftWeight]
  curBest = 0
  foreach item of (weight, val)
    if (leftWeight >= weight)
      curBest = max(curBest, suitcase(leftWeight-weight)+val)
  ans[leftWeight] = curBest
  return curBest

// Calling the function
ans[0,..,8] = {-1,..,-1}
answer = suitcase(8)
```

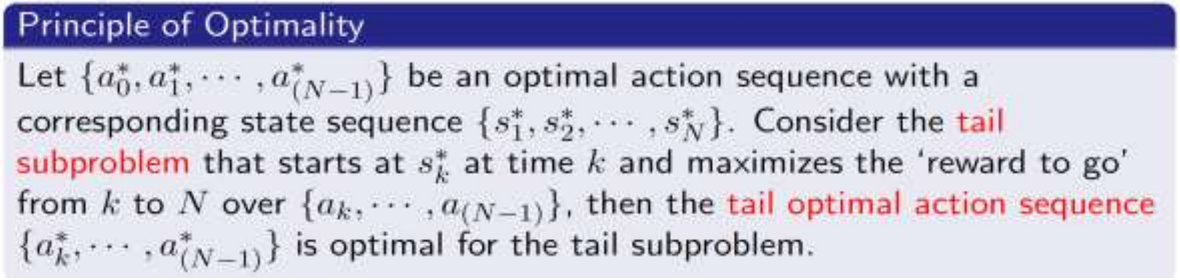
COMPLEXITY??

Source: UC Riverside, CS141 course, Fall 2021

Overlapping Subproblems and Optimal Substructure

- Dynamic Programing addresses a bigger problem by breaking it down as subproblems and then
  - Solving the subproblems
  - Combining solutions to subproblems
- Dynamic Programing is based on the <u>principle of optimality</u>





Principle of Optimality

Let $\{a_0^*, a_1^*, \cdots, a_{(N-1)}^*\}$ be an optimal action sequence with a corresponding state sequence $\{s_1^*, s_2^*, \cdots, s_N^*\}$. Consider the tail subproblem that starts at s_k^* at time k and maximizes the 'reward to go' from k to N over $\{a_k, \cdots, a_{(N-1)}\}$, then the tail optimal action sequence $\{a_k^*, \cdots, a_{(N-1)}^*\}$ is optimal for the tail subproblem.

Requirements of Dynamic Programing

- Optimal substructure i.e., principle of optimality applies.
- Overlapping subproblems, i.e., subproblems recur many times and solutions to these subproblems can be cached and reused
- In the previous example, the problem is to find the optimal value S_w for a total weight of w and the options are:
  suitcase(8)
  - Case 1: First put item a. Total value = suitcase(3) + 150
  - Case 2: First put item b. Total value = suitcase(4) + 100
  - Case 3: First put item c. Total value = suitcase(6) + 10
  - Best = max of the above three
- The subproblems are same problem with new weight limit $w - w_i$
- The subproblems must be the corresponding optimal solutions, otherwise the bigger problem is not going to be optimal

0/1 Knapsack Problem

Item a: 5 lb, $150, Item b: 4 lb, $100, Item c: 2 lb, $10

```
suitcase(leftWeight)
  if (ans[leftWeight] != -1)
    return ans[leftWeight]
  curBest = 0
  foreach item of (weight, val)
    if (leftWeight >= weight)
      curBest = max(curBest, suitcase(leftWeight-weight)+val)
  ans[leftWeight] = curBest
  return curBest


// Calling the function
ans[0,..,8] = {-1,..,-1}
answer = suitcase(8)
```

Will the solution work if we only allow to use an item once?

Source: UC Riverside, CS141 course, Fall 2021

0/1 Knapsack Problem

- No
- Optimal subproblem for “unlimited knapsack”
- After we chose item j, the leftover problem is “best value when weight capacity is $k - w_j$” [We changed k to denote the current capacity]

- In 0/1 Knapsack problem, leftover problem is “best value when weight capacity is $k - w_j$ and we can not use item j again”

- How can we change the state to accommodate this?

- The subproblem we use must not contain item j
- Add another dimension to the problem

Source: UC Riverside, CS141 course, Fall 2021

0/1 Knapsack Problem

- Let $s[i, j]$ be the optimal value for total weight i using only the first j items
- How to calculate $s[i, j]$? For any item, there are really two options
  - Use the item j (value of j + best solution of weight limit $i - w_j$ using first $j - 1$ items)
    $$s\left[i - w_j, j - 1\right] + v_j$$
  - Do not use item j (best solution of weight limit i using first $j - 1$ items)
    $$s[i, j - 1]$$
  - The boundary case: $s[i, 0] = 0$

Source: UC Riverside, CS141 course, Fall 2021

0/1 Knapsack Problem Example

j i	**No item**	**Item 1**	**Item 1 & 2**	**Item 1, 2 & 3**
0				
1				
2				
3				
4				
5				
6				
7				
8				

$s[i][j]$: optimal solution for a knapsack of **capacity i** using **item $1, 2, \ldots, j$**

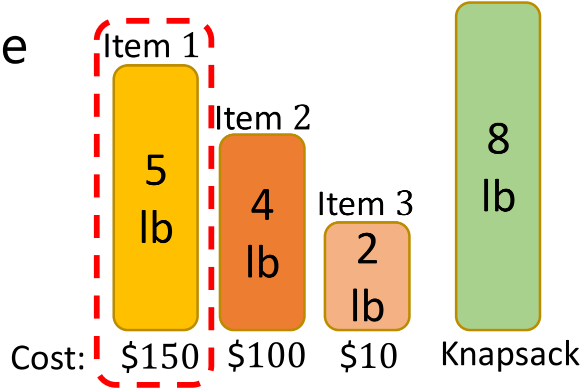
0/1 Knapsack Problem Example

j i	**No item**	**Item 1**	**Item 1 & 2**	**Item 1, 2 & 3**
0	0			
1	0			
2	0			
3	0			
4	0			
5	0			
6	0			
7	0			
8	0			

$s[i][\mathrm{j}]$: optimal solution for a knapsack of **capacity $\boldsymbol{i}$** using **item $\mathbf{1, 2, \ldots, \boldsymbol{j}}$**

0/1 Knapsack Problem Example

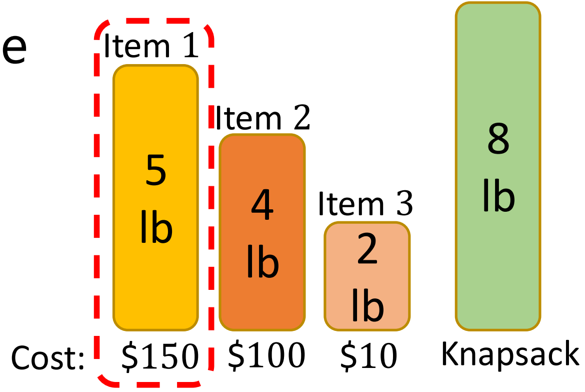
j / i	**No item**	**Item 1**	**Item 1 & 2**	**Item 1, 2 & 3**
0	0			
1	0			
2	0			
3	0			
4	0			
5	0			
6	0			
7	0			
8	0			



$s[i][j]$: optimal solution for a knapsack of **capacity** $\boldsymbol{i}$ using **item** $\mathbf{1, 2, \ldots, \boldsymbol{j}}$

0/1 Knapsack Problem Example

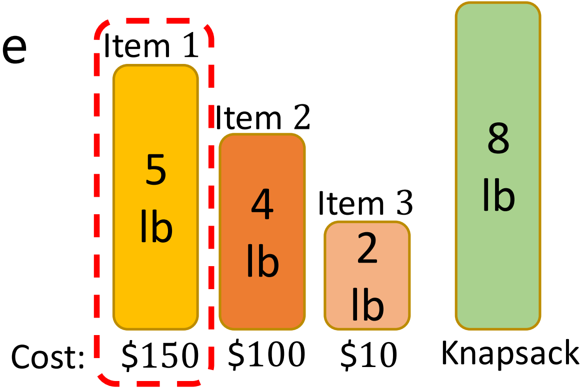
j i	**No item**	**Item 1**	**Item 1 & 2**	**Item 1, 2 & 3**
0	0	0		
1	0	0		
2	0	0		
3	0	0		
4	0	0		
5	0			
6	0			
7	0			
8	0			



$s[i][\mathrm{j}]$: optimal solution for a knapsack of **capacity i** using **item $1, 2, \ldots, j$**

0/1 Knapsack Problem Example

j i	No item	Item 1	Item 1 & 2	Item 1, 2 & 3
0	0	0		
1	0	0		
2	0	0		
3	0	0		
4	0	0		
5	0			
6	0			
7	0			
8	0			



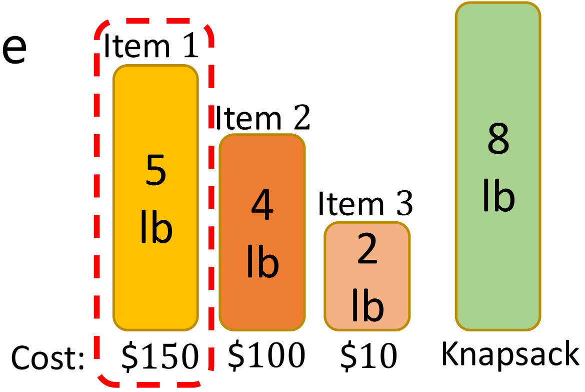
$s[i][j]$: optimal solution for a knapsack of **capacity i** using **item $1, 2, \ldots, j$**

$\text{Opt1} + \text{V}_\text{j} =$

$\text{Opt2} =$

0/1 Knapsack Problem Example

j / *i*	**No item**	**Item 1**	**Item 1 & 2**	**Item 1, 2 & 3**
0	0	0		
1	0	0		
2	0	0		
3	0	0		
4	0	0		
5	0			
6	0			
7	0			
8	0			



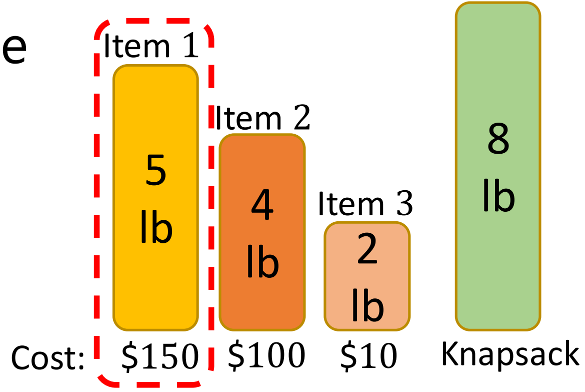
$s[i][j]$: optimal solution for a knapsack of **capacity i** using **item $1, 2, \ldots, j$**

$$\boxed{\text{Opt1}} + V_j = 0 + \$150$$

$$\boxed{\text{Opt2}} = 0$$

0/1 Knapsack Problem Example

j i	**No item**	**Item 1**	**Item 1 & 2**	**Item 1, 2 & 3**
0	0	0		
1	0	0		
2	0	0		
3	0	0		
4	0	0		
5	0	150		
6	0			
7	0			
8	0			



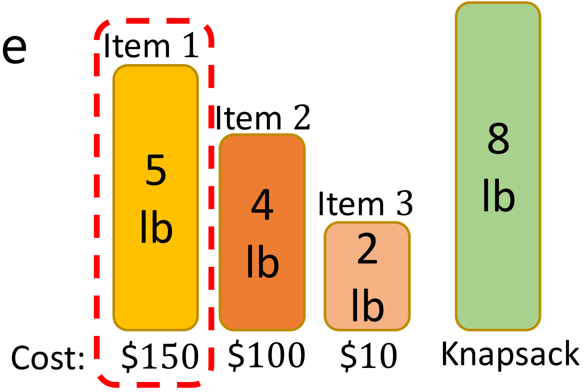
$s[i][j]$: optimal solution for a knapsack of **capacity** $\boldsymbol{i}$ using **item** $\mathbf{1, 2, \ldots, \boldsymbol{j}}$

$\boxed{\text{Opt1}} + V_j = 0 + \$150$

$\boxed{\text{Opt2}} = 0$

0/1 Knapsack Problem Example

j / *i*	**No item**	**Item 1**	**Item 1 & 2**	**Item 1, 2 & 3**
0	0	0		
1	0	0		
2	0	0		
3	0	0		
4	0	0		
5	0	150		
6	0	150		
7	0			
8	0			



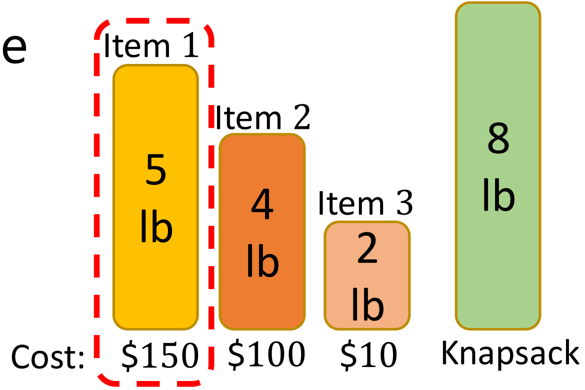
$s[i][j]$: optimal solution for a knapsack of **capacity** $\boldsymbol{i}$ using **item** $\mathbf{1, 2, \ldots, \boldsymbol{j}}$

$\boxed{\text{Opt1}} + \mathrm{V_j} = 0 + \$150$

$\boxed{\text{Opt2}} = 0$

0/1 Knapsack Problem Example

j i	**No item**	**Item 1**	**Item 1 & 2**	**Item 1, 2 & 3**
0	0	0		
1	0	0		
2	0	0		
3	0	0		
4	0	0		
5	0	150		
6	0	150		
7	0	150		
8	0	150		



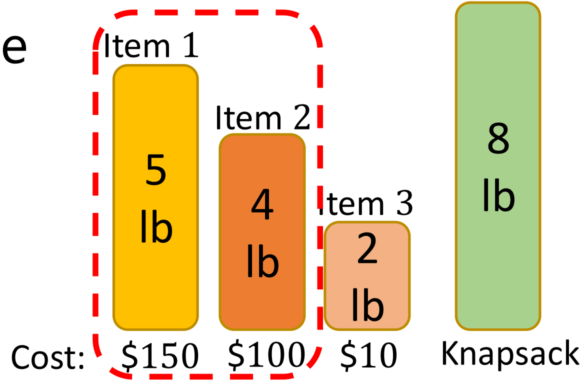
$s[i][j]$: optimal solution for a knapsack of **capacity i** using **item $1, 2, \ldots, j$**

$$\text{Opt1} + \text{V}_\text{j} = 0 + \$150$$

$$\text{Opt2} = 0$$

0/1 Knapsack Problem Example

j / i	No item	Item 1	Item 1 & 2	Item 1, 2 & 3
0	0	0	0	
1	0	0	0	
2	0	0	0	
3	0	0	0	
4	0	0		
5	0	150		
6	0	150		
7	0	150		
8	0	150		



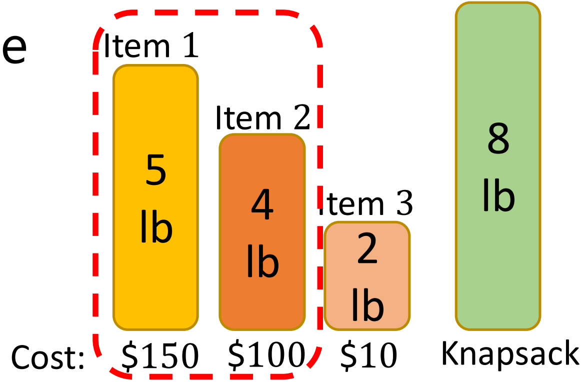
$s[i][j]$: optimal solution for a knapsack of **capacity i** using **item $1, 2, \ldots, j$**

$$\boxed{\text{Opt1}} + \mathrm{V_j} = 0 + \$100$$

$$\boxed{\text{Opt2}} = 0$$

0/1 Knapsack Problem Example

j i	**No item**	**Item 1**	**Item 1 & 2**	**Item 1, 2 & 3**
0	0	0	0	
1	0	0	0	
2	0	0	0	
3	0	0	0	
4	0	0	100	
5	0	150		
6	0	150		
7	0	150		
8	0	150		



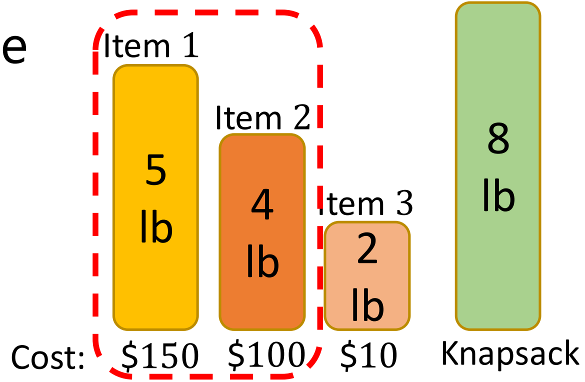
$s[i][j]$: optimal solution for a knapsack of **capacity i** using **item $1, 2, \ldots, j$**

$\boxed{\text{Opt1}} + V_j = 0 + \$100$

$\boxed{\text{Opt2}} = 0$

0/1 Knapsack Problem Example

i \ *j*	No item	Item 1	Item 1 & 2	Item 1, 2 & 3
0	0	0	0	
1	0	0	0	
2	0	0	0	
3	0	0	0	
4	0	0	100	
5	0	150	150	
6	0	150		
7	0	150		
8	0	150		



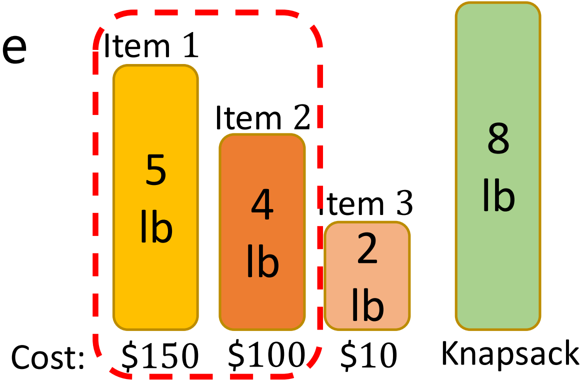
$s[i][j]$: optimal solution for a knapsack of **capacity** $\boldsymbol{i}$ using **item** $\mathbf{1, 2, \ldots, \boldsymbol{j}}$

$\boxed{\text{Opt1}} + V_j = 0 + \$100$

$\boxed{\text{Opt2}} = 150$

0/1 Knapsack Problem Example

j i	No item	Item 1	Item 1 & 2	Item 1, 2 & 3
0	0	0	0	
1	0	0	0	
2	0	0	0	
3	0	0	0	
4	0	0	100	
5	0	150	150	
6	0	150	150	
7	0	150	150	
8	0	150	150	



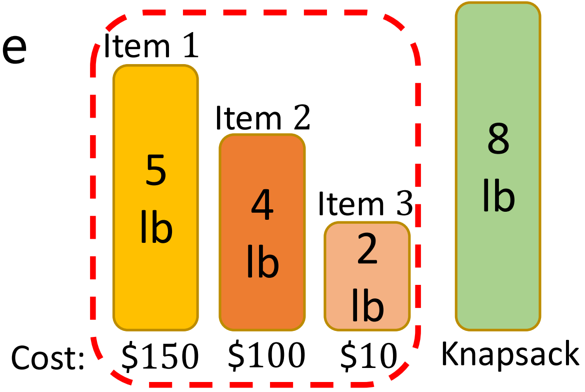
$s[i][j]$: optimal solution for a knapsack of **capacity i** using **item 1, 2, … , j**

$$\mathrm{Opt1} + \mathrm{V_j} = 0 + \$100$$

$$\mathrm{Opt2} = 0$$

0/1 Knapsack Problem Example

j / i	No item	Item 1	Item 1 & 2	Item 1, 2 & 3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	
3	0	0	0	
4	0	0	100	
5	0	150	150	
6	0	150	150	
7	0	150	150	
8	0	150	150	



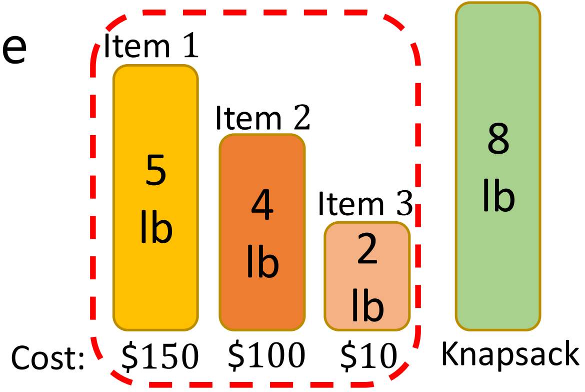
$s[i][j]$: optimal solution for a knapsack of **capacity i** using **item 1, 2, … , j**

$$\boxed{\text{Opt1}} + V_j = 0 + \$10$$

$$\boxed{\text{Opt2}} = 0$$

0/1 Knapsack Problem Example

j i	No item	Item 1	Item 1 & 2	Item 1, 2 & 3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	10
3	0	0	0	
4	0	0	100	
5	0	150	150	
6	0	150	150	
7	0	150	150	
8	0	150	150	



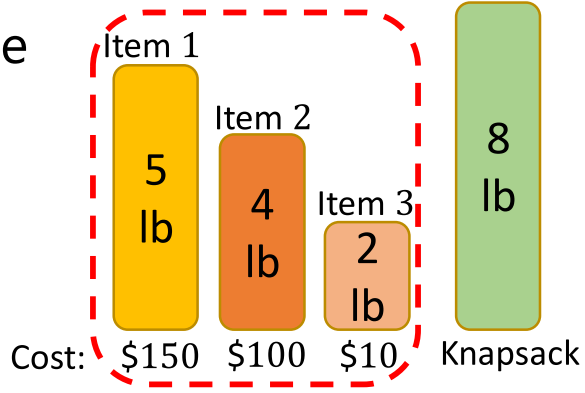
$s[i][j]$: optimal solution for a knapsack of **capacity i** using **item $1, 2, \dots, j$**

$\boxed{\text{Opt1}} + V_j = 0 + \$10$

$\boxed{\text{Opt2}} = 0$

0/1 Knapsack Problem Example

j i	No item	Item 1	Item 1 & 2	Item 1, 2 & 3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	10
3	0	0	0	10
4	0	0	100	
5	0	150	150	
6	0	150	150	
7	0	150	150	
8	0	150	150	



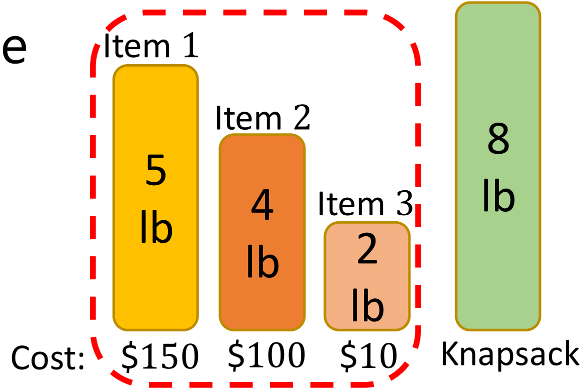
$s[i][j]$: optimal solution for a knapsack of **capacity i** using **item 1, 2, … , j**

$\boxed{\text{Opt1}} + V_j = 0 + \$10$

$\boxed{\text{Opt2}} = 100$

0/1 Knapsack Problem Example

j / i	**No item**	**Item 1**	**Item 1 & 2**	**Item 1, 2 & 3**
0	0	0	0	0
1	0	0	0	0
2	0	0	$\boxed{0}$	10
3	0	0	0	10
4	0	0	$\boxed{100}$	$\boxed{100}$
5	0	150	150	
6	0	150	150	
7	0	150	150	
8	0	150	150	



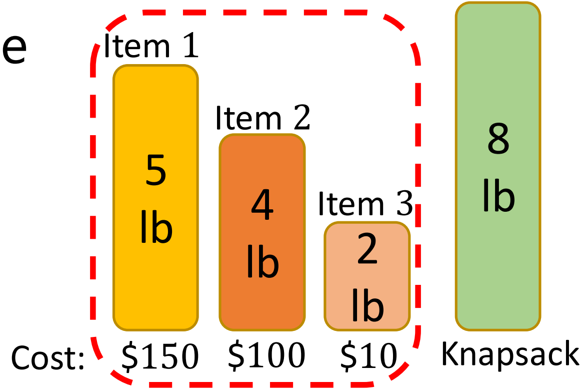
$s[i][j]$: optimal solution for a knapsack of **capacity i** using **item $1, 2, \ldots, j$**

$\boxed{\text{Opt1}} + V_j = 0 + \$10$

$\boxed{\text{Opt2}} = 100$

0/1 Knapsack Problem Example

j i	**No item**	**Item 1**	**Item 1 & 2**	**Item 1, 2 & 3**
0	0	0	0	0
1	0	0	0	0
2	0	0	0	10
3	0	0	0	10
4	0	0	100	100
5	0	150	150	150
6	0	150	150	150
7	0	150	150	160
8	0	150	150	



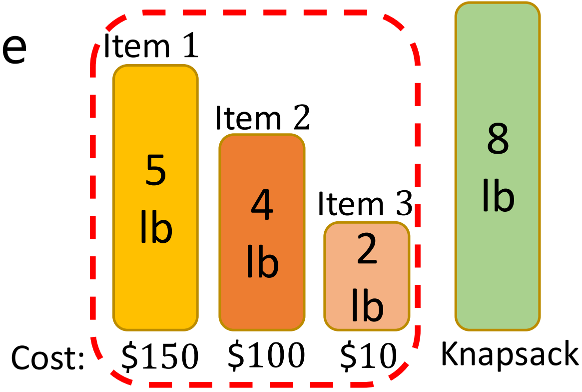
$s[i][j]$: optimal solution for a knapsack of **capacity** $\boldsymbol{i}$ using **item** $\mathbf{1, 2, \ldots, \boldsymbol{j}}$

$\boxed{\text{Opt1}} + V_j = 150 + \$10$

$\boxed{\text{Opt2}} = 150$

0/1 Knapsack Problem Example

j i	No item	Item 1	Item 1 & 2	Item 1, 2 & 3
0	0	0	0	0
1	0	0	0	0
2	0	0	0	10
3	0	0	0	10
4	0	0	100	100
5	0	150	150	150
6	0	150	150	150
7	0	150	150	160
8	0	150	150	160



$s[i][\mathrm{j}]$: optimal solution for a knapsack of **capacity *i*** using **item 1, 2, … , *j***

$\boxed{\text{Opt1}} + \mathrm{V_j} = 150 + \$10$

$\boxed{\text{Opt2}} = 150$

0/1 Knapsack Problem Pseudocode

```
suitcase(i,j) //i is the capacity, j is the item
  if (ans[i][j] != -1)
    return ans[i][j] //If soln to subproblem already exists
  if (j==0) //Base case
    ans[i][j] = 0
    return ans[i][j]
  best = suitcase(i,j-1) //If j is not taken
  if (i >= weight[j]) //If item j is taken
      best = max(best, suitcase(i-weight[j],j-1)+val[j])
  ans[i][j] = best
  return ans[i][j]

// Calling the function
ans[k][n] = {-1,..,-1}
answer = suitcase(8,3)
```

Source: UC Riverside, CS141 course, Fall 2021

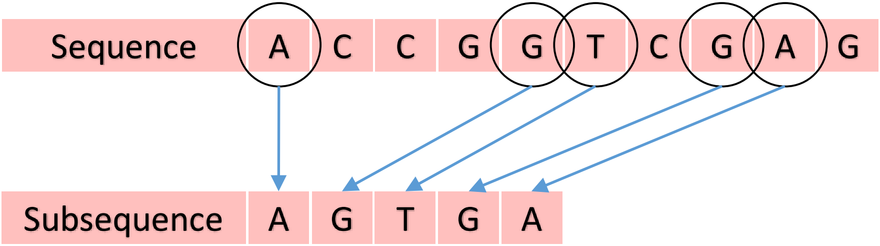
0/1 Knapsack Problem Pseudocode

```
getitems()
  i = k
  for j=n downto 1
    if ans[i][j] > ans[i][j-1]
      Take item j
      i = i - weight[j] //Move to the right row
```

Longest Common Subsequence

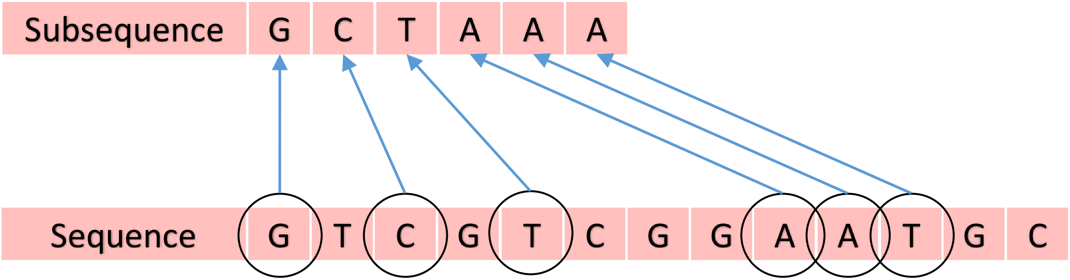
- A strand of DNA can be viewed as a string of bases {A, C, G, T}
    - S_1=ACCGGTCGAGT
    - S_2=GTCGTTCGGAATGC
- Biological applications often need to compare the DNA of two or more organisms to determine how "similar" they are as some measure of how closely related the two organisms are
- Two strands are similar if
    - One is a substring of another
    - Number of changes needed to turn one into the other is small
    - A third string is long enough and has bases appearing in each of the original two in the same order, but not necessarily consecutively
- Longest Common Subsequence (LCS) deals with the third way

Longest Common Subsequence



Sequence A C C G G T C G A G

Subsequence A G T G A

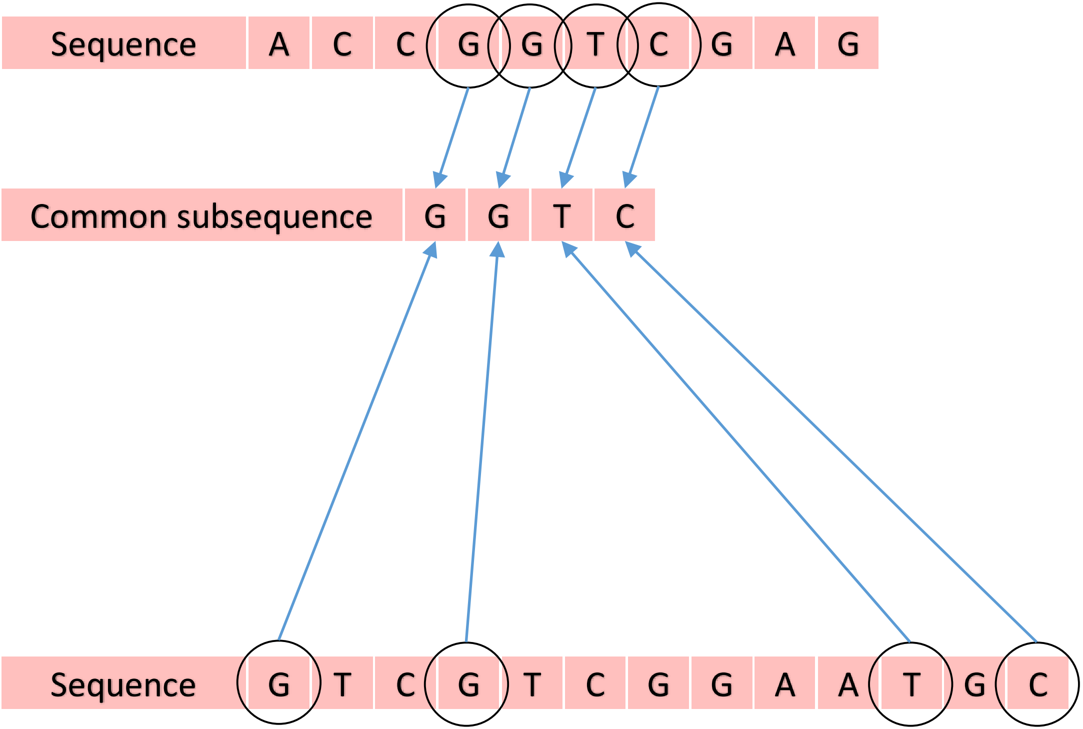


Subsequence G C T A A A

Sequence G T C G T C G G A A T G C

Source: UC Riverside, CS141 course, Fall 2021

Longest Common Subsequence



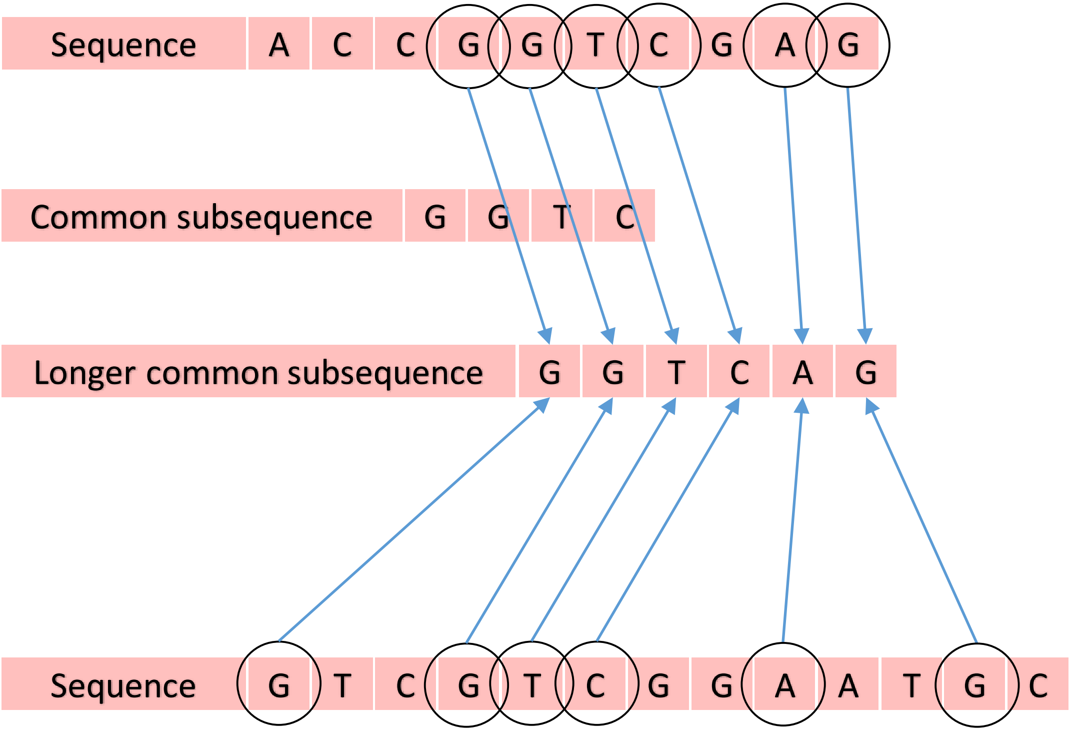
Sequence A C C G G T C G A G

Common subsequence G G T C

Sequence G T C G T C G G A A T G C

Source: UC Riverside, CS141 course, Fall 2021

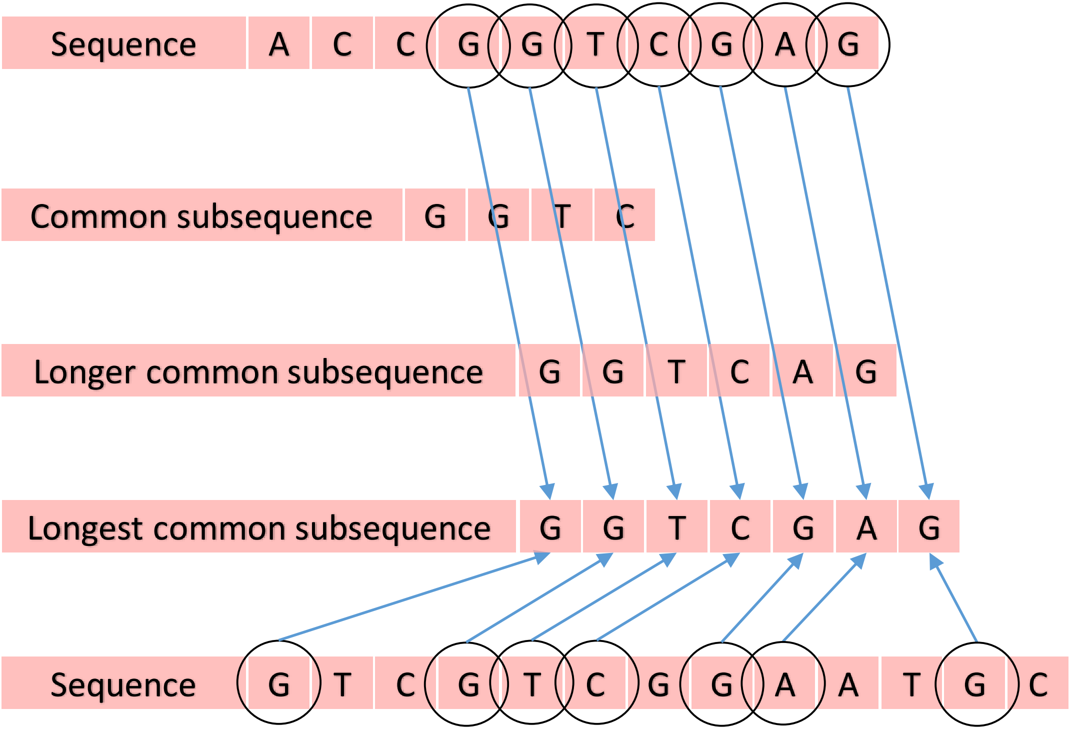
Longest Common Subsequence



Sequence	A	C	C	G	G	T	C	G	A	G			
Common subsequence	G	G	T	C									
Longer common subsequence	G	G	T	C	A	G							
Sequence	G	T	C	G	T	C	G	G	A	A	T	G	C

Source: UC Riverside, CS141 course, Fall 2021

Longest Common Subsequence



Sequence A C C G G T C G A G

Common subsequence G G T C

Longer common subsequence G G T C A G

Longest common subsequence G G T C G A G

Sequence G T C G T C G G A A T G C

Source: UC Riverside, CS141 course, Fall 2021

Problem Definition

- Input: Two sequences X and Y. For example,
  - $X =< A, B, C, B, D, A, B >$
  - $Y =< B, D, C, A, B, A >$
- Output: Longest common subsequence Z of X and Y
  - For X and Y above, the longest common subsequence is $Z =< B, C, B, A >$

- What will be a naïve way to solve it?

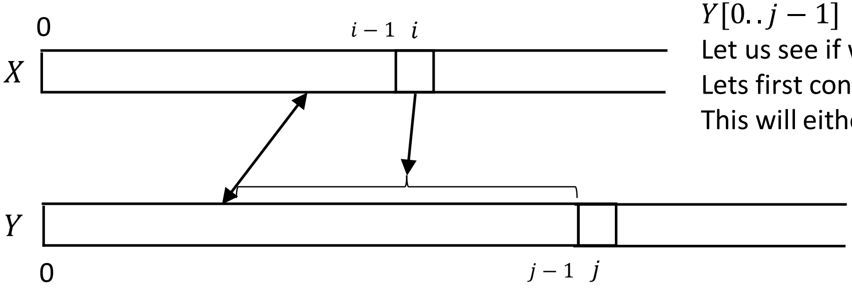
- What are the "Subproblems" here?
- What is the possible "last move"?
  - For ABCBDAB and BDCABA, we want to know, given the LCS between "ABCBDA" and "BDCAB", how should we deal with the last element 'B' and 'A' respectively?

Lets Focus on the Last Characters

- Lets compare the last character $X[i]$ and $Y[j]$

Good news. Whatever was the LCS between the two prefixes, considering the last elements of the two, increases it by 1

- What if $X[i] = Y[j]$?
  - ABCBDA and BDCABA
- What if $X[i] \neq Y[j]$?
  - ABCBDAB and BDCABA
- Let us take a closer look at this



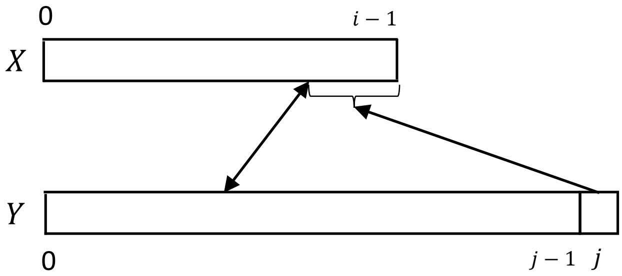
We already have found the LCS for $X[0..i-1]$ and $Y[0..j-1]$
Let us see if we add the elements what happens.
Lets first consider the pair $X[0..i]$ and $Y[0..j-1]$
This will either produce one additional match or not

Lets Focus on the Last Characters

- Lets compare the last character $X[i]$ and $Y[j]$

Good news. Whatever was the LCS between the two prefixes, considering the last elements of the two, increases it by 1

- What if $X[i] = Y[j]$?
  - ABCBDA and BDCABA
- What if $X[i] \neq Y[j]$?
  - ABCBDAB and BDCABA
- Let us take a closer look at this



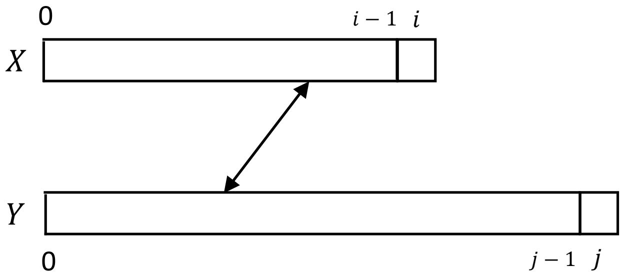
We already have found the LCS for $X[0..i-1]$ and $Y[0..j-1]$
Let us see if we add the elements what happens.
Lets first consider the pair $X[0..i]$ and $Y[0..j-1]$
This will either produce one additional match or not
Lets now consider the pair $X[0..i-1]$ and $Y[0..j]$
This will either produce one additional match or not

Lets Focus on the Last Characters

- Lets compare the last character $X[i]$ and $Y[j]$
- What if $X[i] = Y[j]$?
    - ABCBDA and BDCABA

Good news. Whatever was the LCS between the two prefixes, considering the last elements of the two, increases it by 1

- What if $X[i] \neq Y[j]$?
    - ABCBDAB and BDCABA
- Let us take a closer look at this



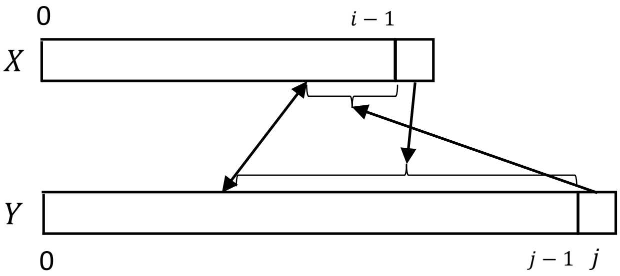
We already have found the LCS for $X[0..i-1]$ and $Y[0..j-1]$
Let us see if we add the elements what happens.
Lets first consider the pair $X[0..i]$ and $Y[0..j-1]$
This will either produce one additional match or not
Lets now consider the pair $X[0..i-1]$ and $Y[0..j]$
This will either produce one additional match or not

Do we now compare $X[0..i]$ and $Y[0..j]$?
No. As, given we have compared the other two cases, this will only make any difference only if $X[i]$ and $Y[j]$ are equal – which we assume are not

Lets Focus on the Last Characters

- Lets compare the last character $X[i]$ and $Y[j]$

- What if $X[i] = Y[j]$?
  - ABCBDA and BDCABA
- What if $X[i] \neq Y[j]$?
  - ABCBDAB and BDCABA
- Let us take a closer look at this



So, in summary, we have to compare $X[0..i-1]$ and $Y[0..j]$ and also $X[0..i]$ and $Y[0..j-1]$ to get the LCS between $X[0..i]$ and $Y[0..j]$

Now, do we need to scan and check every element of the other sequence to know if any match is being added?

No – as both the comparisons ($X[0..i-1]$ with $Y[0..j]$ and $X[0..i]$ with $Y[0..j-1]$) have been made already and stored.

We can reuse them. We have to take the maximum out of the two cases

Solution to LCS – An Example

- Lets use $s[i,j]$ to denote the LCS of the first i characters in X and first j characters in Y
- If $X[i] = Y[j]$
  - $s[i,j] = s[i-1,j-1] + 1$

Index	1	2	3	4	5	6	7
X=	A	B	C	B	D	A	B
				↑			
Y=	B	D	C	A	B	A	
					↑		

LCS of “ABCB” and “BDCAB” must be:
(the LCS of “ABC” and “BDCA”) + “B”

  - $s[4,5] = s[3,4] + 1$

Solution to LCS – An Example

- Lets use $s[i,j]$ to denote the LCS of the first i characters in X and first j characters in Y
- If $X[i] \neq Y[j]$
  - $s[i,j] = \max(s[i-1,j], s[i,j-1])$

Index	1	2	3	4	5	6	7
X=	A	B	C	B	D	A	B
Y=	B	D	C	A	B	A	

  - $s[3,5] = \max(s[2,5], s[3,4])$

- There are really three choices
  - Add $X[i]$ as the new entry to the LCS
  - Add $Y[j]$ as the new entry to the LCS
  - Discard both $X[i]$ and $Y[j]$

LCS of “ABC” and “BDCAB” can be:
the LCS of “AB” and “BDCAB”
the LCS of “ABC” and “BDCA”
the LCS of “AB” and “BDCA”
(included above)

Solution to LCS – An Example

- $s[i,0] = 0; s[0,j] = 0$
- $s[i,j] = \begin{cases} s[i-1,j-1] + 1 \; if \; X[i] = Y[j] \\ \max(s[i-1,j], s[i,j-1]) \,, if \; X[i] \neq Y[j] \end{cases}$

j

i

		0	1	2	3	4	5	6
		None	B	D	C	A	B	A
0	None							
1	A							
2	B							
3	C							
4	B							
5	D							
6	A							
7	B							

Solution to LCS – An Example

- $s[i,0] = 0; s[0,j] = 0$
- $s[i,j] = \begin{cases} s[i-1,j-1]+1 \; if \; X[i] = Y[j] \\ \max(s[i-1,j], s[i,j-1])\,, if \; X[i] \neq Y[j] \end{cases}$

i	j	0	1	2	3	4	5	6
		None	B	D	C	A	B	A
0	None	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Solution to LCS – An Example

- $s[i,0]=0; s[0,j]=0$
- $s[i,j]=\begin{cases} s[i-1,j-1]+1 \; if \; X[i]=Y[j] \\ \max(s[i-1,j], s[i,j-1]) \, , if \; X[i]\neq Y[j] \end{cases}$

i	j	0	1	2	3	4	5	6
		None	B	D	C	A	B	A
0	None	0	0	0	0	0	0	0
1	A	0	↑ 0					
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

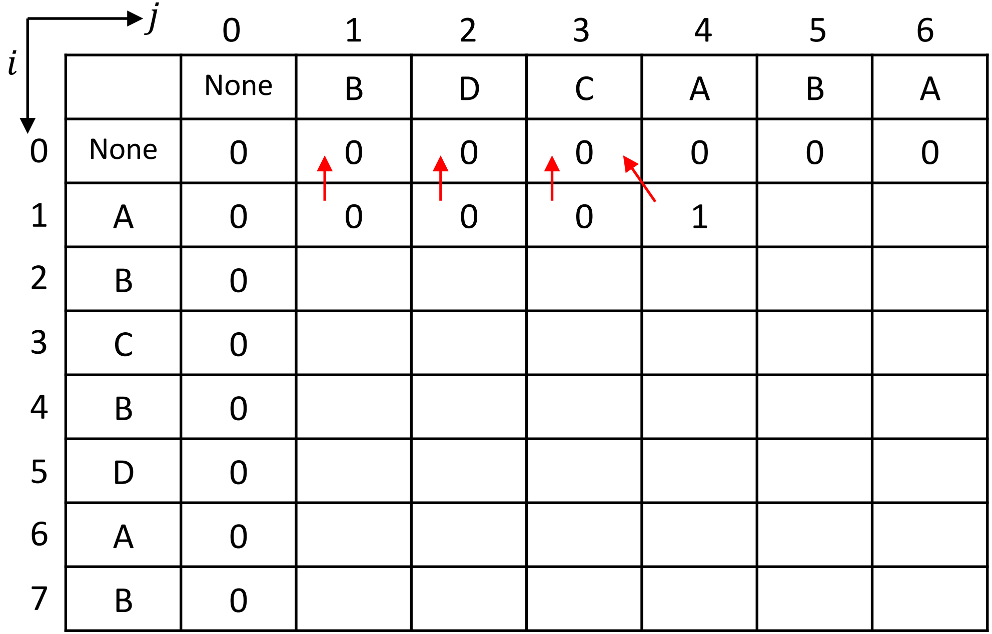
Solution to LCS – An Example

- $s[i,0] = 0; s[0,j] = 0$
- $s[i,j] = \begin{cases} s[i-1,j-1]+1 \; if \; X[i] = Y[j] \\ \max(s[i-1,j], s[i,j-1]) \, , if \; X[i] \neq Y[j] \end{cases}$

j		0	1	2	3	4	5	6
i		None	B	D	C	A	B	A
0	None	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0			
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Solution to LCS – An Example

- $s[i,0] = 0; s[0,j] = 0$
- $s[i,j] = \begin{cases} s[i-1,j-1]+1 \; if \; X[i] = Y[j] \\ \max(s[i-1,j], s[i,j-1]) \, , if \; X[i] \neq Y[j] \end{cases}$



j		0	1	2	3	4	5	6
i		None	B	D	C	A	B	A
0	None	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						

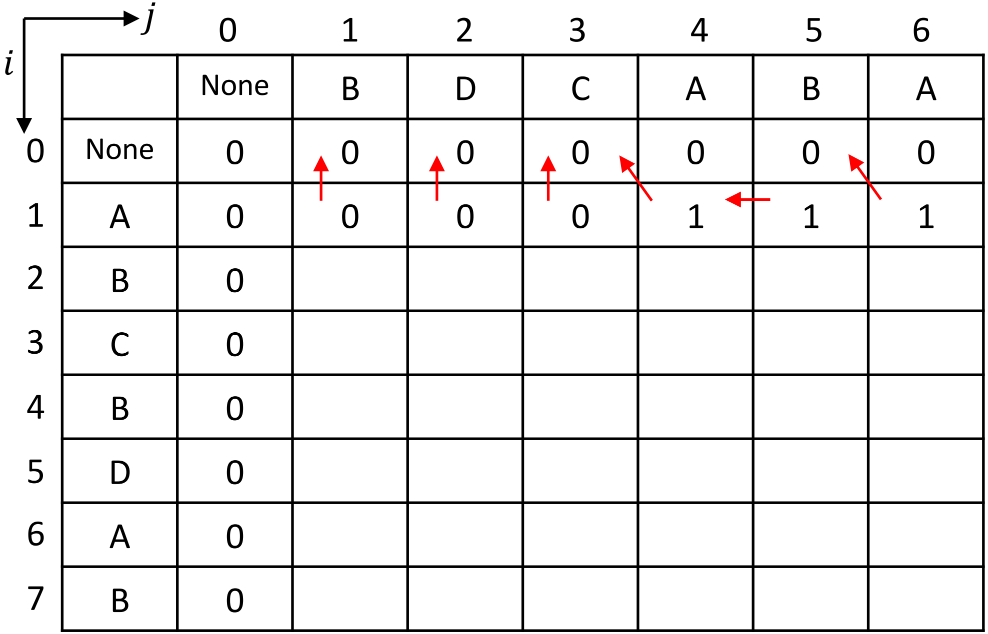
Solution to LCS – An Example

- $s[i,0] = 0; s[0,j] = 0$
- $s[i,j] = \begin{cases} s[i-1,j-1]+1 \; if \; X[i] = Y[j] \\ \max(s[i-1,j], s[i,j-1]) \, , if \; X[i] \neq Y[j] \end{cases}$

i ↓	j →	0	1	2	3	4	5	6
		None	B	D	C	A	B	A
0	None	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Solution to LCS – An Example

- $s[i,0] = 0; s[0,j] = 0$
- $s[i,j] = \begin{cases} s[i-1,j-1]+1 \; if \; X[i] = Y[j] \\ \max(s[i-1,j], s[i,j-1]) \, , if \; X[i] \neq Y[j] \end{cases}$



		0	1	2	3	4	5	6
		None	B	D	C	A	B	A
0	None	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Solution to LCS – An Example

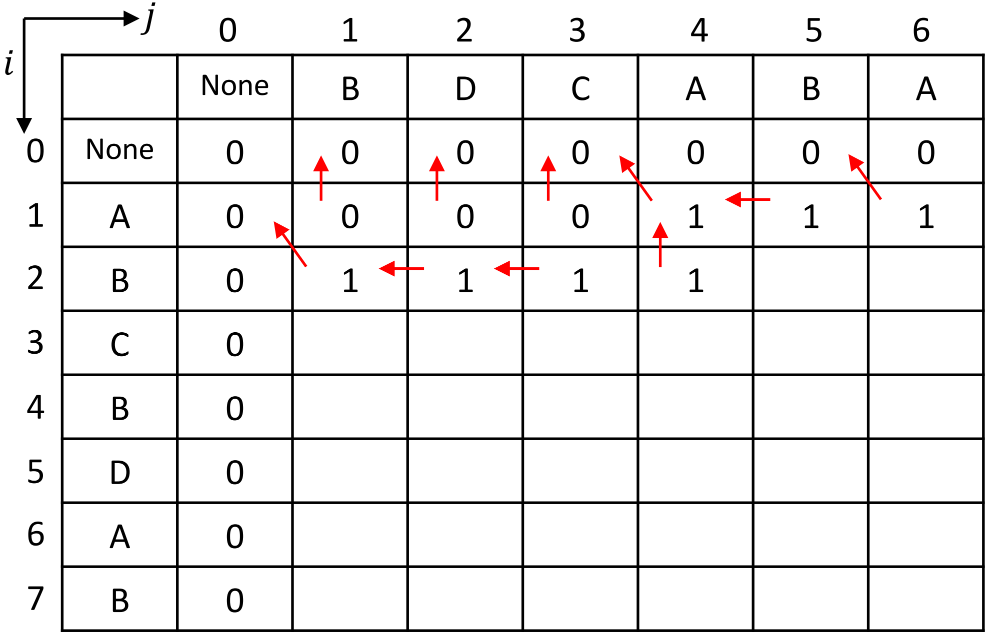
- $s[i,0] = 0; s[0,j] = 0$
- $s[i,j] = \begin{cases} s[i-1,j-1]+1 \; if \; X[i] = Y[j] \\ \max(s[i-1,j], s[i,j-1]) \, , if \; X[i] \neq Y[j] \end{cases}$

$i \downarrow$ $\; \rightarrow j$

		0	1	2	3	4	5	6
		None	B	D	C	A	B	A
0	None	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1					
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Solution to LCS – An Example

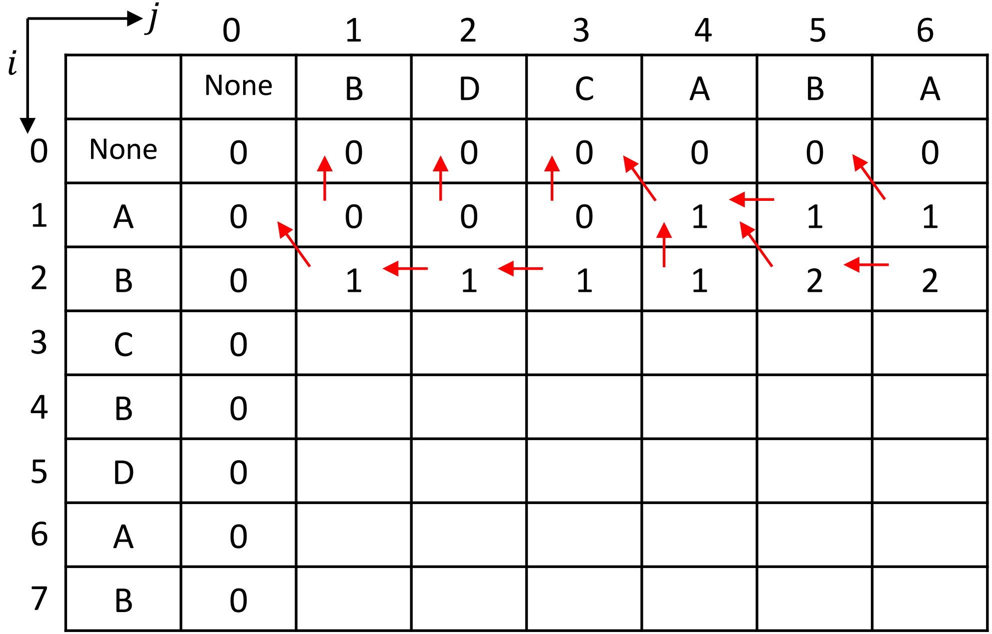
- $s[i,0] = 0; s[0,j] = 0$
- $s[i,j] = \begin{cases} s[i-1,j-1]+1 \; if \; X[i] = Y[j] \\ \max(s[i-1,j], s[i,j-1])\,, if \; X[i] \neq Y[j] \end{cases}$



		0	1	2	3	4	5	6
		None	B	D	C	A	B	A
0	None	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1		
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Solution to LCS – An Example

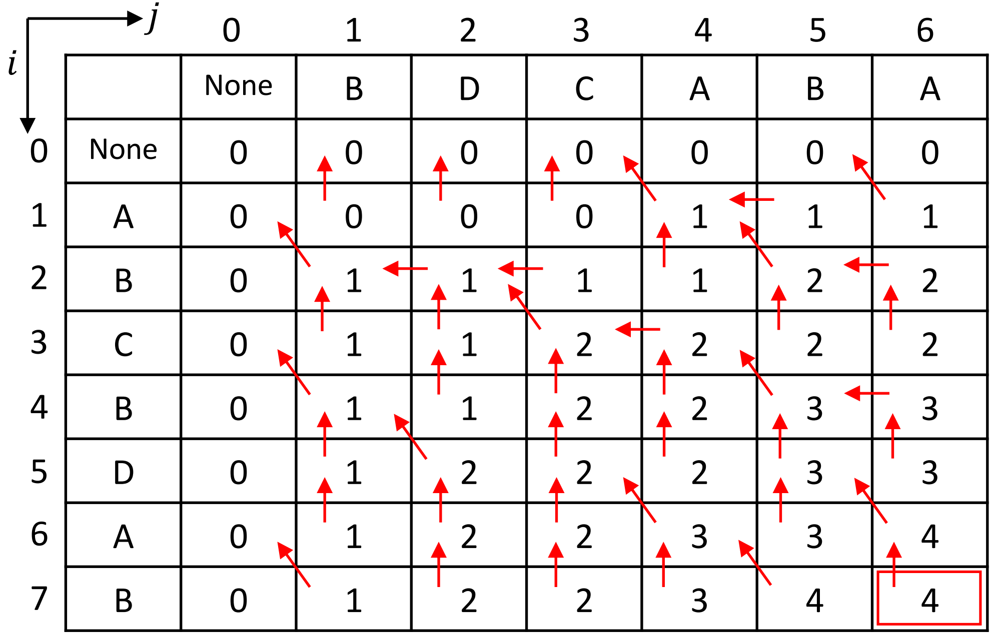
- $s[i,0] = 0; s[0,j] = 0$
- $s[i,j] = \begin{cases} s[i-1,j-1]+1 \; if \; X[i]=Y[j] \\ \max(s[i-1,j], s[i,j-1]) \,, if \; X[i] \neq Y[j] \end{cases}$



		0	1	2	3	4	5	6
		None	B	D	C	A	B	A
0	None	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						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Solution to LCS – An Example

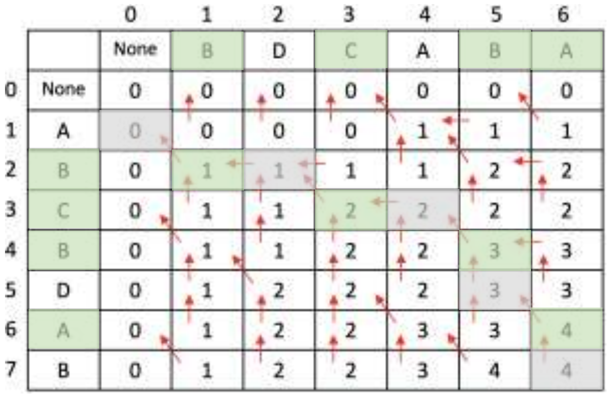
- $s[i,0] = 0; s[0,j] = 0$
- $s[i,j] = \begin{cases} s[i-1,j-1]+1 \; if \; X[i]=Y[j] \\ \max(s[i-1,j], s[i,j-1]) \, , if \; X[i] \neq Y[j] \end{cases}$



$i \downarrow$ $j \rightarrow$		0	1	2	3	4	5	6
		None	B	D	C	A	B	A
0	None	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(X,Y)=4

Solution to LCS – An Example



		0	1	2	3	4	5	6
		None	B	D	C	A	B	A
0	None	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

- How do we know which characters are in the LCS?
- The ones with diagonal arrow are taken
- Even without storing arrow directions separately, how can we know whether a cell has diagonal arrow?
- Compare the characters
- The LCS is BCBA

Solution to LCS – Pseudocode

```
LCS(X,Y)
  m = X.length, n = Y.length
  Create 2-D array s with m+1 rows and n+1 columns
  for i=1 to m //Fill up the first column with 0's
    s[i,0] = 0
  for j=1 to n //Fill up the first row with 0's
    s[0,j] = 0
  for i=1 to m
    for j=1 to n
      if X[i]==Y[j]
        s[i,j] = s[i-1,j-1] + 1
      else
        if s[i-1,j] >= s[i,j-1]
          s[i,j] = s[i-1,j]
        else
          s[i,j] = s[i,j-1]
```

What is the time complexity?

$\Theta(mn)$

Solution to LCS – Pseudocode

```
ConstructLCS(X,Y,s)
  m = X.length, n = Y.length
  i = m, j = n //Start at the last cell of s
  while (i>0 || j>0)
    if X[i] == Y[j]
      Add X[i] to LCS string
      i--; j--
    else
      if s[i-1,j] >= s[i,j-1]
        i--
      else
        j--
```

What is the time complexity?

$$O(m+n)$$

Matrix Chain Multiplication

- How long does it take to multiply matrices?
- Lets consider $C = A \times B$, where A is an $m \times p$ and B is an $p \times n$ matrix
- Then, C is an $m \times n$ matrix and $c_{ij} = \sum_k a_{ik} * b_{kj}$
- That means for each entry of C, we need to multiply p times and there are mn terms. So there needs to be mpn multiplications

- Next suppose that we want to multiply three matrices $A_1 A_2 A_3$
- We can do it in two different ways, $A_1(A_2 A_3)$ or $(A_1 A_2)A_3$
- How long does it take?

Example

- Let A_1 is 2×3, A_2 is 3×4 and A_3 is 4×2

$$A_1 \quad \times \quad A_2 \quad \times \quad A_3$$

$$\begin{array}{cccccc} 2 & 3 & 3 & 4 & 4 & 2 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 \end{array}$$

$$\boxed{\boxed{(A_1 \quad \times \quad A_2)} \quad \times \quad A_3}$$

$$\begin{array}{cccccc} 2 & 3 & 3 & 4 & 4 & 2 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 \end{array}$$

$$2 \times 3 \times 4 = 24$$

$$2 \times 4 \times 2 = 16$$

$$24 + 16 = 40$$

$$\boxed{A_1 \quad \times \quad \boxed{(A_2 \quad \times \quad A_3}})$$

$$\begin{array}{cccccc} 2 & 3 & 3 & 4 & 4 & 2 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 \end{array}$$

$$3 \times 4 \times 2 = 24$$

$$2 \times 3 \times 2 = 12$$

$$24 + 12 = 36$$

- Multiplication order matters!!

Problem Definition

- Find the order to multiply matrices $A_1 A_2 A_3 \dots A_n$ that requires the fewest total operations
- In particular, assume A_1 is an $d_0 \times d_1$ matrix, A_2 is an $d_1 \times d_2$ matrix, generally, A_k is an $d_{k-1} \times d_k$ matrix
- Let us denote the cost (expressed in number of scalar multiplications) of multiplying matrices $A_i A_{i+1} A_{i+2} \dots A_j$ as $c[i, j]$

Observation

- Lets go back to our example and try to find some trend

$$\boxed{\boxed{(A_1 \quad \times \quad A_2)} \quad \times \quad \boxed{A_3}}$$

$$\begin{array}{cccccc} 2 & 3 & 3 & 4 & 4 & 2 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 \end{array}$$

$$2 \times 3 \times 4 = 24 \qquad 0$$

$$2 \times 4 \times 2 = 16$$

$$24 + 0 + \mathbf{16 = 40} \;\dots (1)$$

$$\boxed{\boxed{A_1} \quad \times \quad \boxed{(A_2 \quad \times \quad A_3}\,)}$$

$$\begin{array}{cccccc} 2 & 3 & 3 & 4 & 4 & 2 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 \end{array}$$

$$0 \qquad 3 \times 4 \times 2 = 24$$

$$2 \times 3 \times 2 = 12$$

$$0 + \mathbf{24 + 12 = 36} \;\dots (2)$$

- First of all, we can think of the operations on single matrix costs 0
- Notice that $c[1,2] = 24$ and $c[3,3] = 0$
- Strictly speaking, $c[3,3]$ should represent the cost of multiplying A_3 with itself (that gives you A_3^2), but we will slightly abuse it here
- Using these notations, (1) can be written as $c[1,2] + c[3,3] + 2 \times 4 \times 2 = c[1,3]$
- $\Rightarrow c[1,2] + c[3,3] + d_0 \times d_2 \times d_3 = c[1,3]$

Observation

- Lets go back to our example and try to find some trend

$$\boxed{\boxed{(A_1 \times A_2)} \times \boxed{A_3}}$$

$$\begin{array}{cccccc} 2 & 3 & 3 & 4 & 4 & 2 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 \end{array}$$

$$2 \times 3 \times 4 = 24 \qquad 0$$

$$2 \times 4 \times 2 = 16$$

$$24 + 0 + 16 = 40 \ \dots (1)$$

$$\boxed{\boxed{A_1} \times \boxed{(A_2 \times A_3}})$$

$$\begin{array}{cccccc} 2 & 3 & 3 & 4 & 4 & 2 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 \end{array}$$

$$0 \qquad 3 \times 4 \times 2 = 24$$

$$2 \times 3 \times 2 = 12$$

$$0 + 24 + 12 = 36 \ \dots (2)$$

- $c[1,2] + c[3,3] + d_0 \times d_2 \times d_3 = c[1,3]$
- Similarly, (2) can be written as $c[1,1] + c[2,3] + d_0 \times d_1 \times d_3 = c[1,3]$
- $c[\underset{i}{1},\underset{k}{2}] + c[\underset{k+1}{3},\underset{j}{3}] + \underset{d_{i-1}}{d_0} \times \underset{d_k}{d_2} \times \underset{d_j}{d_3} = c[\underset{i}{1},\underset{j}{3}]$
- $c[\underset{i}{1},\underset{k}{1}] + c[\underset{k+1}{2},\underset{j}{3}] + \underset{d_{i-1}}{d_0} \times \underset{d_k}{d_1} \times \underset{d_j}{d_3} = c[\underset{i}{1},\underset{j}{3}]$

Observation

- $c[\underset{i}{1},\underset{k}{2}] + c[\underset{k+1}{3},\underset{j}{3}] + \underset{d_{i-1}}{d_0} \times \underset{d_k}{d_2} \times \underset{d_j}{d_3} = c[\underset{i}{1},\underset{j}{3}]$ … (1)
- $c[\underset{i}{1},\underset{k}{1}] + c[\underset{k+1}{2},\underset{j}{3}] + \underset{d_{i-1}}{d_0} \times \underset{d_k}{d_1} \times \underset{d_j}{d_3} = c[\underset{i}{1},\underset{j}{3}]$ … (2)
- $c[i,j] = c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j$
- What are the possible values of k?
- $i \leq k < j$
- Finally, we will take the minimum of the $c[i,j]$ over these k's
- $c[i,j] = \min_{i \leq k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$
- This gives us the recurrence relation – a step closer towards the dynamic programing
- Lets see what this gives for multiplication of 4 matrices

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

$$\underset{d_0 \quad d_1}{A_1} \times \underset{d_1 \quad d_2}{A_2} \times \underset{d_2 \quad d_3}{A_3} \times \underset{d_3 \quad d_4}{A_4}$$

- 1. $A_1\ (A_2\ (A_3 A_4))$
- 2. $A_1\ ((A_2 A_3)\ A_4)$
- 3. $(A_1 A_2)(A_3 A_4)$
- 4. $(A_1\ (A_2 A_3))A_4$
- 5. $((A_1 A_2)A_3)A_4$

- Out of the 5 possible ways to multiply these matrices, we will find the optimum one by using the recurrence relation above

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min_{i \le k < j}\{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

$$\underset{d_0 \quad d_1}{A_1} \times \underset{d_1 \quad d_2}{A_2} \times \underset{d_2 \quad d_3}{A_3} \times \underset{d_3 \quad d_4}{A_4}$$

- We are after $c[1,4]$

- $c[1,4] = \min_{1 \le k < 4} \begin{array}{l} k=1 \\ k=2 \\ k=3 \end{array} \begin{cases} c[1,1] + c[2,4] + d_0 \times d_1 \times d_4 & A_1(A_2A_3A_4) \\ c[1,2] + c[3,4] + d_0 \times d_2 \times d_4 & (A_1A_2)(A_3A_4) \\ c[1,3] + c[4,4] + d_0 \times d_3 \times d_4 & (A_1A_2A_3)A_4 \end{cases}$

- Where are the other 2 orders/paranthesizations of multiplication?
- Its inside the subproblems of $A_2A_3A_4$ and $A_1A_2A_3$
- Of the different $c[i,j]$ values, we already know the two base cases - $c[1,1]$ and $c[4,4]$
- We need to expand the others

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min\limits_{i\le k<j}\{c[i,k] + c[k+1,j] + d_{i-1}\times d_k \times d_j\}$

$$\begin{array}{ccccccc} A_1 & \times & A_2 & \times & A_3 & \times & A_4 \\ d_0 \quad d_1 & & d_1 \quad d_2 & & d_2 \quad d_3 & & d_3 \quad d_4 \end{array}$$

- $c[1,4] = \min\limits_{1\le k<4} \begin{array}{l} k=1 \\ k=2 \\ k=3 \end{array} \begin{cases} c[1,1] + c[2,4] + d_0\times d_1\times d_4 & \qquad A_1(A_2A_3A_4) \\ c[1,2] + \boxed{c[3,4]} + d_0\times d_2\times d_4 & \qquad (A_1A_2)(A_3A_4) \\ c[1,3] + c[4,4] + d_0\times d_3\times d_4 & \qquad (A_1A_2A_3)A_4 \end{cases}$
- $c[2,4] = \min\limits_{2\le k<4} \begin{array}{l} k=2 \\ k=3 \end{array} \begin{cases} c[2,2] + \boxed{c[3,4]} + d_1\times d_2\times d_4 \\ \boxed{c[2,3]} + c[4,4] + d_1\times d_3\times d_4 \end{cases}$
- $c[1,3] = \min\limits_{1\le k<3} \begin{array}{l} k=1 \\ k=2 \end{array} \begin{cases} c[1,1] + \boxed{c[2,3]} + d_0\times d_1\times d_3 \\ c[1,2] + c[3,3] + d_0\times d_2\times d_3 \end{cases}$
- Thus we are going to solve smaller problems again and again
- Not only that, subproblems recur also
- Why don’t we start with the smaller problems, cache and reuse it!

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

$$\begin{array}{cccccccc} \multicolumn{2}{c}{A_1} & \times & A_2 & \times & A_3 & \times & A_4 \\ 3 & 2 & 2 & 4 & 4 & 2 & 2 & 5 \end{array}$$

- Lets create two tables c-table and k-table

$i \downarrow \quad \rightarrow j$

c	1	2	3	4
1				
2				
3				
4				

$i \downarrow \quad \rightarrow j$

k	1	2	3	4
1				
2				
3				
4				

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

$$\begin{array}{cccccccc} \multicolumn{2}{c}{A_1} & \times & A_2 & \times & A_3 & \times & A_4 \\ 3 & 2 & 2 & 4 & 4 & 2 & 2 & 5 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 & d_3 & d_4 \end{array}$$

$i \downarrow$, $j \rightarrow$

c	1	2	3	4
1	0			
2		0		
3			0	
4				0

$i \downarrow$, $j \rightarrow$

k	1	2	3	4
1				
2				
3				
4				

- We already know the diagonal values to be 0

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min\limits_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

$$A_1 \times A_2 \times A_3 \times A_4$$

$$\begin{array}{cccccccc} 3 & 2 & 2 & 4 & 4 & 2 & 2 & 5 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 & d_3 & d_4 \end{array}$$

$i \downarrow$ $\xrightarrow{\quad} j$

c	1	2	3	4
1	0			
2		0		
3			0	
4				0

$i \downarrow$ $\xrightarrow{\quad} j$

k	1	2	3	4
1				
2				
3				
4				

- $c[1,2] = \min\limits_{1 \le k < 2} k = 1 \{c[1,1] + c[2,2] + d_0 \times d_1 \times d_2 = 0 + 0 + 24 = 24$

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

$$\begin{array}{cccccccccccc} & A_1 & & \times & & A_2 & & \times & & A_3 & & \times & & A_4 & \\ 3 & & 2 & & 2 & & 4 & & 4 & & 2 & & 2 & & 5 \\ d_0 & & d_1 & & d_1 & & d_2 & & d_2 & & d_3 & & d_3 & & d_4 \end{array}$$

$i \downarrow$, $j \rightarrow$

c	1	2	3	4
1	0	24		
2		0		
3			0	
4				0

$i \downarrow$, $j \rightarrow$

k	1	2	3	4
1		1		
2				
3				
4				

- $c[1,2] = \min_{1 \le k < 2} k = 1 \{c[1,1] + c[2,2] + d_0 \times d_1 \times d_2 = 0 + 0 + 24 = 24$

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min\limits_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

$$\begin{array}{cccccccc} \multicolumn{2}{c}{A_1} & \times & \multicolumn{2}{c}{A_2} & \times & \multicolumn{2}{c}{A_3} \; \times \; A_4 \\ 3 & 2 & 2 & 4 & 4 & 2 & 2 & 5 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 & d_3 & d_4 \end{array}$$

c (i ↓, j →)	1	2	3	4
1	0	24		
2		0	16	
3			0	40
4				0

k (i ↓, j →)	1	2	3	4
1		1		
2			2	
3				3
4				

- $c[2,3] = \min\limits_{2 \le k < 3} k = 2 \left\{c[2,2] + c[3,3] + d_1 \times d_2 \times d_3 = 0 + 0 + 16 = 16\right.$
- $c[3,4] = \min\limits_{3 \le k < 4} k = 3 \left\{c[3,3] + c[4,4] + d_2 \times d_3 \times d_4 = 0 + 0 + 40 = 40\right.$

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min\limits_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

$$\begin{array}{cccccccc} \multicolumn{2}{c}{A_1} & \times & A_2 & \times & A_3 & \times & A_4 \\ 3 & 2 & 2 & 4 & 4 & 2 & 2 & 5 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 & d_3 & d_4 \end{array}$$

j →, i ↓

c	1	2	3	4
1	0	24		
2		0	16	
3			0	40
4				0

j →, i ↓

k	1	2	3	4
1		1		
2			2	
3				3
4				

- $$c[1,3] = \min_{1 \le k < 3} \begin{cases} k = 1 & c[1,1] + c[2,3] + d_0 \times d_1 \times d_3 = 0 + 16 + 12 = 28 \\ k = 2 & c[1,2] + c[3,3] + d_0 \times d_2 \times d_3 = 24 + 0 + 24 = 48 \end{cases}$$

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min\limits_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

$$
\begin{array}{cccccccccccc}
 & A_1 & & \times & & A_2 & & \times & & A_3 & \times & A_4 \\
3 & & 2 & & 2 & & 4 & & 4 \quad 2 & & 2 & 5 \\
d_0 & & d_1 & & d_1 & & d_2 & & d_2 \quad d_3 & & d_3 & d_4
\end{array}
$$

i ↓ j →

c	1	2	3	4
1	0	24	28	
2		0	16	
3			0	40
4				0

i ↓ j →

k	1	2	3	4
1		1	1	
2			2	
3				3
4				

- $c[1,3] = \min\limits_{1 \le k < 3} \begin{cases} k = 1 & c[1,1] + c[2,3] + d_0 \times d_1 \times d_3 = 0 + 16 + 12 = 28 \\ k = 2 & c[1,2] + c[3,3] + d_0 \times d_2 \times d_3 = 24 + 0 + 24 = 48 \end{cases}$

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

$$\begin{array}{cccccccccccc} & A_1 & & \times & & A_2 & & \times & & A_3 & & \times & & A_4 \\ 3 & & 2 & & 2 & & 4 & & 4 & & 2 & & 2 & & 5 \\ d_0 & & d_1 & & d_1 & & d_2 & & d_2 & & d_3 & & d_3 & & d_4 \end{array}$$

c ($i \downarrow$, $j \rightarrow$)	1	2	3	4
1	0	24	28	
2		0	16	36
3			0	40
4				0

k ($i \downarrow$, $j \rightarrow$)	1	2	3	4
1		1	1	
2			2	3
3				3
4				

- $c[2,4] = \min_{2 \le k < 4} \begin{cases} k = 2 & c[2,2] + c[3,4] + d_1 \times d_2 \times d_4 = 0 + 40 + 40 = 80 \\ k = 3 & c[2,3] + c[4,4] + d_1 \times d_3 \times d_4 = 16 + 0 + 20 = 36 \end{cases}$

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

$$\begin{array}{cccccccccccc} & A_1 & & \times & & A_2 & & \times & & A_3 & \times & A_4 \\ 3 & & 2 & & 2 & & 4 & & 4 & 2 & 2 & 5 \\ d_0 & & d_1 & & d_1 & & d_2 & & d_2 & d_3 & d_3 & d_4 \end{array}$$

i ↓ c j →	1	2	3	4
1	0	24	28	58
2		0	16	36
3			0	40
4				0

i ↓ k j →	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

- $$c[1,4] = \min_{1 \le k < 4} \begin{cases} k = 1 & c[1,1] + c[2,4] + d_0 \times d_1 \times d_4 = 0 + 36 + 30 = 66 \\ k = 2 & c[1,2] + c[3,4] + d_0 \times d_2 \times d_4 = 24 + 40 + 60 = 124 \\ k = 3 & c[1,3] + c[4,4] + d_0 \times d_3 \times d_4 = 28 + 0 + 30 = 58 \end{cases}$$

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min\limits_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

$$A_1 \times A_2 \times A_3 \times A_4$$

$$\begin{array}{cccccccc} 3 & 2 & 2 & 4 & 4 & 2 & 2 & 5 \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 & d_3 & d_4 \end{array}$$

c	1	2	3	4
1	0	24	28	58
2		0	16	36
3			0	40
4				0

k	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

- So, the optimum number of scalar multiplications to get $A_1A_2A_3A_4$ is 58
- The optimum order can be got from the k-table

Matrix Chain Multiplication with 4 Matrices

- $c[i,j] = \min_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

$$\begin{array}{cccccccccccc} & A_1 & & \times & & A_2 & & \times & & A_3 & & \times & & A_4 \\ 3 & & 2 & & 2 & & 4 & & 4 & & 2 & & 2 & & 5 \\ d_0 & & d_1 & & d_1 & & d_2 & & d_2 & & d_3 & & d_3 & & d_4 \end{array}$$

$$\begin{array}{c|c|c|c|c} i \downarrow \; c \; \xrightarrow{} j & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 24 & 28 & 58 \\ 2 & & 0 & 16 & 36 \\ 3 & & & 0 & 40 \\ 4 & & & & 0 \end{array}
\qquad
\begin{array}{c|c|c|c|c} i \downarrow \; k \; \xrightarrow{} j & 1 & 2 & 3 & 4 \\ \hline 1 & & 1 & \boxed{1} & \boxed{3} \\ 2 & & & 2 & 3 \\ 3 & & & & 3 \\ 4 & & & & \end{array}$$

- $\boxed{\boxed{A_1} A_2 A_3} A_4$
- $\left(A_1 (A_2 A_3)\right) A_4$

Matrix Chain Multiplication Pseudocode

- $c[i,j] = \min_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$

```
for i=1 to n
  c[i,i] = 0
for s=1 to n-1
  for i=1 to n-s
    j=i+s
    $c[i,j] = \min_{i \le k < j} \{c[i,k] + c[k+1,j] + d_{i-1} \times d_k \times d_j\}$
```

- Runtime: Half the cells in the c-matrix is filled up -> $O(n^2)$
- Time per cell – Need to check $i \le k < j$. Each takes constant time. $O(n)$
- So, final runtime is $O(n^3)$

Thank you