Intro to Dynamic Programming
CIS, Fordham Univ.

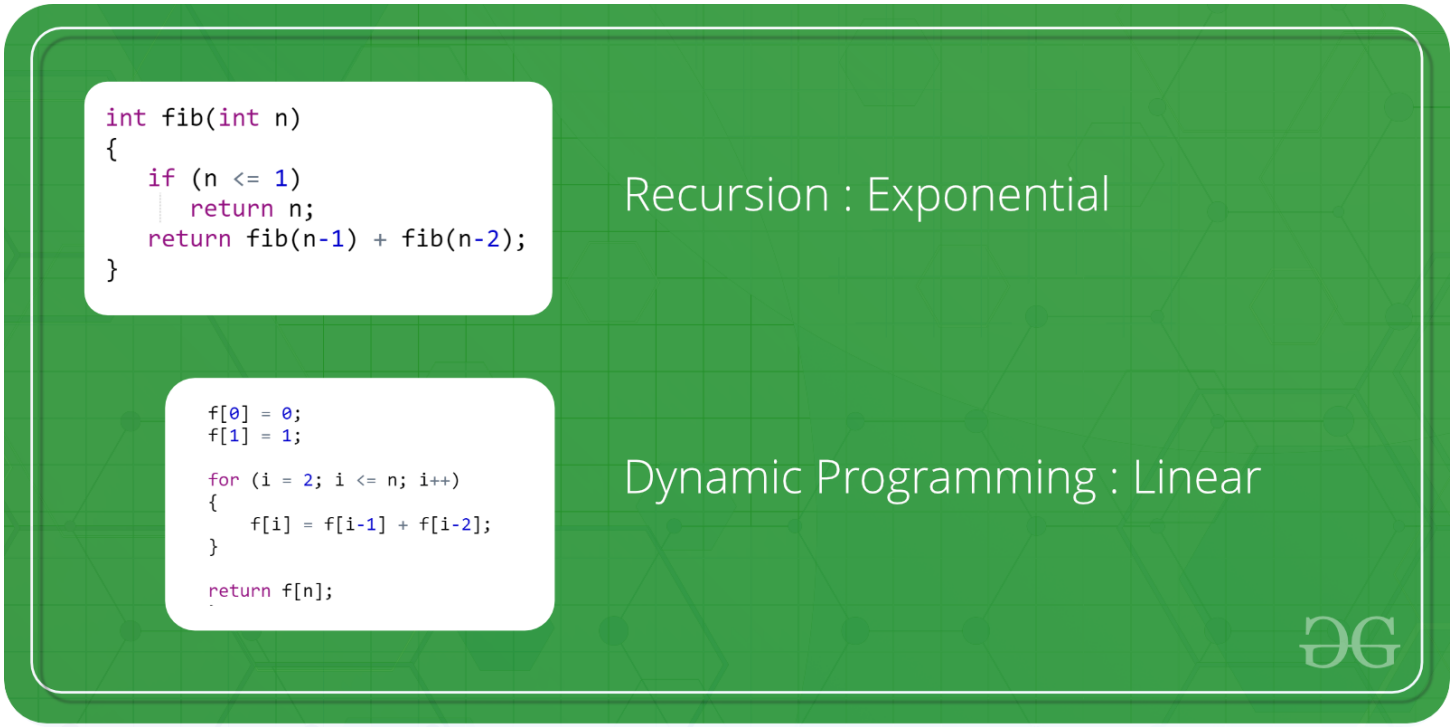
Instructor: X. Zhang

Outline

- Introduction via example: Fibonacci, rod cutting
  - Characteristics of problems that can be solved using dynamic programming
  - Unlimited Knapsack problem
- Two dimensional problem spaces
  - Longest common subsequence
  - Matrix chain multiplication
- Summary

Dynamic Programming

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial. For example, if we write simple recur-



Dynamic Programming: ideas

- **Optimal substructure**: (Optimal) solution to a problem of size n incorporates (optimal) solutions to problems of smaller size (n-1, n-2).
- **Recursive calling tree shows overlapping of subproblems**
  - i.e., same subproblems are called multiple times
- Idea: avoid recomputing subproblems
  - store subproblem solutions in memory/table (hence “programming”)
  - Two approaches:
    - Memoization: recursive with a table
    - Tabulation: non-recursive with a table (tabulation)

Rod Cutting Problem

- A company buys long steel rods (of length n), cuts them into shorter ones to sell
  - integral length only
  - Selling price for rods of different length:

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

- Goal: find maximum (possible) total revenue from selling these rods (and how to achieve it).

Rod Cutting Problem

- Input: length of given rod, n; and p[i], selling price of rod of length i, e.g.,

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

- Output: maximal profit over all possible ways to cut n to shorter pieces and sell
- e.g., for n=4, we could cut it in five ways:

multiset: allow duplicate, order does not matter

- {4}: do not cut ==> profit: $9
- {3,1} ==> $8+$1=$9
- {2,2} ==> $10 <<== this is the maximal profit!
- {2,1,1} ==> $7
- {1,1,1,1}==> $4

Solution Space

- One way: first enumerate all possible ways to cut , then evaluate each possible ways to pick one with highest total selling price
- How to enumerate all possible ways to cut n?
  - A combinatorial problem…

Optimize recursively

- Another way: optimize recursively, find optimal solution to sub-problem directly, and use them to solve original problem
  - A recursive algorithm that return optimal solution

```
/*return maximum profit achievable with a rod of length n
by checking all possible ways of cutting n into rods of length i=
1, 2, ..., k and selling them for p[i] */
Int CutRod (n, p[1...k])
/*
 Base case: smallest problem(s) that we can solve trivially

 General case:
   How to reduce problem to smaller problems? */
```

Optimize recursively

/*return maximum profit achievable with a rod of length n

by checking all possible ways of cutting n into rods of length i= 1, 2, …, k and selling them for p[i] */

Int CutRod (n, p[1…k])

/*

How to reduce the problem to smaller problems, general case?

Hint: use decision tree.

*/

Optimal Substructure => recursive solution

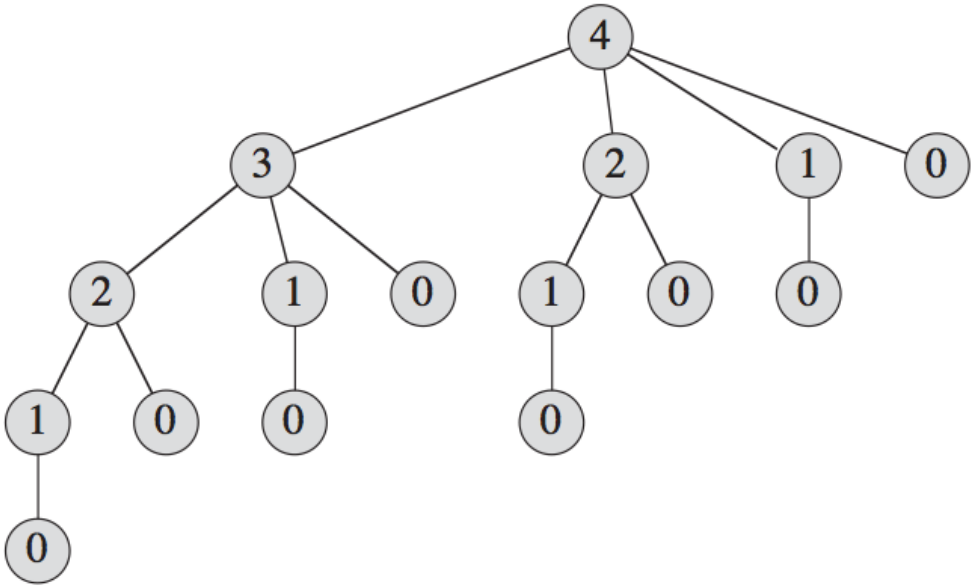
```
//return max. profit one can make with a rod of length n
CutRod (n, p[1…k])
 {  //What's the smallest problem(s) that we can solve trivially?
     if n==0   return 0
     if n==1      return p[1]

    //general case
    curMax=0
    for c1=1, 2, 3, … ,  min (n, k) {
         //c1: consider the first rod to cut out and sell:

        curProfit = p[c1] + CutRod (n-c1,p)
        If (curProfit > curMax)
             curMax = curProfit
      }
   return curMax
   }
```

What if we remove n=1 from base case?

Optimal substructure

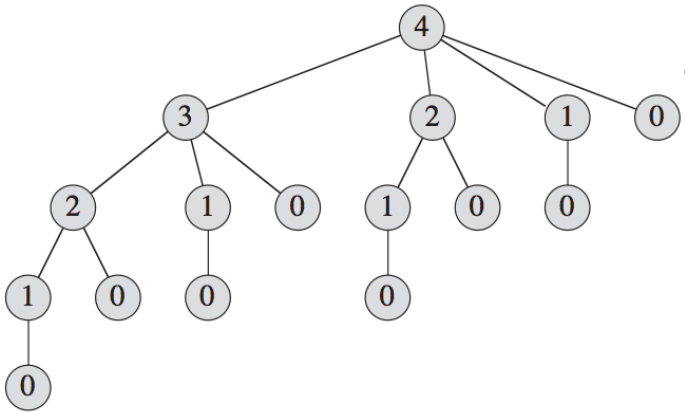
- **Optimal substructure**: Optimal solution to a problem of size n incorporates optimal solutions to problems of smaller size (n-1, n-2,…).

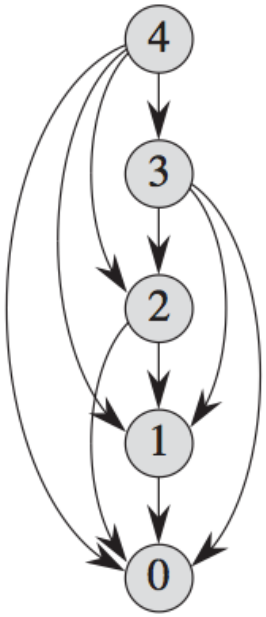
Recursive Tree for CutRod(n=4, p)



How many times CutRod (2) is called?
How about CutRod(1)?

Overlapping of Subproblems





- Recursive calling tree shows overlapping of subproblems
  - i.e., n=4 and n=3 share overlapping subproblems (2,1,0)
- Idea: avoid recomputing subproblems again and again
  - store subproblem solutions in memory/table (hence "programming")

DP with Memoization

- Improve recursive solution by storing subproblem solution in a table
- when need solution of a subproblem, check if it has been solved before,
  - if not, calculate it and store result in table
  - if yes, access result stored in table

Recursion=> Memoization

```
//return max. profit one can make with a rod of length n
CutRod (n, p[1…k])
 {  //What's the smallest problem(s) that we can solve trivially?
     if n==0   return 0
     if n==1      return p[1]

    //general case
    curMax=0
    for c1=1, 2, 3, … ,  min (n, k) {
         //c1: consider the first rod to cut out and sell:

        curProfit = p[c1] + CutRod (n-c1,p)
        If (curProfit > curMax)
             curMax = curProfit
      }
   return curMax
   }
```

What kind of table?

Where to allocate the table?

Where to update table entry?

When to look up the table entry?

Memoization illustrated in code

```
//return max. profit one can make with a rod of length n
CutRod (n, p[1…k])
1.      create an array r[1…n], filled with -1 (indicate “not calculated yet”)
2.      CutRodHelper (n, p, r)

CutRodHelper (n, p[1…k], r[])
1.    if r[n] >=0 return r[n]  //if it has been calculated already
2.     // no need to recalculate, return the stored result
3.
4.    if n==0 return 0 //base case
5.    //general case
6.    curMax=0
7.    for  c1=1, 2, 3, … ,  min (n, k)
8.          curProfit = p[c1] + CutRodHelper (n-c1)
9.          curMax = max(curProfit, curMax)
10.
11.   r[n]= curMax   //save result in r[] for future reference
12.   return curMax
```

DP:Tabulation

- Tabulation
  - Iteratively solve smaller problems first, move the way up to larger problems
  - bottom-up method: as we solve smaller problems first, and then larger and larger one
    - => when solving a problem, all subproblems solutions that are needed have already been calculated

Bottom-up

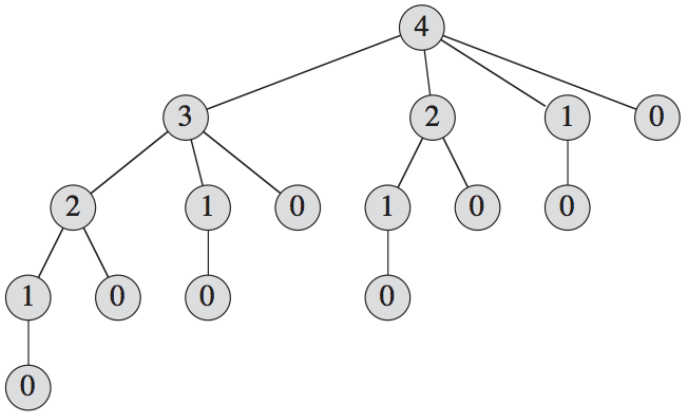
```
//return max. profit one can make with a rod of length n
1. CutRodBottomUp (n, p[1…k])
2.     create an array r[1…n] //store subproblem solutions
3.     r[0] = 0
4.     for i=1 to n  // solve smaller problems first …
5.          // calculate r_i , max revenue for rod length i
6.         curMax=0
7.         for  c1=1, 2, 3, … ,  min (i, k)
8.              curProfit = p[c1]+ r[i-c1]
9.              curMax = max(curProfit, curMax)
10.         r[i]= curMax   //save result in r[] for future reference
11.
12. return r[n]
```

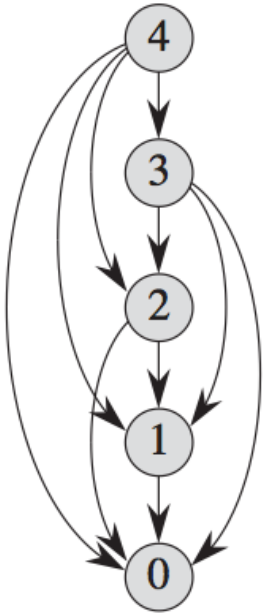
Recap

- We analyze rod cutting problem
- Two characteristics of problems that can benefit from dynamic programming:
  - optimal substructure: a recursive formular

$$r_n = \max_{c_1=1,2...min\{n,k\}} \{p[c_1] + r_{n-c_1}\}$$

  - overlapping subproblems





Recap (2)

- How dynamic programming works:
  - Memoization: recursion with table
  - Tabulation: iteratively solve all possible subproblems, and work our way from small problems to large problems

Tracing: CutRod(n=3,p)

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

```
//return max. profit one can make with a rod of length n
CutRodTopLevel (n, p[1…k])
1.      create an array r[1…n], filled with -1 (indicate "not calculated yet")
2.      CutRodHelper (n, p, r)

CutRod (n, p[1…k], r[])
1.     if r[n] >=0 return r[n]  //if it has been calcula
2.      // no need to recalculate, return the stored
3.
4.     if n==0
5.          r[0] = 0
6.           return 0 //base case
7.     //general case
8.     curMax=0
9.     for  c1=1, 2, 3, … ,  min (n, k)
10.             curProfit = p[c1] + CutRod (n-c1, p, r)
11.             curMax = max(curProfit, curMax)
12.
13.    r[n]= curMax   //save result in r[] for future reference
14.    return curMax
```

Tracing with n=3

table r :

c1
curProfit
curMax

Tabulation: Tracing n=5

```
//return max. profit one can make with a rod of length n
CutRodBottomUp (n, p[1…k])
{
    create an array r[1…n] //store subproblem solutions
    r[0] = 0
    for i=1 to n  // solve smaller problems first …
        // calculate r_i , max revenue for rod length i
        curMax=0
        for  c1=1, 2, 3, … ,  min (i, k)
            curProfit = p[c1]+ r[i-c1]
            curMax = max(curProfit, curMax)
        r[i]= curMax   //save result in r[] for future reference

 return r[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

Cutting that achieves max profit?

```
//return max. profit one can make with a rod of length n
CutRodBottomUp (n, p[1…k])
{
    create an array r[1…n] //store subproblem solutions
    r[0] = 0
    for i=1 to n  // solve smaller problems first …
        // calculate r_i , max revenue for rod length i
        curMax=0
        for  c1=1, 2, 3, … ,  min (i, k)
            curProfit = p[c1]+ r[i-c1]
            curMax = max(curProfit, curMax)
        r[i]= curMax   //save result in r[] for future reference

 return r[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

Outline

- Introduction via example: Fibonacci, rod cutting
  - Characteristics of problems that can be solved using dynamic programming
  - Knapsack: with repetition
- Two dimensional problem spaces
  - Longest common subsequence
  - Matrix chain multiplication
- Summary

Knapsack Problem

- Given:
  - A backpack with weight capacity of W
  - n different types of objects, i-th type of objects weighs w[i] and has a value of v[i]
  - there are infinite quantities of each object type
- What to put into backpack so that total value is maximized and total weights <= W

e.g., W=13, w[4]={5,3,8,4}, v[4]={10,20,25,8}

Knapsack Problem

Input: W=13, w[4]={5,3,8,4}, v[4]={10,20,25,8}

Output: maximum value achievable, assuming there is infinity amount of each object.

For what value of W you know the answer directly?

For a larger W, how can you reduce it to smaller problems?

Optimal substructure in Knapsack

- Input: weight capacity of a knapsack, W; n different objects (of infinite quantities) with weights and values given by array w[ ], v[ ]
- Output: objects so that **total value is maximized and total weights <= W**

- **Let V_k be max total value possible when weight capacity is k**
- **Recursive formula for V_k**

input: W=13, w[4]={5,3,8,4}, v[4]={10,20,25,8}

Output: maximum value achievable, assuming there is infinity amount of each object.

Knapsack: extension

- Input: weight capacity of a knapsack, W; n different objects (of infinite quantities) with weights and values given by array w[ ], v[ ]
- Output: objects so that total value is maximized and total weights <= W
- Let V_k be max total value possible when weight capacity is k

- What's the set of objects (multi-set) that achieve V_k?
  - The first obj i chosen that achieves the max value above
  - and then object i_1 chosen for weight capacity k-w[i]
  - and then object i_2 chosen for weight capacity k-w[i]-w[i_1]
  - … until the weight capacity ==0 or < min(w)
- Use a table obj[]
  - obj[k] store the first object to chose when capacity is k (the one that maximize …)

Optimal substructure

- **Optimal substructure**: Optimal solution to a problem of size n incorporates optimal solution to problem of smaller size (1, 2, 3, … n-1).
- Rod cutting: find r_n (max. revenue for rod of len n)

Sol to problem instance of size n

Sol to problem instance of size n-1, n-2, … 1

$$r_n = \max(p[1]+r_{n-1}, p[2]+r_{n-2}, p[3]+r_{n-3}, \ldots, p[n-1]+r_1, p[n])$$

  - A recurrence relation (recursive formula)

- => Dynamic Programming: Build an optimal solution to the problem from solutions to subproblems
  - We solve a range of sub-problems as needed

Outline

- Introduction via example: rod cutting
- Characteristics of problems that can be solved using dynamic programming
- Two dimensional problem spaces
  - 0/1 Knapsack (i.e., without repetition)
  - Minimum Edit Distance
  - Matrix chain multiplication
- Summary

Knapsack without repetition

- Given
  - a weight capacity of a knapsack, W
  - n different objects (one of each): with weights and values given by arrays w[ ], v[ ]
  - **finding a subset of objects …**
- Goal: choose a subset of objects so that total value is maximized and total weights <= W
- Plan:
  - Recall you solved this problem in Lab4
  - Pure recursive solution
  - use memoization or tabulation to improve

Recursion Solution
Let's fill in the blank

```
/*  Output max. value achievable
@param W: given weight capacity, >=0
@ param n: we can choose from first n obj
@param w, v: weights and values
@return max value achievable from the first n obj under W
*/
Knapsack_Norepeat (W, w, v, n)
{
   if W==0 or n == 0      //base case
      [                              ] //fill in the blank

   /* general case */
   if (w[n-1] > W) // the last obj is too heavy …


  else  { //The last obj can fit
     //option 1: if we include n-1-th obj, what's the max value achievable …


    //option 2: if we don't include (n-1)-th obj at all, what's the max value achievable?
    // Which option is better?
}
```

DP/Memoization vs Pure Recursion

	Memoization	*Pure Recursion*
How does it work?		
Solve subproblems only if needed		
Recursion overhead?		
When to choose it?		

Memoization vs Tabulation

	Memoization	*Tabulation*
How does it work?		
Solve subproblems only if needed		
Recursion overhead?		
When to choose it?		

Outline

- Introduction via example: rod cutting
- Characteristics of problems that can be solved using dynamic programming
- More one dimensional examples
  - Knapsack with repetition
- Two dimensional problem spaces
  - Knapsack without repetition
  - Longest common subsequence
  - Matrix chain multiplication (skipped)
- Summary

Longest Common Subsequence

- Given a sequence, $X=\langle x_1, x_2, \ldots, x_m \rangle$, where each x_i is a letter from a certain alphabet, a subsequence of X is a sequence of elements taken in order from X but not necessarily consecutive
- Example:
  - X = <A, B, C, B, D, A, B>
  - <A, B, D>, <B, C, D, B>, <A>, <>, <A,B,C,B,D,A,B> are subsequence of X
  - <A, C, C>, <B, B, C> are not subsequence of X

- How many possible subsequences are there for X?

- Denote length of a sequence X by |X|, which is the number of letters in sequence
  - e.g., X = <A, B, C, B, D, A, B>, |X|=7

Longest Common Subseq.

- Given two sequences

  $X=\langle x_1, x_2, \ldots, x_m \rangle$, $Y=\langle y_1, y_2, \ldots y_n \rangle$
- Find ***a* longest common** subsequence (in short, LCS) of X and Y, i.e., a sequence that
  - is a subsequence of X, and is a subsequence of Y
  - and is no shorter than any other common subsequences of X and Y

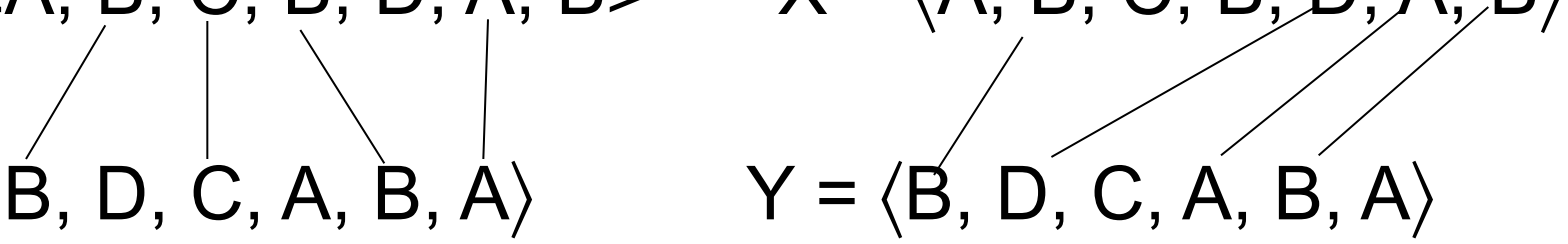
LCS examples

X = <A, B, C, B, D, A, B>

Y = <B, D, C, A, B, A⟩

X = ⟨A, B, C, B, D, A, B⟩

Y = ⟨B, D, C, A, B, A⟩



- ⟨B, C, B, A⟩ and ⟨B, D, A, B⟩ are both longest common subsequences of X and Y (length = 4)

LCS examples

2. X = <A, A, C, A, G, T, T, A, C, C>,

   Y = <T, A, A, G, G, T, C, A>

   What's the LCS of these two sequences?

Brute-Force Solution

```
1. /* Check every subsequence of X[1 . . m] to see if it is also a subsequence of Y[1 .. n]. */
2. LCS (X, Y)
3. {
4.     for each of 2^m subsequence, s, of X
5.         //check if s is a subsequence of Y, O(n) time
6.          k = |s|  //k is length of s
7.          j = 1  //used to scan through Y
8.
9.         for i=1 to k. //for each letter in s, finds a matching char in Y
10.                 while ( Y[j]!=s[i] and j<=n)
11.                     j++                 // scan Y for a letter matching s[i]
12.                 if j>n   // cannot match s[i]
13.                         break and s not a subsequence Y
14.
15.         s is subsequence of Y, update longest
16.
17.}
```


Worst-case running time: $O(n2^m)$

Sequence Prefix

- Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$,
- Def: ***i*-th prefix of *X*,** $X_i = \langle x_1, x_2, \ldots, x_i \rangle$

- Practice: $X = \langle A, B, C, B, D, A, B \rangle$,
  - what's X_2, X_4, X_0, X_7?

Thinking about subproblems

How to calculate LCS?

X= A B C D

Y= E A C F D

- Assuming subproblems have been solved, i.e., LCS of prefix of X and Y have been found…

Thinking about subproblems

How to calculate LCS?

X= A B C A D

Y= E A C D E

- Assuming subproblems have been solved, i.e., LCS of prefix of X and Y have been found…

Recursive Solution. Case 1

```
/* Return a longest subsequence of X, Y
@param X: is a sequence
@param Y: is a sequence
@return the longest common subsequence of X and Y*/
LCS (X, Y)
{
   m = |X|, n = |Y|
   if X[m] ==Y[n]
```

/* X = ⟨A, B, D, E⟩

Y = ⟨Z, B, E⟩ */

- Todo: find a LCS of X_{m-1} and Y_{n-1}, (here, X3 =<A, B, D>, and Y2 = <Z, B>)
- append X[m] to the end to get LCS of X_m, Y_n

Recursive Solution. Case 2

if X[m] != Y[n]

e.g., X = ⟨A, B, D, G⟩

Y = ⟨Z, B, D⟩

This G and D cannot be both in LCS

$$LCS(X, Y) = \text{Longer}\ \{\ \underline{LCS(X_{m-1}, Y)},\ LCS(X, Y_{n-1})\}$$

If we ignore last element in X

If we ignore last element in Y

- Must solve two subproblems

```
/* Return a longest subsequence of X, Y
@param X: is a string
@param Y: is a string
@return the longest common subsequence of X and Y*/
LCS (X, Y)
{
    m=|X|, n=|Y|
    if (|X| == 0 or |Y|==0) return " "; //empty string
    //general case
    // can we match last letters of X and Y?
    if X[m] == Y[n]
          return LCS(X_{m-1},Y_{n-1})+X[m]  //concatenated with last letter
     else
          // X[m] and Y[n] cannot be both in LCS
          s1 = LCS(X_{m-1},Y_n) //  X[m] is not in LCS
          s2 = LCS(X_m, Y_{n-1}) //Y[n-1] is not in LCS
          return longer one among s1 and s2
}
```

Recursion

Three Questions?

$X = \langle A, B, D, E \rangle$
$Y = \langle Z, B, E \rangle$

$X = \langle A, B, D, G \rangle$
$Y = \langle Z, B, D \rangle$

Here we need to calculate prefix (Xm-1, Yn-1), and pass them to recursive calls

```
/* Return the longest subsequence of X, Y
@param X : is a string
@param Y: is a string
@return the longest common subsequence of Xm and Yn*/
LCS (X, Y, m, n )
{
    if (m== 0 or n==0) return " "; //empty string
    //general case
    // can we match last letters of X and Y?
    if X[m] == Y[n]
          return LCS(X,Y, m-1, n-1)+X[m]  //concatenated with last letter
     else
          // X[m] and Y[n] cannot be both in LCS
          s1 = LCS(X, Y, m-1, n )
          s2 = LCS(X, Y, m, n-1)
          return longer of s1 and s2
}
```

Recursion

Three Questions?

X = ⟨A, B, D, E⟩
Y = ⟨Z, B, E⟩

X = ⟨A, B, D, G⟩
Y = ⟨Z, B, D⟩

// keep X, Y as they are, use parameters to specify prefix length
// subproblem's size is given by m and n, at least one dimension is decreased

Optimal substructure & Overlapping Subproblems

- A recursive solution contains a “small” number of distinct subproblems repeated many times.
  - e.g., LCS (5,5) depends on LCS(4,4), LCS(4,5), LCS(5,4)
  - Exercise: Draw subproblem dependence graph
    - each node is a subproblem
    - directed edge represents “calling”, “uses solution of” relation
- Small number of distinct subproblems:
  - total number of distinct LCS subproblems for two strings of lengths *m* and *n* is *mn*.

Tabulation to avoid recalculation

- Given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$, $Y = \langle y_1, y_2, \ldots, y_n \rangle$
  - To ease tracing, we focus on finding **length** of LCS

$c[i, j] = |\ LCS\ (X_i, Y_j)\ |$
// length of LCS of i-th prefix of X and j-th prefix of Y

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise (i.e., if } X[i] \neq Y[j]) \\ 0, \text{ if } i=0 \text{ or } j=0 & \end{cases}$$

Tabulation

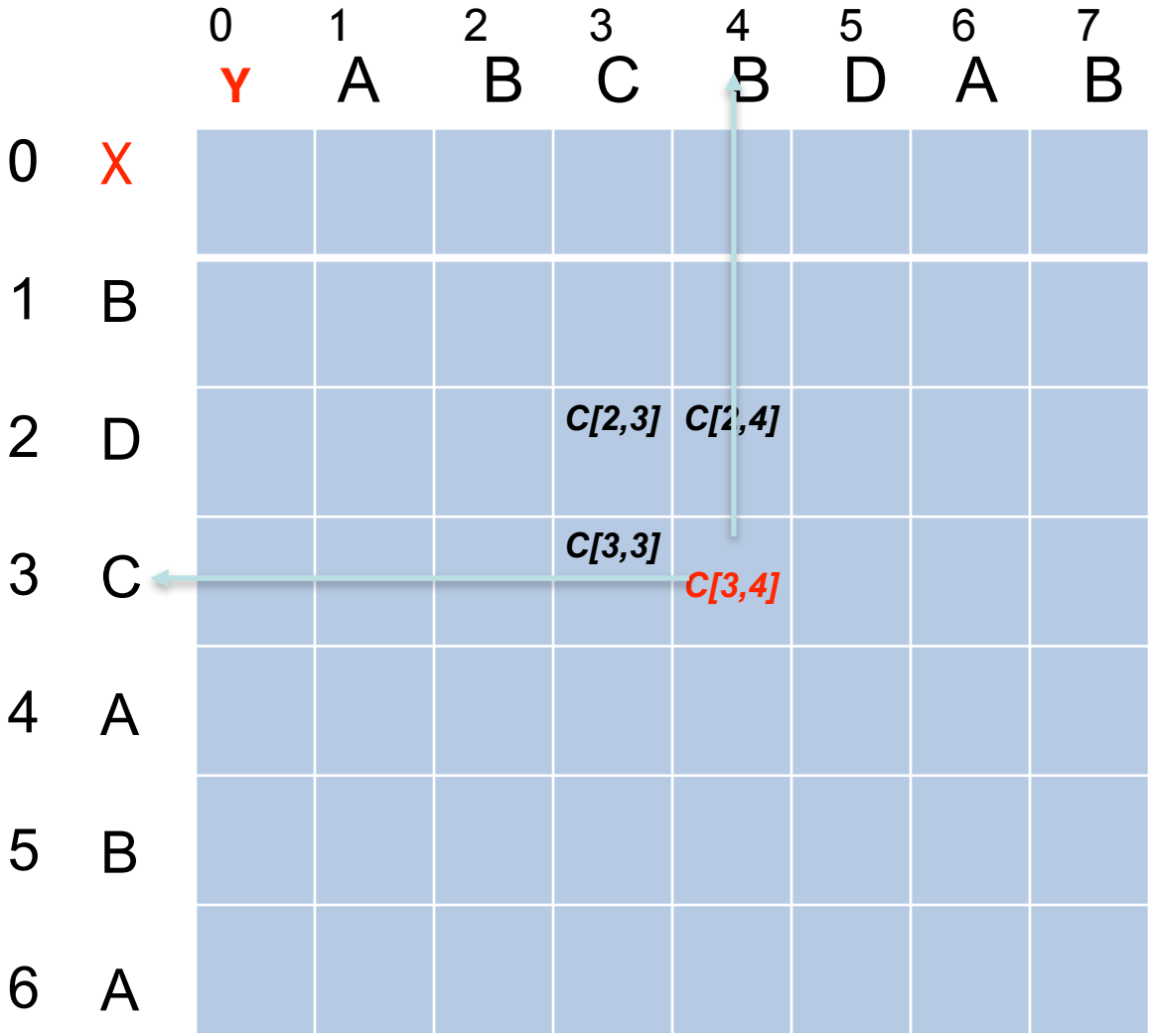
LCS (X, Y)
X=<B, D, C, A, B, A>
Y=<A, B, C, B, D, A, B>
|X|=6, |Y|=7

c is 7x8 array

C[3,4]= length of LCS (X_3, Y_4)
= Length of LCS (BDC, ABCB)

3rd row, 4th column element

Goal: calculate C[6,7] (bottom right corner)



Bottom-Up

Initialization: base case
c[i,j] = 0 if i=0, or j=0

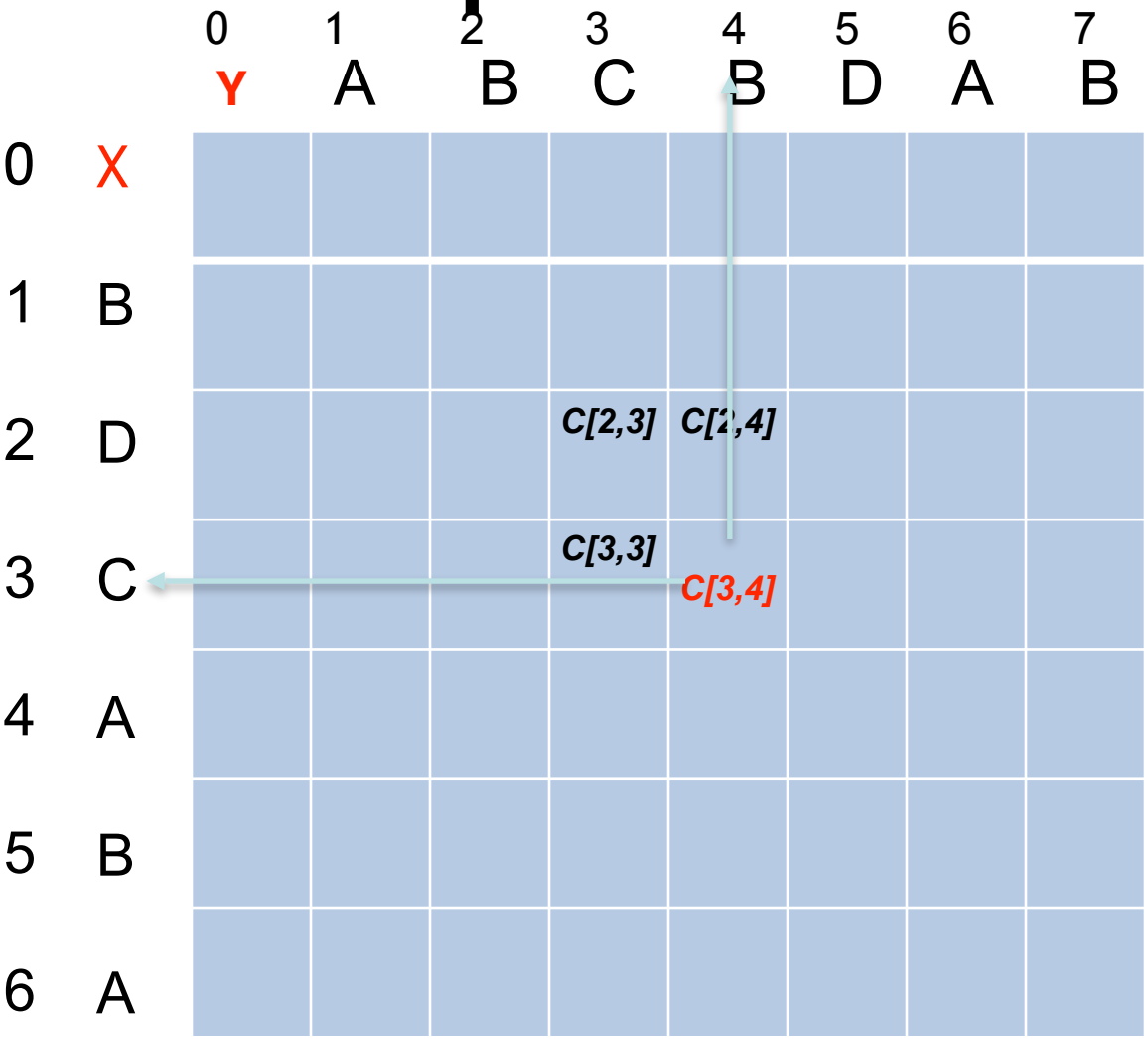
//Fill table row by row
// from left to right
for (int i=1; i<=m;i++)
    for (int j=1;j<=n;j++)
        Calculate c[i,j]

return c[m, n]

Running time = Θ(mn)

C[3,4]= length of LCS (X_3, Y_4)
= Length of LCS (BDC, ABCB)

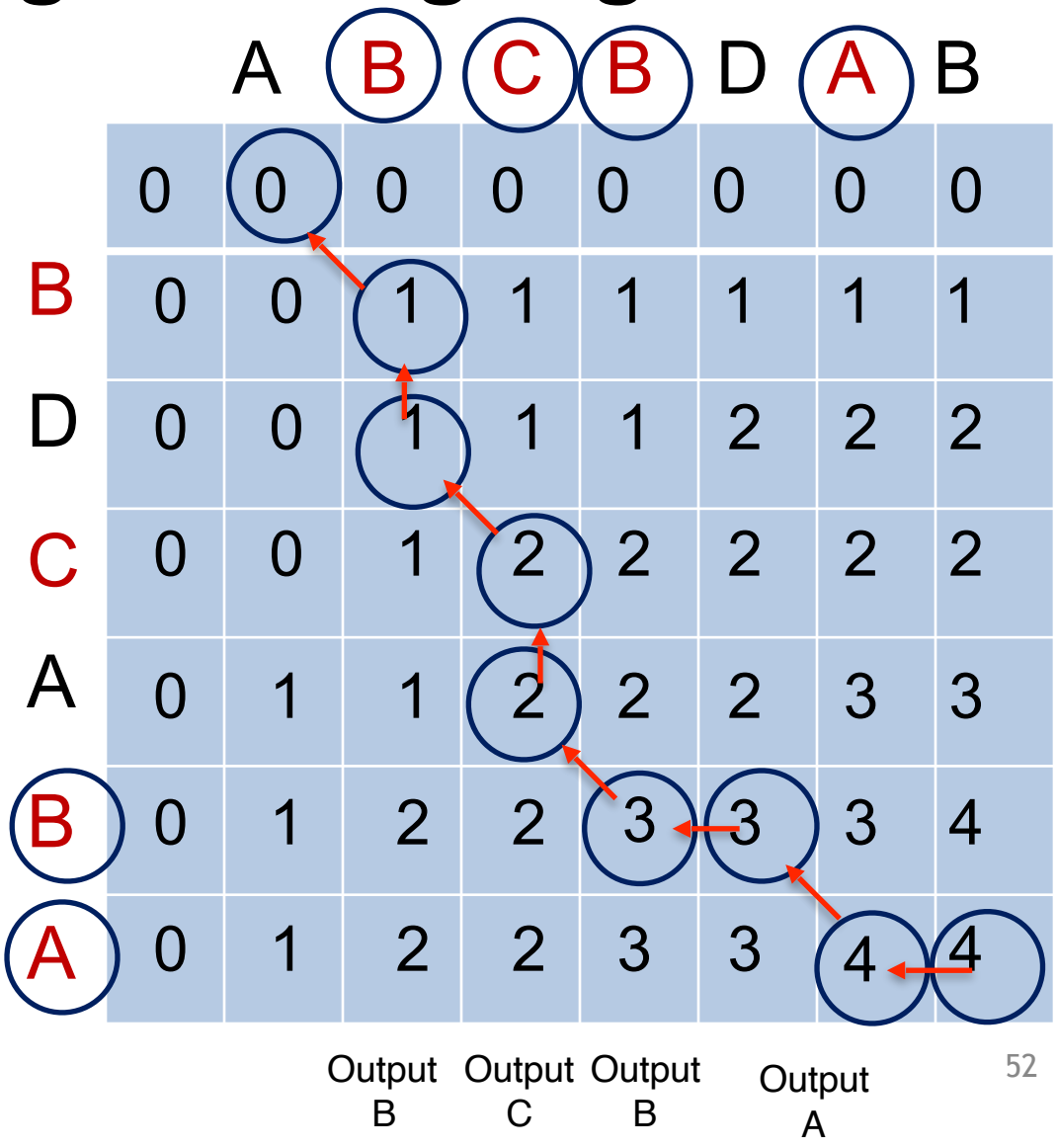
3rd row, 4-th column element



Dynamic-Programming Algorithm

Reconstruct LCS by tracing backward:

Where do we get value of C[i,j] from?



Remark

- Longest common subsequence algorithm is similar to
  - minimum edit distance problem (used by spell checker to suggest a correction)

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

    - If each operation has cost of 1
      - Distance between these is 5
    - If substitutions cost 2 (Levenshtein)
      - Distance between them is 8

  - Needleman-Wansh Alg. (used in bioinformatics to align protein or nucleotide sequences)

Matrix

Matrix: a 2D (rectangular) array of numbers, symbols, or expressions, arranged in rows and columns.

e.g., a 2 × 3 matrix (there are two rows and three columns)

$$\begin{bmatrix} 1 & 9 & -13 \\ 20 & 5 & -6 \end{bmatrix}.$$

Each element of a matrix is denoted by a variable with two subscripts, $a_{2,1}$ element at second row and first column of a matrix A.

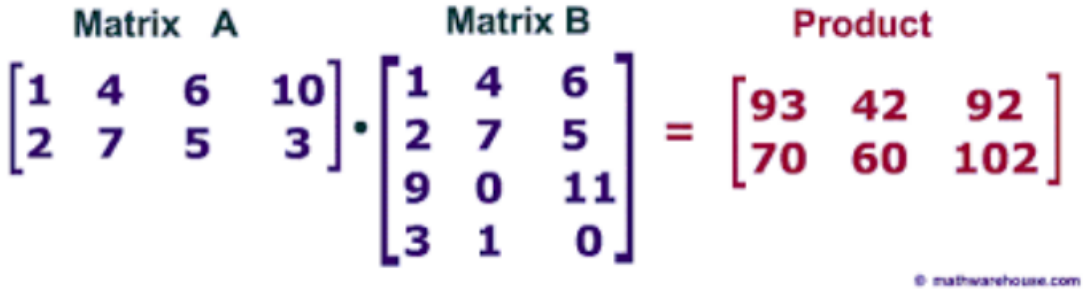
an m × n matrix A:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Matrix Multiplication

Matrix Multiplication:

Dimension of A, B, and A x B?



$$[\mathbf{AB}]_{i,j} = A_{i,1}B_{1,j} + A_{i,2}B_{2,j} + \cdots + A_{i,n}B_{n,j} = \sum_{r=1}^{n} A_{i,r}B_{r,j},$$

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2       error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4       for i = 1 to A.rows
5            for j = 1 to B.columns
6                 c_ij = 0
7                 for k = 1 to A.columns
8                      c_ij = c_ij + a_ik · b_kj
9       return C
```

Total (scalar) multiplication: 4x2x3=24

Total (scalar) multiplication: $n_2 x n_1 x n_3$

Multiplying a chain of Matrix Multiplication

Given a sequence/chain of matrices, e.g., A_1, A_2, A_3, there are different ways to calculate $A_1A_2A_3$

1. $(A_1A_2)A_3)$

2. $(A_1(A_2A_3))$

Dimension of A_1*: 10 x 100*

A_2*: 100 x 5*

A_3*: 5 x 50*

all yield the same result

But not same efficiency

Matrix Chain Multiplication

*G*iven a chain $<A_1, A_2, \ldots A_n>$ of matrices, where matrix A_i has dimension $p_{i-1} \times p_i$, find optimal fully parenthesize product $A_1 A_2 \ldots A_n$ that minimizes number of scalar multiplications.

Chain of matrices $<A_1, A_2, A_3, A_4>$: five distinct ways

A_1: $p_1 \times p_2$ **A_2**: $p_2 \times p_3$ **A_3**: $p_3 \times p_4$ **A_4**: $p_4 \times p_5$

$(A_1(A_2(A_3A_4)))$ ← # of multiplication: $p_3p_4p_5 + p_2p_3p_5 + p_1p_2p_5$

$(A_1((A_2A_3)A_4))$

$((A_1A_2)(A_3A_4))$

$((A_1(A_2A_3))A_4)$

$(((A_1A_2)A_3)A_4)$

Find the one with minimal multiplications?

Matrix Chain Multiplication

- *Given* a chain $<A_1, A_2, \ldots A_n>$ of matrices, where matrix A_i has dimension $p_{i-1} x p_i$, find optimal fully parenthesize product $A_1A_2\ldots A_n$ that minimizes number of scalar multiplications.
- Let m[i, j] be the minimal # of scalar multiplications needed to calculate $A_iA_{i+1}\ldots A_j$ (m[1…n]) is what we want to calculate)
- Recurrence relation: how does m[i…j] relate to smaller problem
  - First decision: pick k (can be i, i+1, …j-1) where to divide $A_iA_{i+1}\ldots A_j$ into two groups: $(A_i\ldots A_k)(A_{k+1}\ldots A_j)$
  - $(A_i\ldots A_k)$ dimension is $p_{i-1} x p_k$, $(A_{k+1}\ldots A_j)$ dimension is $p_k x p_j$

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

Summary

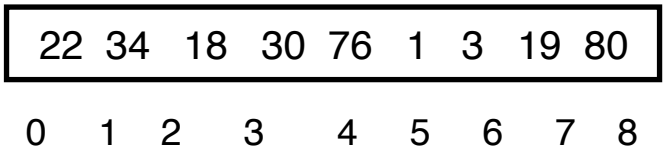
- Keys to DP
  - Recursive algorithm => optimal Substructure
  - overlapping subproblems
- Write recurrence relation for subproblem: i.e., how to calculate solution to a problem using sol. to smaller subproblems
- Implementation:
  - memoization (table+recursion)
  - bottom-up table based (smaller problems first)
- Insights and understanding comes from practice!

K-Sum Problem

- Given a list of integers, and an integer K
- Is there a subset of these numbers that adds up to K?
  - e.g., cannot use a number more than once
- Discussion: brute force approach?
  - How many subsets are there?
  - How to enumerate all subsets in program/code?

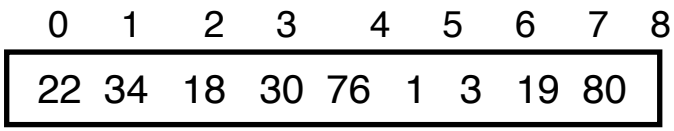
```
// return true if there is a subset of numbers taken from n[0…len-1]
//  that adds up to K
bool AddUpToK (n[], int len, int K)
```

- K=100, len=9, n:



K-Sum Problem

- bool AddUpToK (n[], n_len, int K)
  - K=100, n_len=9, n:



- Think recursively!
  - (base case) for what inputs do you know the answer right away?
    -

K-Sum Problem

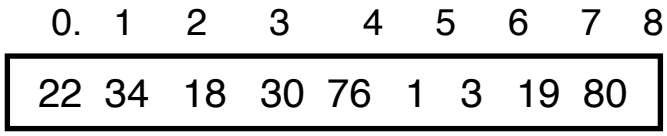
- bool AddUpToK (n[], len, int K)
  - K=100, len=9 n:

0.	1	2	3	4	5	6	7	8
22	34	18	30	76	1	3	19	80

- Think recursively!
  - (general case) for a general input, how to reduce it to smaller problem(s)?
    - Hint: try to make one single decision first?
    -

K-Sum Problem

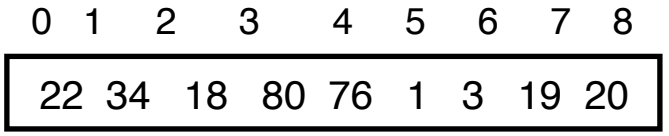
- bool AddUpToK (n[], len, int K)
  - K=100, len=9 n:



- Think recursively!
  - (general case) for a general input, how to reduce it to smaller problem(s)?
  - Decision: include last number, n[len-1], or not?
    - if included it, then we need to see if we can add up to K-n[len-1] using the rest of the numbers
    - if not, we need to see if we can add up to K using the rest of the numbers.
      - How to solve two smaller subproblems?
    - If either one returns true, return true

K-Sum Problem

```
bool AddUpToK (int n[], int len, int K)
{   //base cases
   if K==0 return true
   if K>0 and len==0 return false

   // general case: consider include last number, or not?
   if  ( AddUpToK (n, len -1, K-n[len-1]) )
        return true;     //we can make K by including last num…
   else
        if (AddUpToK (n, len-1, K))
             return true; //we can make K without using n[len-1]
    else
         return false;  //not possible
   }
```



K-Sum Problem

```
bool AddUpToK (int n[], int len, int K)
{   //base cases
   if K==0 return true
   if K>0 and len==0 return false

   // general case: consider include last number, or not?
   if  ( AddUpToK (n, len-1, K-n[len-1]) )
        return true;     //we can make K by including last num…
   else
        if (AddUpToK (n, len-1, K))
             return true; //we can make K without using n[len-1]
    else
         return false;  //not possible
   }
```

0	1	2	3	4	5	6	7	8
22	34	18	80	76	1	3	19	20

Draw recursion tree for AddUpToK (n,9,100)

Overlapping subproblems?

K-Sum Problem: tabulation

```
/*whether there is a subset of these numbers that add up to K, and output one such subset */
bool AddUpToK_Tabulation (int numbers[], int num_len, int K)
{   bool C[K+1][num_len+1];
     // C[k][n]: can we add up to k using  numbers[1…n]
    for n=0 to num_len C[0][n] = 0 // if K==0 return true
    for k=1 to K C[k][0] = false

    //fill in array row by row, left to right
    for k=1 to K
       for n=1 to num_len
            if C[k][n-1]==true C[k][n]=true  //we can make k without last number

            //otherwise, can we include numbers[n] to make k?
            else if k==numbers[n]
                 C[k][n] = true;
           else  if k>numbers[n] and C[k-numbers[n]][n-1]==true
                C[k][n] = true;
           else //k<numbers[n] or cannot make k-numbers[n] using numbers[1…n-1]
              C[k][n] = false
   return C[K][num_len];
  }
```

0	1	2	3	4	5	6	7	8
22	34	18	80	76	1	3	19	20