CS-3510: Design and Analysis of Algorithms

Dynamic Programming III

Instructor: Shahrokh Shahi

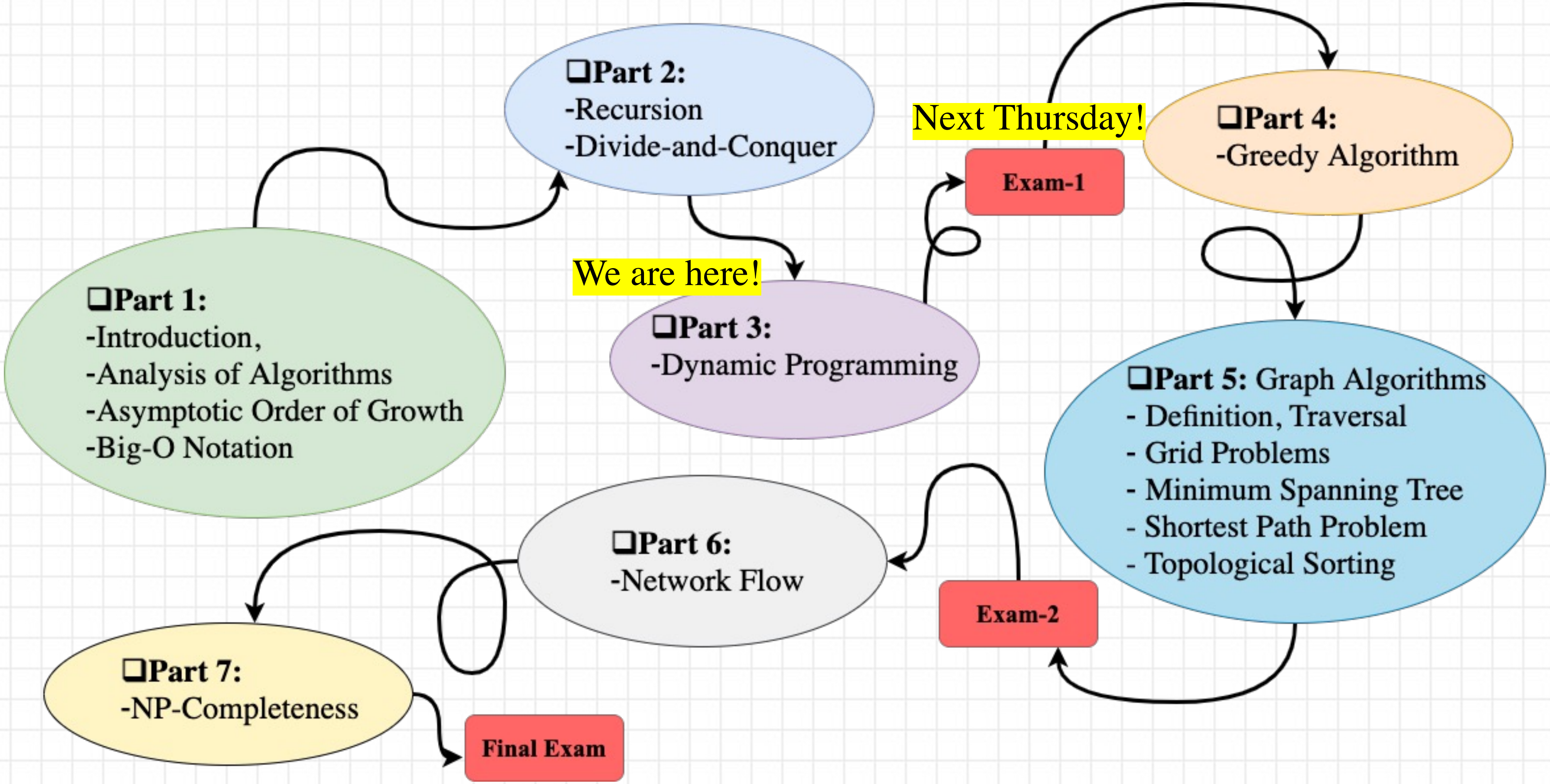
College of Computing
Georgia Institute of Technology
Summer 2022

Overview

- Part 1
  - Dynamic programming

- Part 2:
  - Exam 1 Review

Roadmap



❑Part 1:
-Introduction,
-Analysis of Algorithms
-Asymptotic Order of Growth
-Big-O Notation
❑Part 2:
-Recursion
-Divide-and-Conquer
We are here!
❑Part 3:
-Dynamic Programming
Next Thursday!
Exam-1
❑Part 4:
-Greedy Algorithm
❑Part 5: Graph Algorithms
- Definition, Traversal
- Grid Problems
- Minimum Spanning Tree
- Shortest Path Problem
- Topological Sorting
Exam-2
❑Part 6:
-Network Flow
❑Part 7:
-NP-Completeness
Final Exam

Dynamic Programming (DP)

- Dynamic Programming vs. Divide-and-Conquer

Divide-and-Conquer:

- Divide problem into subproblems
- Recursively solve the subproblems and aggregate solutions

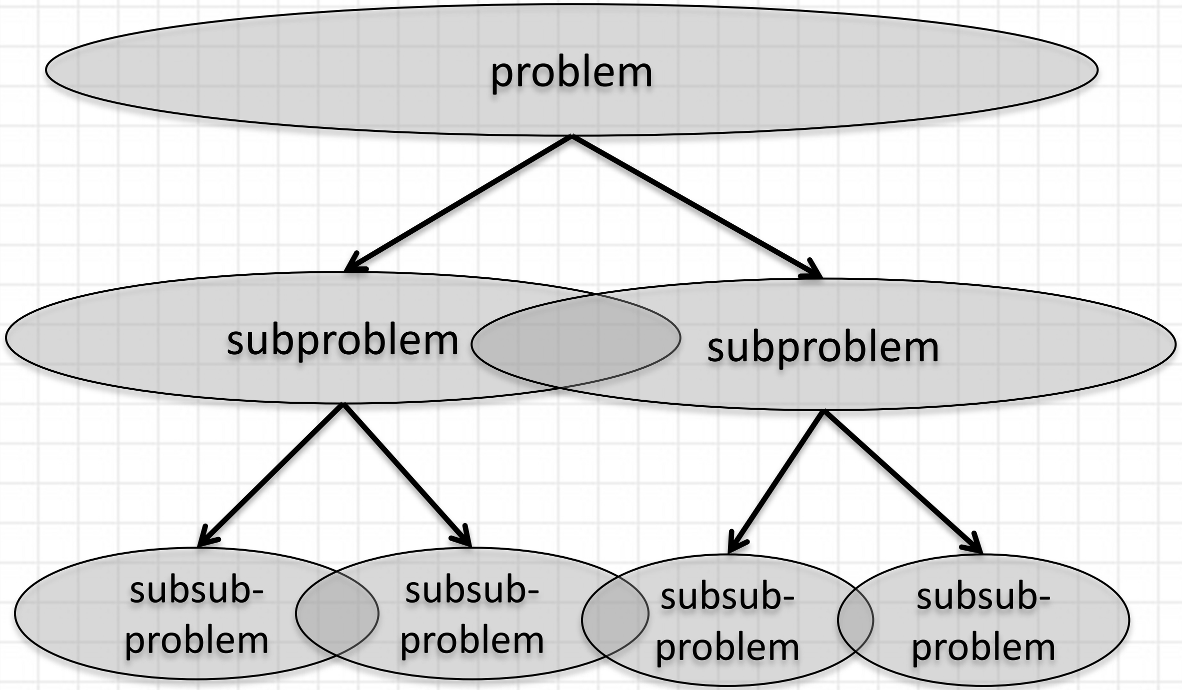
Note: The subproblems do not overlap

Dynamic Programming

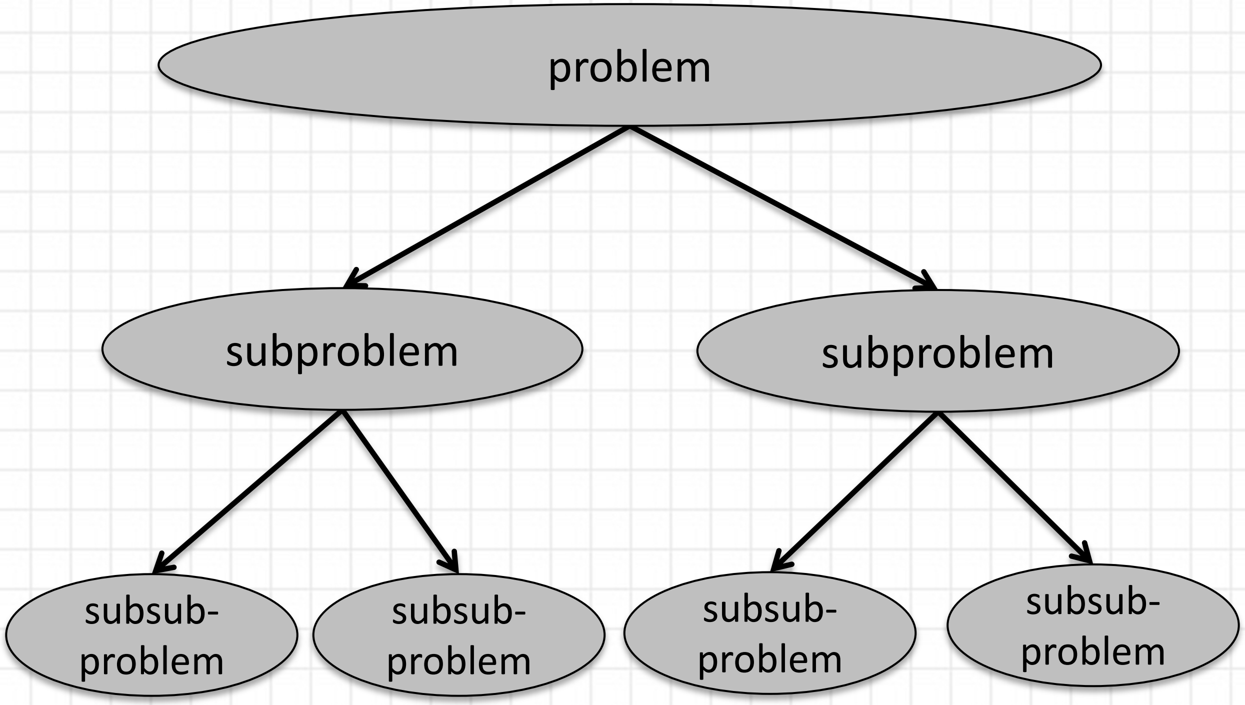
- Divide problem into subproblems, recursively solve them
- Subproblems overlap
- When a subproblem has been solved, remember its solution and reuse that solution rather than resolving it later (memoization)

Dynamic Programming (DP)

- Dynamic Programming vs. Divide-and-Conquer



Subproblems overlap



Subproblems do not overlap

Dynamic Programming

- Top-down vs. Bottom-up Approach

  - "Top-down" dynamic programming
    - Begin with problem description
    - i.e., begin at root of tree and work downwards
    - Recursively subdivide problem into subproblems

Recursive with memoization

  - "Bottom-up" dynamic programming
    - Start at the leaf nodes of tree, i.e., the base case(s).
    - Build up solution to larger problem from solutions of the simpler subproblems

Iterative

Dynamic Programming (DP)

- Dynamic Programming Elements
  - DP often (not always!) applicable to optimization problems
    - Large number of possible solutions
    - Must find the "best" one (maximum or minimum)
  - "Optimal substructure"
    - Finding the optimal solution involves finding the optimal solution to subproblems
    - The subproblems are the same as the original problem, but are "smaller" (e.g., involve smaller-sized input data) Similar to D&C
  - "Overlapping subproblems" Key difference to D&C
    - Different subproblems operate on the same input data
    - Allows exploitation of memoization

Dynamic Programming (DP)

- Dynamic Programming Recipe
    1. Show the problem has optimal substructure, i.e., the optimal solution can be constructed from optimal solutions to subproblems (This step is concluded by writing the recurrence relation and its base case).
    2. Show subproblems are overlapping, i.e., subproblems may be encountered many times but note the total number of distinct subproblems is polynomial (Recall the recursion tree for Fibonacci and Rod-cutting problems, where the total number of distinct subproblems was linear, i.e., O(n)).
    3. Construct an algorithm that computes the optimal solution to each subproblem only once and reuses the stored result all other times (This can be done by using either top-down (recursive+memoization) or bottom-up (iterative) approach).
    4. Analysis: show that time and space complexity is polynomial.

DP Examples

- One-dimensional
  1. Fibonacci sequence
  2. Staircase climbing
  3. Rod-cutting
  4. Red-black game

- Two-dimensional
  5. Longest common subsequence (LCS)
  6. Coin-changing
  7. Knapsack

DP Example: (5) LCS (continue)

- Given two sequences:

$$X = \langle x_1, x_2, \ldots x_m \rangle$$
$$Y = \langle y_1, y_2, \ldots y_n \rangle$$

Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.

Compute: LCS(X,Y) = longest common subsequence of X and Y

Example:

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

<B, C, A> is a common subsequence of X and Y

<B, C, A, B> is an LCS of X and Y

<B, C, B, A> and <B, D, A, B> are also LCS’s of X and Y

(LCS may not be unique!)

DP Example: (5) LCS (continue)

- Given a sequence: $X = <x_1, x_2, \ldots x_m>$
  $X_i = <x_1, x_2, \ldots x_i>$ is defined as the i^{th} prefix of X, $i=0, 1, \ldots m$
  (X_i is the first i elements of X)

- Example: X = <A, B, C, B>
- $X_0 = <>$
- $X_1 = <A>$
- $X_2 = <A, B>$
- $X_3 = <A, B, C>$
- $X_4 = <A, B, C, B>$

- Key Observation:
- The LCS of sequences X and Y can be found by finding the LCS of prefixes of X and Y
- This leads to development of a recursive solution to computing LCS

DP Example: (5) LCS (continue)

- Compute the length of the LCS
  - Involves computing LCS of prefixes to X and Y
- Let $c[i,j] = LCS(X_i, Y_j)$
  - Data structure used for memoization

If ($x_m == y_n$):
- $z_k = x_m$;
- compute $LCS(X_{m-1}, Y_{n-1})$

Else:
- compute $LCS(X_{m-1}, Y)$ and $LCS(X, Y_{n-1})$
- pick the longer subsequence of the two

- $$c[i,j] = \begin{cases} 0, & \text{if } (i=0 \text{ or } j=0) \\ c[i-1,j-1] + 1, & \text{if } i>0, j>0, \text{ and } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i>0, j>0, \text{ and } x_i \neq y_j \end{cases}$$

- $c[m,n]$ is the length of $LCS(X, Y)$

LCS: Computation

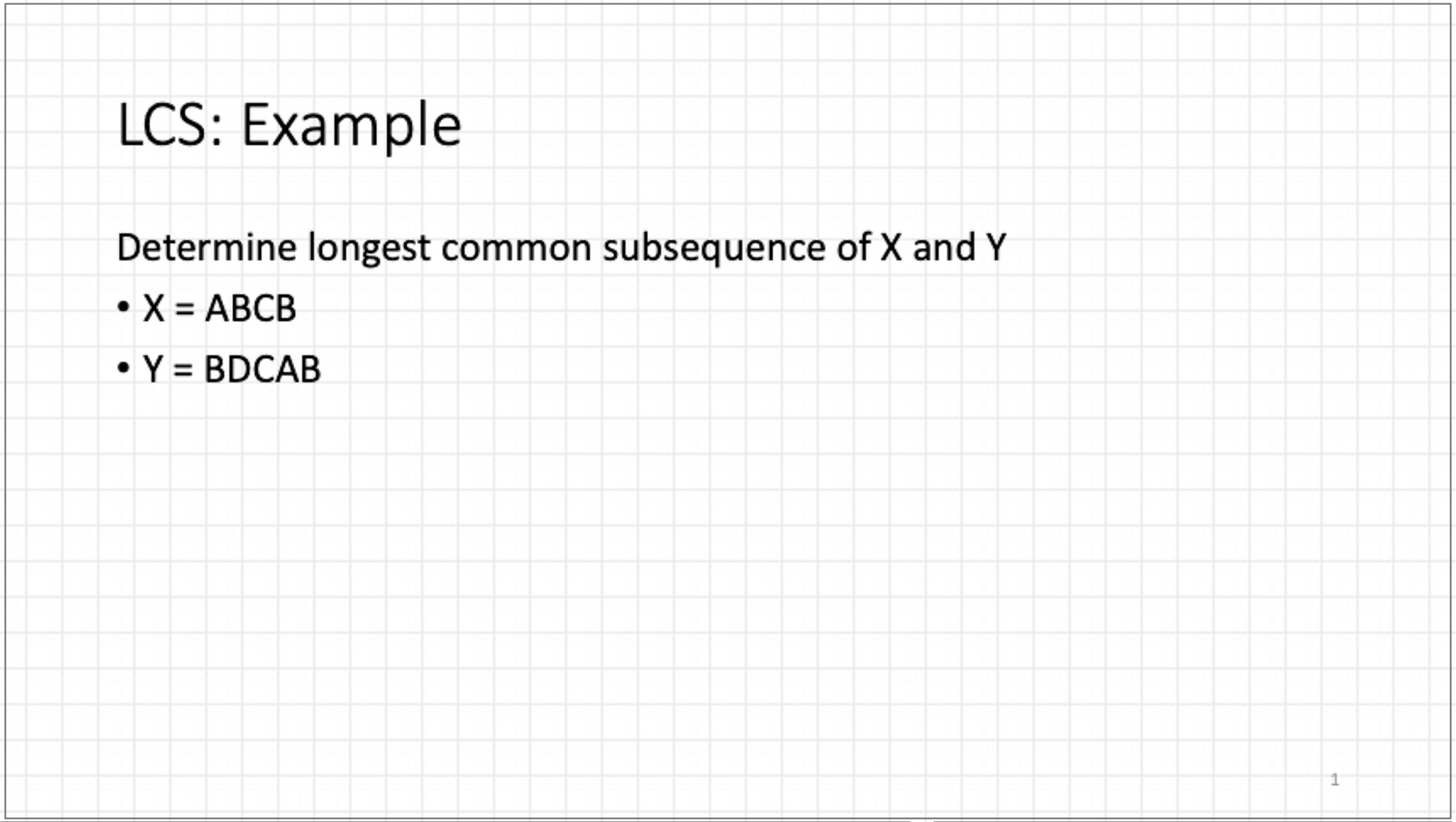
- $c[i,j] = 0$, if $(i=0$ or $j=0)$
  $= c[i-1,j-1] + 1$, if $i>0$, $j>0$, and $x_i = y_j$
  $= \max(c[i, j-1], c[i-1, j])$ if $i>0$, $j>0$, and $x_i \neq y_j$

```
// compute LCS for 0 length cases
for (i=0; i<=m; i++) c[i,0]=0;
for (j=0; j<=n; j++) c[0,j]=0;
// compute in row-major order
for (i=1; i<=m; i++)
      for (j=1; j<=n; j++)
            if (x_i==y_j) c[i][j]=c[i-1][j-1]+1;
            // c[i][j]=max(c[i-1][j],c[i][j-1])
            else if (c[i-1][j]>=c[i][j-1]): c[i][j] = c[i-1][j];
            else: c[i][j] = c[i][j-1];
```

LCS: Example

Determine longest common subsequence of X and Y

- X = ABCB
- Y = BDCAB



LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

LCS: Example

ABCB
BDCAB

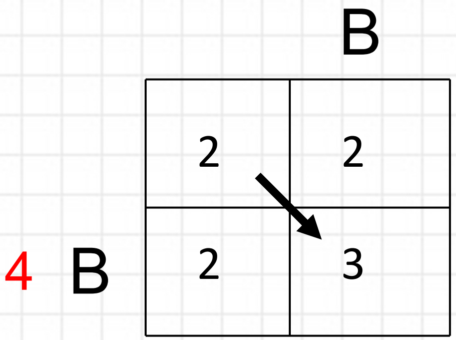
i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1	1	2	2	3

Length of LCS!

```
if (x_i==y_j) c[i][j]=c[i-1][j-1]+1;
else: c[i][j] = max(c[i-1][j],c[i][j-1])
```

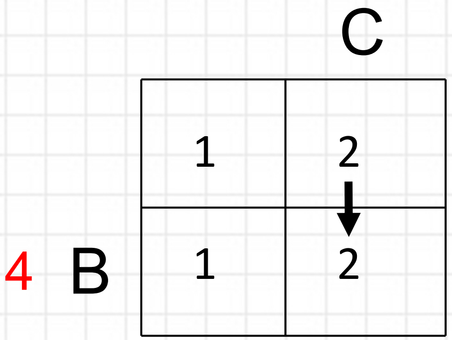
LCS: Computing the LCS

- The previous step determined the *length* of LCS, but not the LCS itself.
- Each c[i,j] depends on c[i-1,j] and c[i,j-1] or c[i-1, j-1]
- For each c[i,j] we can record how it was acquired:



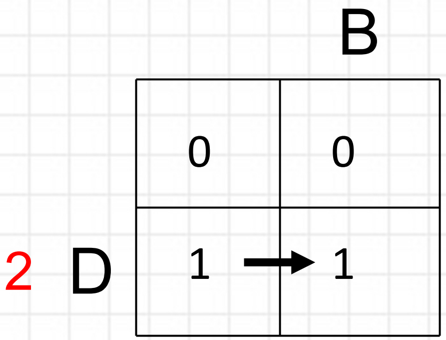

```
if (x_i==y_j)
    c[i][j]=
     c[i-1][j-1]+1;
```

“F”=found




```
else if (c[i-1][j]
       >= c[i][j-1])
c[i][j] = c[i-1][j];
```

“X”=advance X




```
else c[i][j] =
  c[i][j-1];
```

“Y”=advance Y

LCS: Computing the LCS

- Remember that

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

- So, we can start from *c[m,n]* and go backwards
- Whenever *c[i,j] = c[i-1, j-1]+1*, remember *x[i]* (because *x[i]* is a part of the LCS computed)
- When i=0 or j=0 (i.e., we reached the beginning), output the remembered letters in reverse order

LCS: Computing the LCS

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0,X	0,X	0,X	1,F	1,Y
2	B	0	1,F	1,Y	1,Y	1,X	2,F
3	C	0	1,X	1,X	2,F	2,Y	2,X
4	B	0	1,F	1,X	2,X	2,X	3,F

```
// annotate: found("F"),
// advance X("X"),advance Y("Y")
for (i=1; i<=m; i++)
  for (j=1; j<=n; j++)
    if (xi==yj):
      c[i][j]=c[i-1][j-1]+1;
      b[i][j]="F";
    else if (c[i-1][j]>=c[i][j-1])
      c[i][j] = c[i-1][j];
      b[i][j]="X";
    else
      c[i][j] = c[i][j-1];
      b[i][j]="Y";
```

LCS: Computing the LCS

i	Xi	j=0 Yj	1 B	2 D	3 C	4 A	5 B
0		0	0	0	0	0	0
1	A	0	0,X	0,X	0,X	1,F	1,Y
2	B	0	1,F	1,Y	1,Y	1,X	2,F
3	C	0	1,X	1,X	2,F	2,Y	2,X
4	B	0	1,F	1,X	2,X	2,X	3,F

LCS (reversed order): B C B → B C B (forward)

```
// annotate: found("F"),
// advance X("X"),advance Y("Y")
for (i=1; i<=m; i++)
  for (j=1; j<=n; j++)
    if (xi==yj):
      c[i][j]=c[i-1][j-1]+1;
      b[i][j]="F";
    else if (c[i-1][j]>=c[i][j-1])
      c[i][j] = c[i-1][j];
      b[i][j]="X";
    else
      c[i][j] = c[i][j-1];
      b[i][j]="Y";
```

LCS: Output (Printing) the LCS

```
// annotate: found("F"),
// advance X("X"),advance Y("Y")
for (i=1; i<=m; i++)
  for (j=1; j<=n; j++)
    if (x_i==y_j):
      c[i][j]=c[i-1][j-1]+1;
      b[i][j]="F";
    else if (c[i-1][j]>=c[i][j-1])
      c[i][j] = c[i-1][j];
      b[i][j]="X";
    else
      c[i][j] = c[i][j-1];
      b[i][j]="Y";
```

```
// to print LCS, call Print_LCS:
Print_LCS(b, X, m, n);


// follow annotations to print out
Print_LCS(b, X, i, j):
  if ((i==0) || (j==0)) return;
  if (b[i][j] == "F")
    Print_LCS(b, X, i-1, j-1);
    print (x);
  else if (b[i][j] == "X")
    Print_LCS(b, X, i-1, j);
  else
    Print_LCS(b, X, i, j-1);
```

LCS: Running Time

- What is the execution time for each step of this algorithm?
  - Step 1: Computing LCS
  - Step 2: Printing

LCS: Running Time

- What is the execution time for each step of this algorithm?
  - Step 1: Computing LCS
    - $O(m \times n)$ to fill in matrix
  - Step 2: Printing
    - $O(m+n)$

DP Example: (6) Coin-changing*

- Problem: We want to make change for S cents, and we have infinite supply of each coin in the set Coins = $[v_1, v_2, \dots, v_n]$, where v_i is the value of the i-th coin. What is the minimum number of coins required to reach value S?

* BRV: Benoit, A., Robert, Y., & Vivien, F. (2013). *A guide to algorithm design: paradigms, methods, and complexity analysis*. CRC Press.

DP Example: (6) Coin-changing

- Problem: We want to make change for S cents, and we have infinite supply of each coin in the set Coins = $[v_1, v_2, \dots, v_n]$, where v_i is the value of the i-th coin. What is the minimum number of coins required to reach value S?
- Choosing the maximum value first?
  - Counter example: S=8, Coins=[6, 4, 1]
    starting with max $v_i = 6$ → S = 6 + 1 + 1 → 3 coins,
    but the optimum value is S = 4 + 4 → 2 coins
- Solving more subproblems
  - Must be able to comeback to a choice already made and try another set of coins
  - Choosing a coin affects choosing the rest of them

DP Example: (6) Coin-changing

- Problem: We want to make change for S cents, and we have infinite supply of each coin in the set Coins = $[v_1, v_2, \dots, v_n]$, where v_i is the value of the i-th coin. What is the minimum number of coins required to reach value S?
- Define:
  $OPT(i, T)$ = min number of coins to reach $T \leq S$ with the first i coints $i \leq n$.
- Recurrence relation:

$$OPT(i,T) = \min \begin{cases} OPT(i-1,T) & ,\text{i} - \text{th coin not used} \\ OPT(i, T - v_i) + 1, & \text{i} - \text{th coin used at least once} \end{cases}$$

DP Example: (6) Coin-changing

- Define:
  $OPT(i, T)$ = min number of coins to reach $T \leq S$ with the first i coints $i \leq n$.

- Recurrence relation:

$$OPT(i,T) = \min \begin{cases} OPT(i-1,T) & \text{, i − th coin not used} \\ OPT(i, T - v_i) + 1 & \text{, i − th coin used at least once} \end{cases}$$

- Base cases:
  $OPT(0,T) = +\infty$ if $T > 0$ no coins, cannot reach to S
  $OPT(i,T) = +\infty$ if $T < 0$ too much change given, exceeded the sum.
  $OPT(i,0) = 0$ means we are done! we've used enough coins to reach S.

DP Example: (6) Coin-changing

- Define:
  $OPT(i, T)$ = min number of coins to reach $T \leq S$ with the first i coints $i \leq n$.

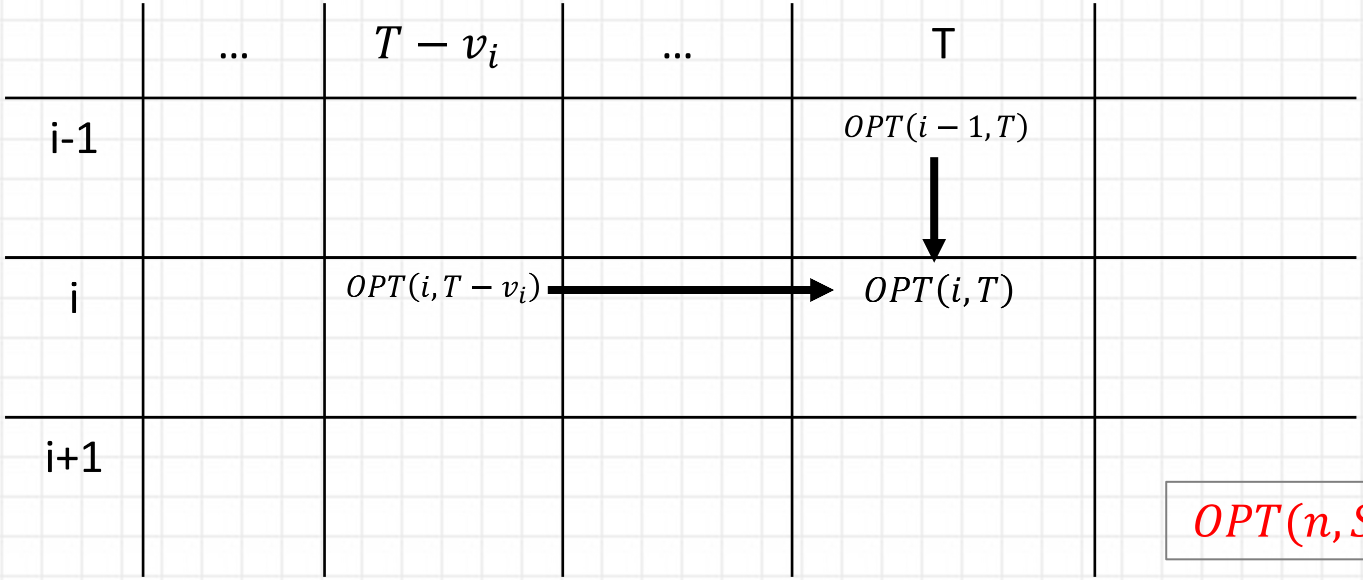
- Recurrence relation:

$$OPT(i,T) = \min \begin{cases} OPT(i-1,T) \\ OPT(i,T-v_i) + 1 \end{cases}$$

- Base cases:

$$OPT(0,T) = +\infty \text{ if } T > 0$$
$$OPT(i,T) = +\infty \text{ if } T < 0$$
$$OPT(i,0) = 0$$

- Dynamic programming
  - Top-down
  - Bottom-up



$OPT(n,S)$

DP Example: (6) Coin-changing

Demo

- Define:
  $OPT(i, T)$ = min number of coins to reach $T \leq S$ with the first i coints $i \leq n$.

- Recurrence relation:
  $$OPT(i, T) = \min \begin{cases} OPT(i-1, T) \\ OPT(i, T - v_i) + 1 \end{cases}$$

- Base cases:
  $OPT(0, T) = +\infty \quad \text{if} \quad T > 0$
  $OPT(i, T) = +\infty \quad \text{if} \quad T < 0$
  $OPT(i, 0) = 0$

- Dynamic programming
  - Top-down
  - Bottom-up

```python
def coin_change(coins, s):

  n = len(coins)

  # creating a 2D array
  opt = [[0] * (s+1) for _ in range(n+1)]

  # OPT(0, T) = +∞
  for t in range(s+1):
    opt[0][t] = float("inf")

  for i in range(1, n+1):
    vi = coins[i-1]
    for t in range(1, s+1):
      opt[i][t] = opt[i-1][t]
      if t - vi >= 0:
        opt[i][t] = min(opt[i][t], opt[i][t-vi]+1)

  return opt[n][s]
```

DP Example: (7) Knapsack

- Given n items and a "knapsack."
- Item i weights $w_i > 0$ and value $v_i > 0$
- Knapsack has weight capacity of W.
- Goal: Pack knapsack such that the total value is maximized.

DP Example: (7) Knapsack

- Given n items and a "knapsack."
- Item i weighs $w_i > 0$ and value $v_i > 0$
- Knapsack has weight capacity of W.
- Goal: Pack knapsack such that the total value is maximized.
- Examples
  - $\{1, 2, 5\}$
    Total value = 1+6+28 = 35
    Total weight = 1 + 2 + 7 = 10 ≤ 11
  - $\{3, 4\}$
    Total value = 18+22 = 40
    Total weight = 5+6 = 11 ≤ 11
  - $\{3, 5\}$
    Total value = 18+28 = 46
    Total weight = 5 + 7 = 12 ≰ 11

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Weight limit W = 11

DP Example: (7) Knapsack

- Given n items and a "knapsack". Item i weighs $w_i > 0$ and value $v_i > 0$. Knapsack has weight capacity of W. Pack knapsack such that the total value is maximized.

- Possible subproblems?
  - $OPT(i)$: optimal value with items $1, 2, \ldots, i$ ($i \leq n$)
  - $OPT(w)$: optimal value with weight limit w ($w \leq W$)

DP Example: (7) Knapsack

- Given n items and a "knapsack". Item i weighs $w_i > 0$ and value $v_i > 0$. Knapsack has weight capacity of W. Pack knapsack such that the total value is maximized.

- Possible subproblems?
  - $OPT(i)$: optimal value with items $1, 2, \ldots, i$ ($i \leq n$)
  - $OPT(w)$: optimal value with weight limit w ($w \leq W$)

We need to know both selected items and the remaining wight limit.

DP Example: (7) Knapsack

- Given n items and a "knapsack". Item i weighs $w_i > 0$ and value $v_i > 0$. Knapsack has weight capacity of W. Pack knapsack such that the total value is maximized.

- Possible subproblems?
  - $OPT(i)$: optimal value with items $1, 2, \ldots, i$ $(i \leq n)$
  - $OPT(w)$: optimal value with weight limit w $(w \leq W)$
  - $OPT(i, w)$: optimal value with items $1, 2, \ldots, i$ subject to weight limit w

We need to know both selected items and the remaining wight limit.

DP Example: (7) Knapsack

- Def: $OPT(i, w)$ = max profit subset of items $1, 2, \ldots, i$ with weight limit w
- Goal: $OPT(n, W)$
- Possible cases:
  - $OPT(i, w)$ does not select item i (because $w_i > w$) → select best of $1, 2, \ldots, i-1$
  - $OPT(i, w)$ selects item i → collect v_i→ new weight limit $w - w_i$
- Recurrence relation:

$$OPT(i, w) = \begin{cases} OPT(i-1, w) & \text{if } w_i > w \\ \max\begin{cases} OPT(i-1, w) \\ OPT(i-1, w-w_i) + v_i \end{cases} & \text{Otherwise} \end{cases}$$

- Base case:

$$OPT(0, w) = 0$$

DP Example: (7) Knapsack

- Def: $OPT(i, w)$ = max profit subset of items $1, 2, \dots, i$ with weight limit w
- Goal: $OPT(n, W)$
- Recurrence relation:

$$OPT(i,w) = \begin{cases} OPT(i-1,w) & \text{if } w_i > w \\ \max\begin{cases} OPT(i-1,w) \\ OPT(i-1,w-w_i)+v_i \end{cases} & \text{Otherwise} \end{cases}$$

- Base case: $OPT(0, w) = 0$

	…	$w - w_i$	…	w	
i-1		$OPT\binom{i-1}{w-w_i}$		$OPT\binom{i-1}{w}$	
i				$OPT\binom{i}{w}$	
i+1					

$\boxed{OPT(n, W)}$

DP Example: (7) Knapsack

- Dynamic Programming

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ v_i + OPT(i-1, w-w_i) \,\} & \text{otherwise} \end{cases}$$

KNAPSACK($n, W, w_1, \ldots, w_n, v_1, \ldots, v_n$)

FOR $w = 0$ TO W

  $M[0, w] \leftarrow 0.$

FOR $i = 1$ TO n

  FOR $w = 0$ TO W

    IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w].$

    ELSE $M[i, w] \leftarrow \max\{\, M[i-1, w],\ v_i + M[i-1, w-w_i] \,\}.$

previously computed values

RETURN $M[n, W]$.

Complexity?

Time: $\Theta(nW)$

Space: $\Theta(nW)$

DP Example: (7) Knapsack

- Example

i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

weight limit w

subset of items 1, …, i	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

OPT(i, w) = optimal value of knapsack problem with items 1, …, i, subject to weight limit w

DP: Summary

- Dynamic programming is a general algorithm approach similar to divide and conquer, but with shared/overlapped subproblems rather than disjoint ones.
- Efficiency is obtained by recording (memoization) the solution of subproblems rather than recomputing them.
- Dynamic programming applicable to many optimization problems
- Two main elements:
  - Optimal substructure
  - Overlapping subproblems

References

- The lecture slides are heavily based on the suggested textbooks and the corresponding published lecture notes:

- CLRS: Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009.
- KT: Kleinberg, J., & Tardos, E. Algorithm design. Pearson/Addison-Wesley, 2006.
- DPV: Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. Algorithms, McGraw-Hill Higher Education., 2008.
- BRV: Benoit, A., Robert, Y., & Vivien, F. (2013). *A guide to algorithm design: paradigms, methods, and complexity analysis*. CRC Press.
- Slides by Kevin Wayne. Copyright © 2005 Pearson-Addison Wesley.
- Slides by Elizabeth Cherry, Georgia Institute of Technology.

CS-3510: Design and Analysis of Algorithms

Exam 1: Review

Instructor: Shahrokh Shahi

College of Computing
Georgia Institute of Technology
Summer 2022

Exam 1

- Date: Thursday, June 09, 2022
- Time: 03:30 pm – 05:00 pm
- Location: Klaus 2443

- Closed book; No calculator
- One page sheet of notes
  - Letter size
  - Both sides
  - Typed or hand-written

Exam 1

- Contents:
  - Asymptotic order of growth, time and space complexity
  - Divide-and-conquer
  - Dynamic programming

Exam 1: Time Complexity

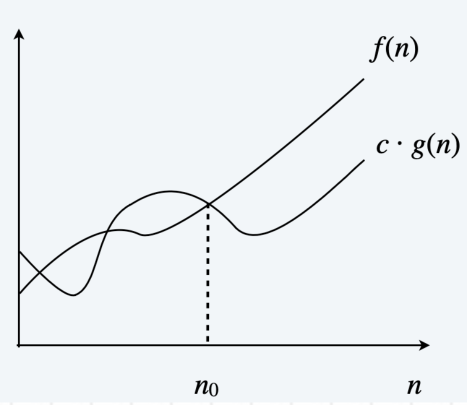
- Asymptotic Order of Growth
  - It is easier to talk about the lower bound and upper bound of the running time.
  - To practically deal with time complexity analysis, we use asymptotic notations.
  - The asymptotic growth of a function (in this case $T(n)$) is specified using Θ, O, and Ω notations.
  - Asymptotic means for "very large" input size, as n grows without bound or "asymptotically".

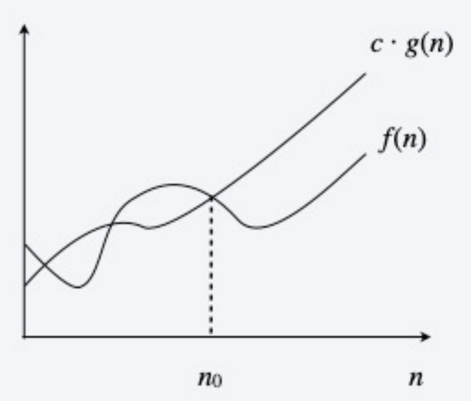
Exam 1: Time Complexity

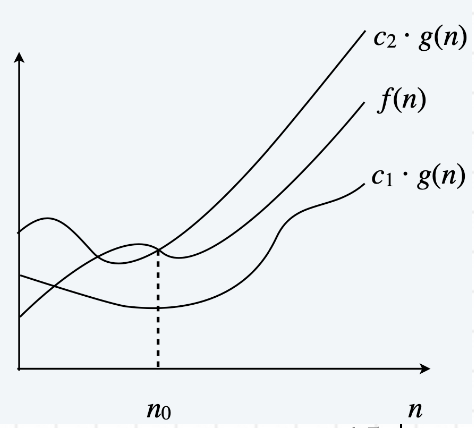
- Asymptotic Order of Growth
  - In general, the asymptotic notations define bounds on the growth of a function. Informally, a function $f(n)$ is:
    - $\Omega(g(n))$ if $g(n)$ is an asymptotic lower bound for $f(n)$
    - $\mathrm{O}(g(n))$ if $g(n)$ is an asymptotic upper bound for $f(n)$
    - $\Theta(g(n))$ if $g(n)$ is an asymptotic tight bound for $f(n)$

Exam 1: Time Complexity

- Asymptotic Order of Growth (Formal definition):
  - **Big Omega (lower bound):**
    $f(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \geq cg(n) \geq 0$ for all $n \geq n_0$.
  - **Big O (upper bound):**
    $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$
  - **Big Theta (tight bound):**
    $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$.
  - Note: $f(n)$ is $\Theta(g(n))$ iff $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$







Exam 1: Time Complexity

- Big O Notation Properties

Reflexivity	f is $O(f)$
Constants	If f is $O(g)$ and $c > 0$, then cf is $O(g)$
Products	If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 f_2$ is $O(g_1\, g_2)$
Sums (Additivity)	If f_1 is $O(g_1)$ and f_2 is $O(g_2)$, then $f_1 + f_2$ is $O(\max\,\{g_1, g_2\})$ Ex. If $f_1 \in O(n^2)$ and $f_2 \in O(n^4)$. Then, $f_1 + f_2 \in O(n^4)$
Transitivity	If f is $O(g)$ and g is $O(h)$, then f is $O(h)$

- So, we can ignore the lower terms and constants:
  - Ex. $f = 2n^3 + 4n^2 - 5n + 1 \in O(n^3)$
  - Ex. $f = 4n^5 \in O(n^5)$

Exam 1: Time Complexity

- Asymptotic Bounds for Some Common Functions

Polynomials	$f(n) = a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ and thus, $O(n^d)$ if $a_d > 0$.
Logarithms	$\log_a n$ is $\Theta(\log_b n)$ for every $a>1$ and $b>1$. Note: $O(\log_a n) = O(\log_b n)$ (Recall $\log_b n = \log_b a \times \log_a n$)
Logarithms vs polynomials	$\log_a n$ is $O(n^d)$ for every $a>1$ and $d>0$. Logarithms grow slower than every polynomial regardless of how small d is.
Exponential vs Polynomials	n^d is $O(r^n)$ for every $d>0$ and $r>1$. Exponentials grow faster than every polynomial regardless of how big d is.

Asymptotic Order of Growth Hierarchy

$g(n)$

$f \in O(g)$

$f(n)$

n^n

$n!$

3^n

2^n

$n^3, 2n^3, (n-1000)^3, n^3 - n^2$

$n^2, 2n^2, -1000n^2 + n, 100n^2 + \log n$

$n \log n$

$n, 2n, 1000n, 10^{10}n + 1000$

$\sqrt{n}$

$\log^3 n$

$\log^2 n$

$\log n, \log_2 n, \log_3 n, \log_{10000} n$

$\log \log n$

$1, 10, 1000, C$

$f(n)$

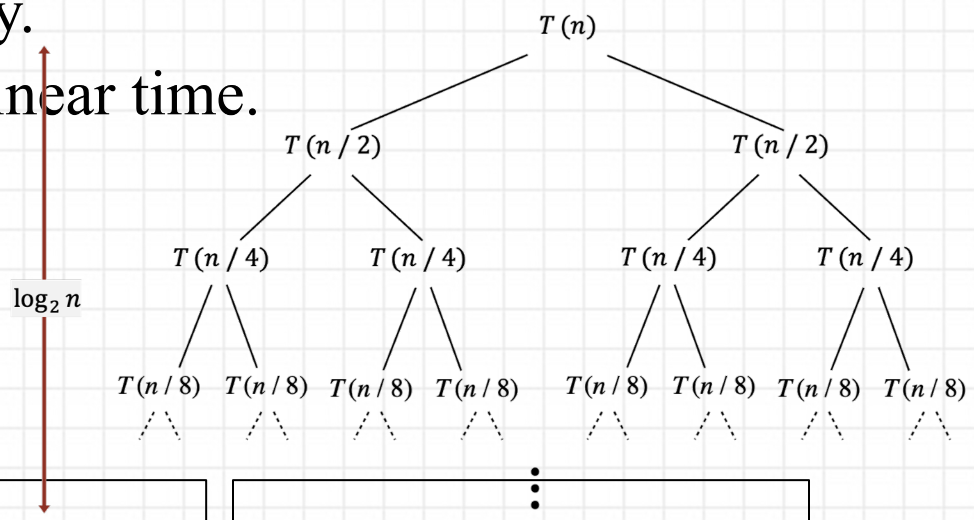
$f \in \Omega(g)$

$g(n)$

$f(n) \quad f \in \Theta(g) \quad g(n)$

Exam 1: Divide-and-Conquer (D&C)

- Main steps
  - Divide up problems into several subproblems (of the same type).
  - Solve (conquer) each subproblem (usually recursively).
  - Combine the solutions.
- Most common framework
  - Divide the problem of size n into two subproblems of size $n/2$ in linear time
  - Solve (conquer) the two subproblems recursively.
  - Combine two solutions into overall solution in linear time.



Exam 1: Divide-and-Conquer (D&C)

- Discussed examples:
  - **Binary-search**
    → Variant/applications of binary search
  - **Merge-sort**
    → Variant/applications of merge-sort
  - **Quick-sort**
    → Variant/applications of quick-sort
  - **Matrix multiplication**
  - **Closest pair of points**

Search Algorithm

Sorting Algorithm

Sorting Algorithm

Type of questions:
- Variant (Design) /applications /parts of binary search, merge-sort, or quick-sort
- True/False questions
- Worst case/best case
- Time and space complexity
- Complete the given incomplete solution

Exam 1: Master Theorem

- Goal. Recipe for solving common divide-and-conquer recurrences,

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

$$\boxed{T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right), & \text{if } a > b^d \quad \text{(case 1)} \\ \Theta\left(n^d \log n\right), & \text{if } a = b^d \quad \text{(case 2)} \\ \Theta\left(n^d\right), & \text{if } a < b^d \quad \text{(case 3)} \end{cases}}$$

- Limitation. Master theorem cannot be used if
  - $T(n)$ is not monotone, e.g., $T(n) = \sin(n)$
  - $f(n)$ is not polynomial, e.g., $T(n) = 2\,T\left(\frac{n}{2}\right) + 2^n$
  - b cannot be expressed as a constant, e.g., $T(n) = a\,T(\sqrt{n}) + f(n)$

Application of Master Theorem

- The recurrence relation is given → direct
- Dominated by root/leaves/evenly distributed
- An algorithm (D&C) is given, you need to find the recurrence first. Then, apply the Master Theorem → indirect

Exam 1: Dynamic Programming (DP)

- Dynamic Programming vs. Divide-and-Conquer

Divide-and-Conquer:

- Divide problem into subproblems
- Recursively solve the subproblems and aggregate solutions

Dynamic Programming

- Divide problem into subproblems, recursively solve them
- Subproblems overlap
- When a subproblem has been solved, remember its solution and reuse that solution rather than resolving it later (memoization)

Dynamic Programming

- Top-down vs. Bottom-up Approach
  - "Top-down" dynamic programming
    - Begin with problem description
    - i.e., begin at root of tree and work downwards
    - Recursively subdivide problem into subproblems
  - "Bottom-up" dynamic programming
    - Start at the leaf nodes of tree, i.e., the base case(s).
    - Build up solution to larger problem from solutions of the simpler subproblems

DP Examples

- One-dimensional
  1. Fibonacci sequence
  2. Staircase climbing
  3. Rod-cutting
  4. Red-black game

Type of questions:
- Design a DP algorithm
- Discuss the optimal substructure
- Write the recurrence relation/base case
- Top-down / bottom-up
- Time and space complexity

- Two-dimensional
  5. Longest common subsequence (LCS)
  6. Coin-changing
  7. Knapsack

Type of questions:
- Discuss the optimal substructure
- Recurrence given
- Solving part of the problem
- Time and space complexity

Exam 1: Practice Problems

Course website

CS-3510 | Algorithms — home policies lectures assignments resources

	[pdf | tex | solution]		
3	**hw3:** [pdf | tex | solution]	06/03	06/10
4	**hw4:** [pdf | tex | solution]	06/10	06/17
5	**hw5:** [pdf | tex | solution]	06/17	07/08
6	**hw6:** [pdf | tex | solution]	07/08	07/15

You can use this LaTeX template file to prepare your solutions on the cloud-based LaTeX editor OverLeaf.

#	Exam	Date (mm/dd)	Time (EST)	Location
1	**Exam 1:** Complexity, Devide-and-Conquer, Dynamic Programming [practice | pdf | solution]	06/09 Thursday	03:30 pm	Klaus 2443
2	**Exam 2:** Graph Algorithms [pdf | solution]	07/07 Thursday	03:30 pm	Klaus 2443
3	**Final Exam:** Inclusive (including all discussed topics) [practice | solution]	07/28 Thursday	03:00 pm	Klaus 2443