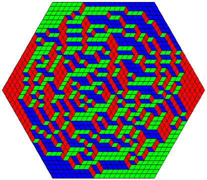
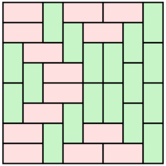
Unit II: Dynamic Programming

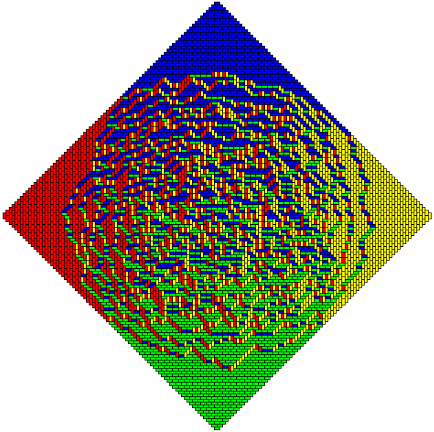
CISC 380 Algorithms

Dr. Sarah Miracle

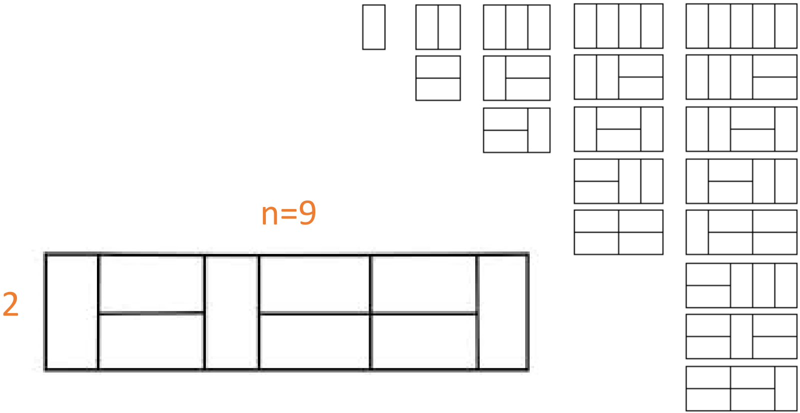
Domino Tilings







Domino Tilings



Problem: Find the number of domino tilings of a 2 by n region.

Example: Domino Tilings WITH recursion

Problem: Count the number of domino tilings of a $2 \times n$ region.

function $\textsc{DominoTilings}(n)$
    if $n \leq 0$ **then return** 0
    if $n == 1$ **then return** 1
    if $n == 2$ **then return** 2
    return $\textsc{DominoTilings}(n-1) + \textsc{DominoTilings}(n-2)$

The running time exponential!

A Bottom-Up Approach to Fibonacci

function $\text{FIB2}(n)$
    if $n = 0$ **then**
        return 0
    if $n = 1$ **then**
        return 1
    Create an array $F[0 \ldots n]$ $F[0] = 0, F[1] = 1$
    for $i = 2 \to n$ **do**
        $F[i] = F[i-1] + F[i-2]$
    return $F[n]$

Recursive Version of Longest Increasing Subsequence

WARNING: THIS CODE BLOWS UP!!!!

function $\mathrm{LIS}(a_1, \ldots, a_m)$
    if $m = 1$ **then**
        return 1
    $maxSoFar = 1$
    for $i = 1 \to m - 1$ **do**
        if $a_i < a_m$ **then**
            $\text{solnI} = \mathrm{LIS}(a_1, \ldots, a_i)$
            if $solnI + 1 > maxSoFar$ **then**
                $maxSoFar = solnI + 1$
    return $maxSoFar$

Longest Increasing Subsequence

Let $L[j]$ = the length of the longest increasing subsequence in $a_1, \ldots, a_j$ which ends at a_j <u>and</u> includes a_j.

```
function LIS(A = [a_1, ..., a_n])
    for j = 1 → n do                        ▷ solve each of the n subproblems
        L[j] = 1
        for i = 1 → j − 1 do
            if (L[i] + 1 > L[j] AND a_i < a_j) then
                L[j] = L[i] + 1
    Let max = 1
    for k = 1 → n do                        ▷ find the max subproblem sol'n
        if (L[k] > L[max]) then
            max = k
    return (L[max])
```

Longest Increasing Subsequence: Finding the Sequence

This modified code keeps track of the next to last index i that gives the max in the *Prev* array. This allows us to find the actual subsequence and not just the length.

function $\mathrm{LIS}(A = [a_1, \ldots, a_n])$
  for $j = 1 \to n$ **do** ▷ solve each of the n subproblems
    $L[j] = 1$, $Prev[j] = \text{NULL}$
    for $i = 1 \to j - 1$ **do**
      if $(L[i] + 1 > L[j]$ AND $a_i < a_j)$ **then**
        $\mathrm{L[j]} = \mathrm{L[i]} + 1$
        $Prev[j] = i$
  Let $max = 1$
  for $k = 1 \to n$ **do** ▷ find the max subproblem sol'n
    if $(L[k] > L[max])$ **then**
      $max = k$

Longest Increasing Subsequence: Finding the Sequence

This code allows us to use the values stored in L and $Prev$ to output the actual longest increasing subsequence.

Let $max = 1$
for $k = 1 \to n$ **do** ▷ find the max subproblem sol'n
    if $(L[k] > L[max])$ **then**
        $max = k$
Let $i = max$
output(a_i) ▷ output the LIS ending at (and including) max
while $(Prev[i] \neq \text{NULL})$ **do**
    $i = Prev[i]$
    ouptut(a_i)

Longest Increasing Subsequence: Finding the Subsequence

This is an alternative (more general) approach for recovering the LIS that does not use the *Prev* array.

function $\textsc{RecoverLIS}(i, L)$
  if $i = NULL$ **then**
    return ()
  output(a_i) ▷ find the previous index - Prev[i]
  $maxSoFar = 1$
  $maxSoFarINDEX = NULL$
  for $i = 1 \rightarrow m - 1$ **do**
    if $a_i < a_m$ **then**
      $solnI = L[i]$
      if $solnI + 1 > maxSoFar$ **then**
        $maxSoFar = solnI + 1$
        $maxSoFarIndex = i$
  $\textsc{RecoverLIS}(maxSoFarIndex, L)$

Longest Increasing Subsequence: Example

A	100	9	10	200	300	12	13
L	?	?	?	?	?	?	?
Prev	?	?	?	?	?	?	?

Longest Increasing Subsequence: Example

Solution:

A	100	9	10	200	300	12	13
L	1	1	2	3	4	3	4
Prev	null	null	2	3	4	3	6

Longest Common Subsequence (LCS)

Let $L[i][j]$ = length of LCS of $x_1 x_2 \ldots x_i$ with $y_1 y_2 \ldots y_j$

```
function LCS(X = x_1 x_2 ... x_n, Y = y_1 y_2 ... y_m )
    for i = 0 → n do                              ▷ initialize first column 0
        L[i][0] = 0
    for j = 0 → m do                              ▷ initialize first row 0
        L[0][j] = 0
    for i = 1 → n do                              ▷ each row
        for j = 1 → m do                          ▷ each column
            if x_i == y_j then
                L[i][j] = 1 + L[i − 1][j − 1]
            else
                L[i][j] = max{L[i − 1][j], L[i][j − 1]}
    return (L[n][m])
```

Longest Common Subsequence (LCS): Recursive Sol'n (Attempt 1)

function $\mathrm{LCSR}(X = x_1 x_2 \ldots x_i,\ Y = y_1 y_2 \ldots y_j\ )$
    if $i == 0$ OR $j == 0$ **then** $\triangleright$ base case
        return 0
    if $x_i == y_i$ **then** $\triangleright$ last characters match
        result $= 1 + \mathrm{LCSR}([x_1, \ldots, x_{i-1}], [y_1, \ldots, y_{j-1}])$
    else $\triangleright$ last characters are different
        result $= \max\{\mathrm{LCSR}([x_1, \ldots, x_{i-1}], Y)$,
$\mathrm{LCSR}(X, [y_1, \ldots, y_{j-1}]\ )$
    return result

The running time is exponential!

Longest Common Subsequence (LCS): Memoization

Initialize a n by m matrix A to unknown

function $\mathrm{LCSM}(X = x_1 x_2 \dots x_i,\ Y = y_1 y_2 \dots y_j\ )$

  if $i == 0$ OR $j == 0$ **then** ▷ base case

    return 0

  if $A[i][j] \neq$ unknown **then** ▷ Check if already solved.

    return $A[i][j]$

  if $x_i == y_i$ **then** ▷ last characters match

    result $= 1 + \mathrm{LCSM}([x_1, \dots, x_{i-1}], [y_1, \dots, y_{j-1}])$

  else ▷ last characters are different

    result $= \max\{\mathrm{LCSM}([x_1, \dots, x_{i-1}], Y)\ ,$
$\mathrm{LCSM}(X, [y_1, \dots, y_{j-1}]\ )$

  $A[i][j] = result$ ▷ Store answer if not already solved.

  return result

Running Time: $O(mn)$

Longest Increasing Subsequence Revisited

Finding the actual LIS

```
function RecoverLIS(i, L)
    if i = NULL then
        return ()
    output(a_i)
    maxSoFar = 1.                          ▷ find the previous index - Prev[i]
    maxSoFarINDEX = NULL
    for i = 1 → m − 1 do
        if a_i < a_m then
            solnI = L[i]
            if solnI + 1 > maxSoFar then
                maxSoFar = solnI + 1
                maxSoFarIndex = i
    RecoverLIS(maxSoFarIndex, L)
```

LCS: Recovering the Subsequence

function $\textsc{RecoverLCS}(i, j, L, X, Y)$
  if $i = 0$ or $j = 0$ **then**
    return
  else if $x_i = y_i$ **then**
    $\textsc{Output}(x_i)$
    $\textsc{RecoverLCS}(i-1, j-1, L, X, Y)$
  else if $L[i, j-1] > L[i-1, j]$ **then**
    $\textsc{RecoverLCS}(i, j-1, L, X, Y)$
  else
    $\textsc{RecoverLCS}(i-1, j, L, X, Y)$

What is the running time?

LCS: Recovering the Subsequence

```
function RecoverLCS(i, j, L, X, Y)
    if i = 0 or j = 0 then
        return
    else if x_i = y_i then
        Output(x_i)
        RecoverLCS(i − 1, j − 1, L, X, Y)
    else if L[i, j − 1] > L[i − 1, j] then
        RecoverLCS(i, j − 1, L, X, Y)
    else
        RecoverLCS(i − 1, j, L, X, Y)
```

Running Time: $O(m+n)$

(m and n are the lengths of X and Y)

Knapsack: Without Repetition

Let $K[b][j]$ = max value achievable using a subset of objects $1, \ldots, j$ and total capacity b

```
function KnapsackNoRepeat(B, w_1, ..., w_n, v_1, ..., v_n )
    for j = 0 → n do                                  ▷ initialize first row 0
        K[0][j] = 0
    for b = 0 → B do                                  ▷ initialize first column 0
        K[b][0] = 0
    for j = 1 → n do                                  ▷ each row
        for b = 1 → B do                              ▷ each column
            if w_j > b then
                K[b][j] = K[b][j − 1]
            else
                K[b][j] = max{v_j + K[b − w_j][j − 1], K[b][j − 1]}
    return K[B][n]
```

Running Time: $O(nB)$

Knapsack: Without Repetition Example

Capacity: 5

object	1	2	3	4
weight	1	3	4	4
value	3	4	5	6

1. Fill in the dynamic programming table K.
2. What is the optimal value?
3. What subset does it correspond to?
4. How would you find the corresponding subset programmatically?

Knapsack: Without Repetition Example Soln

Capacity: 5

object	1	2	3	4
weight	1	3	4	4
value	3	4	5	6

Columns are objects and rows are capacities.

	0	1	2	3	4
0	0	0	0	0	0
1	0	3	3	3	3
2	0	3	3	3	3
3	0	3	4	4	4
4	0	3	7	7	7
5	0	3	7	8	9

Knapsack: Without Repetition Advanced Example

Capacity: 20

object	1	2	3	4	5
weight	2	3	4	5	9
value	3	4	5	8	10

1. Fill in the dynamic programming table K.
2. What is the optimal value?
3. What subset does it correspond to?
4. How would you find the corresponding subset programmatically?

Knapsack: Without Repetition Advanced Example

Capacity: 20

object	1	2	3	4	5
weight	2	3	4	5	9
value	3	4	5	8	10

Columns are objects and rows are capacities.

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

	0	1	2	3	4	5
11	0	3	7	12	16	16
12	0	3	7	12	17	17
13	0	3	7	12	17	17
14	0	3	7	12	20	20
15	0	3	7	12	20	20
16	0	3	7	12	20	21
17	0	3	7	12	20	22
18	0	3	7	12	20	23
19	0	3	7	12	20	25
20	0	3	7	12	20	26

Knapsack: Without Repetition

How to recover the subset of items? Work backwards!

1. run KnapsackNoRepeat to populate K
2. call the recursive function below with parameters B, n to output the actual subset of items

function $\textsc{RecoverItems}$(b,j)
    if $b = 0$ OR $j = 0$ **then** ▷ Base Case
        return
    if $K[b][j] = K[b][j-1]$ **then** ▷ We didn't use jth item
        $\textsc{RecoverItems}(b, j-1)$
    else
        Output Item j ▷ We did use jth item
        $\textsc{RecoverItems}(b - w_j, j-1)$

Running Time: $O(n)$

CISC 380: Today's Agenda

- Class Logistics
  - Welcome to Zoom!
  - Changes to Student Notetaker Assignment
  - Assignment 4 posted today (or tomorrow)
- Knapsack WITH repetition
  - Subproblem definition & recurrence
  - Filling the table - iteration & memoization
- On your own: Knapsack WITHOUT Repetition Example (2 slides before this)

Knapsack: WITH Repetition

Recall the solution to Knapsack WITHOUT repetition:

$K[b][j]$ = max value achievable using a subset of objects $1, \ldots, j$ and total capacity b If $w_j \leq b$, then

$$K[b, j] = \max\{v_j + K[b - w_j, j - 1], K[b, j - 1]\}$$

else

$$K[b, j] = K[b, j - 1]$$

Base Cases:

$$K[b, 0)] = 0, K[0, j] = 0$$

Now we are allowed to use objects multiple times (Knapsack WITH repetition). Come up with a subproblem & recurrence for this new problem.

Knapsack: With Repetition

Let $K[b]$ = max value achievable with capacity b, all objects $1, \ldots, n$ are allowed.

function $\textsc{KnapsackWithRepeat}(B,\ w_1, \ldots, w_n,\ v_1, \ldots, v_n)$
  $K[0] = 0$
  for $b = 0 \to B$ **do**
    $K[b] = 0$
    for $l = 1 \to n$ **do** $\triangleright$ each possible "last" object
      if $w_l \leq b$ AND $K[b] < K[b - w_l] + v_l$ **then**
        $K[b] = K[b - w_l] + v_l$
  return $K[B]$

Running Time: $O(nB)$

Knapsack: WITH Repetition Example

Capacity: 10

object	1	2	3	4
weight	6	3	4	2
value	30	14	16	9

1. Fill in the dynamic programming table K.
2. What is the optimal value?
3. What subset does it correspond to?
4. How would you find the corresponding subset programmatically?

Knapsack: WITH Repetition Example Soln

Capacity: 10

object	1	2	3	4
weight	6	3	4	2
value	30	14	16	9

	0	1	2	3	4	5	6	7	8	9	10
B:	0	0	9	14	18	23	30	32	39	44	48

Knapsack: WITH Repetition Example

Capacity: 300

object	1	2
weight	100	200
value	50	150

Chain Matrix Multiply

Subproblem: $C[i][j]$ = min cost of computing $A_i \times \ldots \times A_j$

Let $s = j - i$ be the "width" of the subproblem $C[i][j]$

```
function ChainMatrixMultiply(m_0, m_1, ..., m_n)
    for i = 1 → n do                                    ▷ Base Case
        C[i][i] = 0
    for s = 1 → n − 1 do                                ▷ each possible "width"
        for i = 1 → n − s do
            Let j = i + s
            C[i][j] = ∞
            for l = i → j − 1 do
                if C[i][j] > m_{i−1} m_l m_j + C[i][l] + C[l + 1][j] then
                    C[i][j] = m_{i−1} m_l m_j + C[i][l] + C[l + 1][j]
    return C[1][n]
```

Chain Matrix Multiply

Subproblem: $C[i][j] =$ min cost of computing $A_i \times \ldots \times A_j$

Let $s = j - i$ be the "width" of the subproblem $C[i][j]$

function $\textsc{ChainMatrixMultiply}(m_0, m_1, \ldots, m_n)$
  for $i = 1 \to n$ **do** ▷ Base Case
    $C[i][i] = 0$
  for $s = 1 \to n-1$ **do** ▷ each possible "width"
    for $i = 1 \to n - s$ **do**
      Let $j = i + s$
      $C[i][j] = m_{i-1}m_i m_j + C[i][i] + C[i+1][j]$
      for $l = i+1 \to j-1$ **do**
        if $C[i][j] > m_{i-1}m_l m_j + C[i][l] + C[l+1][j]$ **then**
          $C[i][j] = m_{i-1}m_l m_j + C[i][l] + C[l+1][j]$
  return $C[1][n]$

Chain Matrix Multiply: Example

$$A \times B \times C \times D$$

$$A = 5 \times 2, \quad B = 2 \times 1, \quad C = 1 \times 1, \quad D = 1 \times 10$$

	1	2	3	4
1	?	?	?	?
2		?	?	?
3			?	?
4				?

Chain Matrix Multiply: Example Soln

$$A \times B \times C \times D$$

$$A = 5 \times 2, \quad B = 2 \times 1, \quad C = 1 \times 1, \quad D = 1 \times 10$$

	1	2	3	4
1	0	10	12	62
2		0	2	22
3			0	10
4				0

CMM: Recovering the Solution

```
Recovery(i,j)
    if (i == j)  print "A_" + i
    else
        l = ``best l"
        //to get l you could:
        //(1)store it in a ``prev" table as you fill C
        //(2)or use the recurrence to recompute it here
        print "("
        Recovery(i,l)
        print ") x ("
        Recovery(l+1,j)
        print ")"
```

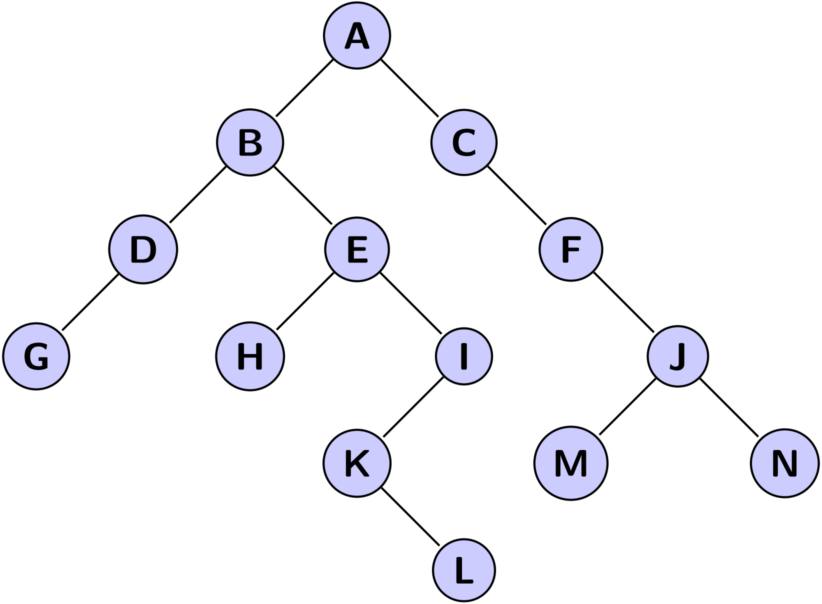
What is the running time? Why is this not exponential? Why do we not need to use memoization for this recovery code?

Independent Sets in Trees

Subproblem: $I(j)$ = size of the maximum IS in the subtree rooted at j.

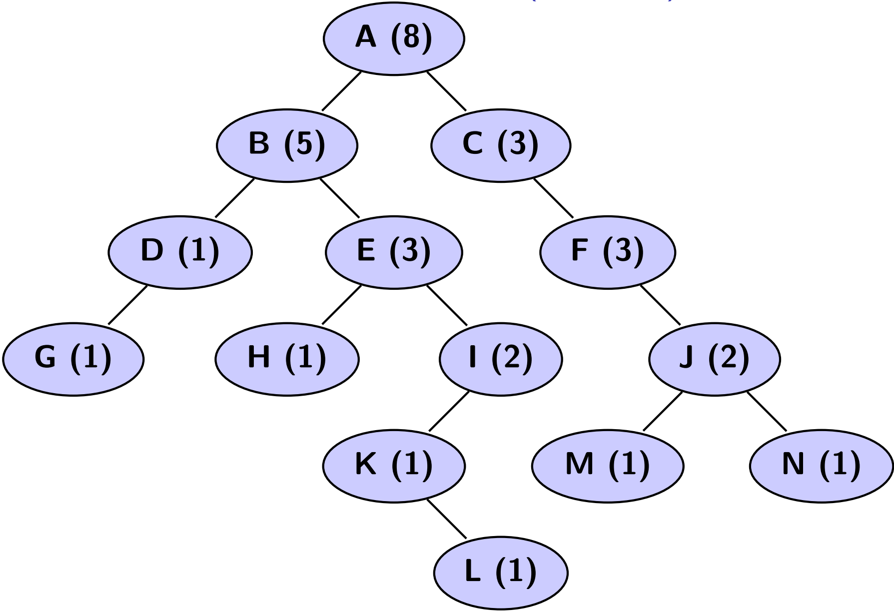
function $\textsc{MaxIS}$(node n)
  if n is a leaf **then** ▷ Base Case
    return 1
  if n.maxIS $\neq$ NULL **then** ▷ Check if it's already solved.
    return n.maxIS
  else
    n.maxIS = max$\{\sum_{k \in n.children} \textsc{MaxIS}(k), 1 + \sum_{k \in n.children} \sum_{j \in k.children} \textsc{MaxIS}(j)\}$
  return n.maxIS

Example: Independent Sets in Trees



Find the size of the max IS in the subtree rooted at each node.

Example: Independent Sets in Trees (Solution)



Find the size of the max IS in the subtree rooted at each node.

Independent Sets in Trees

How to recover the actual independent set.

```
function RecoverSet(node n)
    if  n is a leaf then                                  ▷ Base Case
        output n
    else
        if  n.maxIS == Σ_{k∈n.children} k.maxIS  then
            for all k ∈ n.children do
                RecoverSet(k)
        else                                              ▷ n is in the max IS
            output n
            for all k ∈ n.children do
                for all j ∈ k.children do
                    RecoverSet(j)
```