COMP 8547
Advanced Computing Concepts

Course instructor: Dr. Olena Syrotkina

Winter 2024: 05 – Advanced Design and Analysis

Chapter 5 – Advanced Design and Analysis

University of Windsor

Contents

- Recursion
- Divide and conquer
- Recurrences
- Master method
- Dynamic programming
- Longest common subsequence
- Edit distance
- Applications
  - Spellchecking
  - Document processing

Recursion

- Def.: A method, function or procedure that makes calls to itself
- Main components of recursive function:
  - Base case(s): Does not include recursion
  - Recur: Calls to itself. Must always lead to a base case
- Recursive algorithms:
  - Apply to problems that are defined recursively
  - Some problems are easily defined recursively
  - and hence algorithms easier to design
  - But are usually (very) inefficient
  - Complexity of recursive algorithms is done via recurrence equations

- Example: factorial of a number n
  - Definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{if } n \geq 1 \end{cases}$$

  - Algorithm:

```
Algorithm factorial(n)
    if n = 0 then
        return 1
    else
    return n * factorial(n – 1)
```

Example – Fibonacci numbers

Fibonacci numbers

0	1	1	2	3	5	8	13	…
0	1	2	3	4	5	6	7	8

Defined recursively:

$F_0 = 0$
$F_1 = 1$
$F_i = F_{i-1} + F_{i-2}$ for $i > 1$

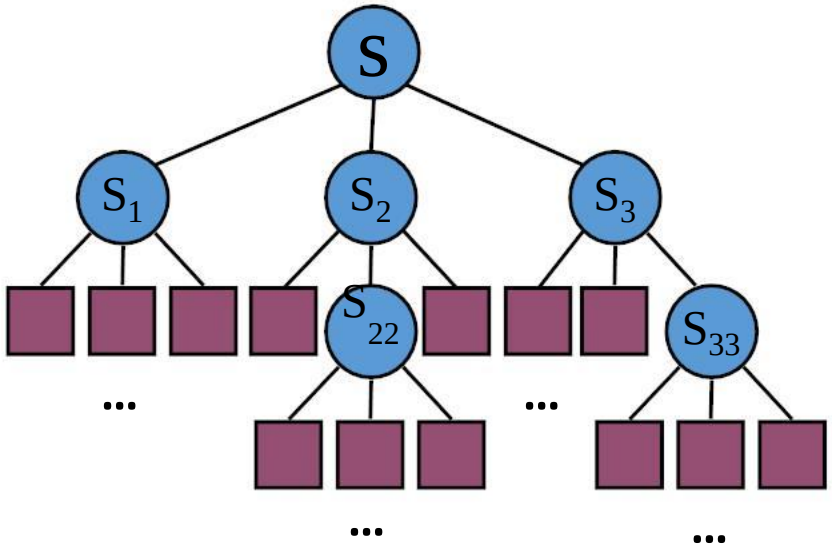
Recursive algorithm:

```
Algorithm rF(k):
    Input: Nonnegative integer k
    Output: k_th Fibonacci number F_k
    if k = 0 or k = 1 then
        return k
    else
        return rF(k - 1) + rF(k - 2)
```

Running time of recursive algorithm is exponential!

Divide and Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the problem or input data S in two or more disjoint sub-problems S_1, S_2, …
  - Recur: solve sub-problems S_1, S_2, … recursively
  - Conquer: combine the solutions for S_1, S_2, …, into a solution for S
- The base case for the recursion are sub-problems of constant size
- Analysis can be done using **recurrence equations**



Merge sort – review

- Merge-sort on an input sequence S with n elements consists of three steps:
  - Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
  - Recur: recursively sort S_1 and S_2
  - Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm mergeSort(S, C)
Input: sequence S with n objects and comparator C
Output: sequence S sorted according to C
if S.size() > 1
  (S_1, S_2) ← partition(S, n/2)
  mergeSort(S_1, C)
  mergeSort(S_2, C)
  S ← merge(S_1, S_2)

Recurrence equation analysis

- The conquer step of merge-sort consists of merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes at most cn steps, for some constant c
- Likewise, the base case ($n < 2$) will take at most c steps
- Therefore, if we let $T(n)$ denote the running time of merge-sort:

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ 2T(n/2) + cn & \text{if } n \geq 2 \end{cases}$$

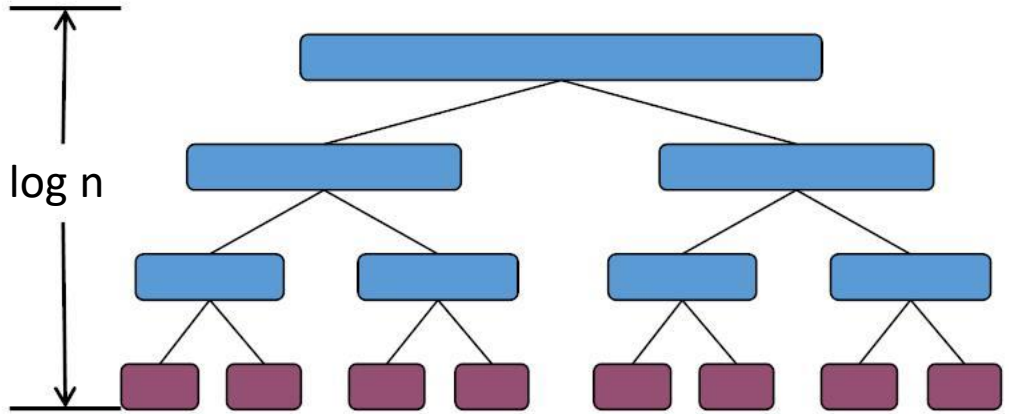
- T(.) is on the right hand side
- Need to find a closed-form expression for T(n)

Solving recurrences - The recursion tree

- Draw the tree of recursive calls and count the number of operations needed for each depth
- Total time is the sum of costs for all depths

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ 2T(n/2) + cn & \text{if } n \geq 2 \end{cases}$$

depth	T's	size	time
0	1	n	cn
1	2	$n/2$	cn
i	2^i	$n/2^i$	cn
…	…	…	…



Total time: T(n) = $cn + cn \log n$

(cost for leaves plus all previous depths)

The Master Method – optional (not in exam)

- The Master method provides a "recipe" for solving recurrences
- With some exceptions, most recurrences can be solved using the Master method
- It mostly applies to recurrences derived from divide-and-conquer algorithms
- The proof of the Master theorem is rather involved, and can be found in [4]

The Master theorem

Given a recurrence of the form:

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{if } n \geq d \end{cases}$$

Then:

1. if $f(n)$ is $O(n^{\log_b a - \varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a + \varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \leq \delta f(n)$ for some $\delta < 1$.

where $\varepsilon > 0$

Meaning of the Master Theorem

It compares two functions:

$$f(n) \text{ and } \Theta(n^{\log_b a})$$

Case 1: $\Theta(n^{\log_b a})$ is the largest function and then $T(n) = \Theta(n^{\log_b a})$

Case 3: $f(n)$ is the largest function, then $T(n) = \Theta(f(n))$

Case 2: Both functions have the "same complexity",

i.e., $\Theta(n^{\log_b a}) = \Theta(f(n))$ and then

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$

Example 1 – Binary search

Master method:

1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

Running time of binarySearch:

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ T(n/2) + c & \text{if } n \ge 2 \end{cases}$$

Solution:

$\log_b a = \log_2 1 = 0$ and $k = 0$

$$f(n) = c = \Theta(n^0 \log^0 n) = \Theta(1)$$

Case 2 applies:

$$T(n) = \Theta(\log n)$$

Algorithm binarySearch(S, *k*, low, high)
if low > high **then**
    return null
else
    mid ← ⌊(low + high) / 2⌋
    e ← S.get(mid)
    if *k* = *e*.getKey() **then**
        return *e*
    else if *k* < *e*.getKey() **then**
        return binarySearch(S, *k*, low, mid-1)
    else
        return binarySearch(S, *k*, mid+1, high)

Example 2 – Merge sort

Master method:

1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

Algorithm mergeSort(S, C)

Input: sequence S with n objects and comparator C

Output: sequence S sorted according to comparator C

if S.size() > 1

  (S_1, S_2) ← partition(S, n/2)

  mergeSort(S_1, C)

  mergeSort(S_2, C)

  S ← merge(S_1, S_2)

Running time of mergeSort:

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ 2T(n/2) + cn & \text{if } n \ge 2 \end{cases}$$

Solution:

$\log_b a = \log_2 2 = 1$ and $k = 0$

$f(n) = cn = \Theta(n^1 \log^0 n) = \Theta(n)$

Case 2 applies:

$T(n) = \Theta(n \log n)$

Example 3

Master method:

1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
   provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

Running time:

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ 4T(n/2) + cn & \text{if } n \ge 2 \end{cases}$$

Solution:

$\log_b a = \log_2 4 = 2$

$f(n) = cn = \Theta(n^{2-1}) = \Theta(n)$

Case 1 applies:

$T(n) = \Theta(n^2)$

Example 4

Master method:

1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$,
   provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

Running time:

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ T(n/3) + cn & \text{if } n \ge 2 \end{cases}$$

Solution:

$\log_b a = \log_3 1 = 0$

$f(n) = cn = \Omega(n^{0+1}) = \Omega(n)$

Case 3 applies:

$T(n) = \Theta(n)$

Example 5 – Heap bottom-up construction

Master method:

1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b) \le \delta f(n)$ for some $\delta < 1$.

Running time of BottomUpHeap:

Downheap(H, root) takes $c \log n$

$$T(n) = \begin{cases} c & \text{if } n < 2 \\ 2T(n/2) + c\log n & \text{if } n \ge 2 \end{cases}$$

Solution:

$\log_b a = \log_2 2 = 1$

$f(n) = c\log n = O(n^{1-0.5}) = O(n^{0.5})$

Case 1 applies:

$T(n) = \Theta(n)$

Algorithm BottomUpHeap(*S*)

Input: Seq. *S* with $n = 2^h$-1 keys

Output: A heap *H*

if *S* is empty **then**

return empty heap

$k \leftarrow S[0]$

Split $S[1..n\text{-}1]$ into S_1 and S_2

$H_1 \leftarrow$ BottomUpHeap(S_1)

$H_2 \leftarrow$ BottomUpHeap(S_2)

Create *H* with *k* as the root, H_1 left child and H_2 as right child

Downheap(H, root)

return *H*

Example 6 – Matrix multiplication

Master method:

1. if $f(n)$ is $O(n^{\log_b a-\varepsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. if $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
3. if $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$, then $T(n)$ is $\Theta(f(n))$, provided $af(n/b)\le \delta f(n)$ for some $\delta<1$.

Running time of SMM:

$$T(n)=\begin{cases} c & \text{if } n<2 \\ 7T(n/2)+cn^2 & \text{if } n\ge 2 \end{cases}$$

Solution:

$\log_b a = 2.81$

$f(n)=cn^2=O(n^{2.81-0.81})=O(n^2)$

Case 1 applies:

$T(n)=\Theta(n^{2.81})$

Strassen's matrix multiplication algorithm

Algorithm SMM(A,B)

Input: Two square matrices A and B

Output: C = A B

if $n = 1$

  return $A \times B$

else

$M_1 = \text{SMM}(A_{1,2}-A_{2,2}, B_{2,1}+B_{2,2})$

$M_2 = \text{SMM}(A_{1,1}+A_{2,2}, B_{2,1}+B_{2,2})$

$M_3 = \text{SMM}(A_{1,1}-A_{2,1}, B_{1,1}+B_{1,2})$

$M_4 = \text{SMM}(A_{1,1}+A_{1,2}, B_{2,2})$

$M_5 = \text{SMM}(A_{1,1}, B_{1,2}-B_{2,2})$

$M_6 = \text{SMM}(A_{2,2}, B_{2,1}-B_{1,1})$

$M_7 = \text{SMM}(A_{2,1}+A_{2,2}, B_{1,1})$

$C_{1,1} = M_1+M_2-M_4+M_6$

$C_{1,2} = M_4+M_5$

$C_{2,1} = M_6+M_7$

$C_{2,2} = M_2-M_3+M_5-M_7$

return C

Dynamic programming

- Applies to a problem that seems to require a lot of time (possibly exponential), provided we have:
  - **Simple subproblems:** the subproblems can be defined in terms of a few variables, such as j, k, l, m, and so on.
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap
  - **Bottom-up construction**: Smaller sub-problems solved first and solutions stored in a table for use by larger sub-problems

Example:
Fibonacci numbers

0	1	1	2	3	5	8	13	…
0	1	2	3	4	5	6	7	8

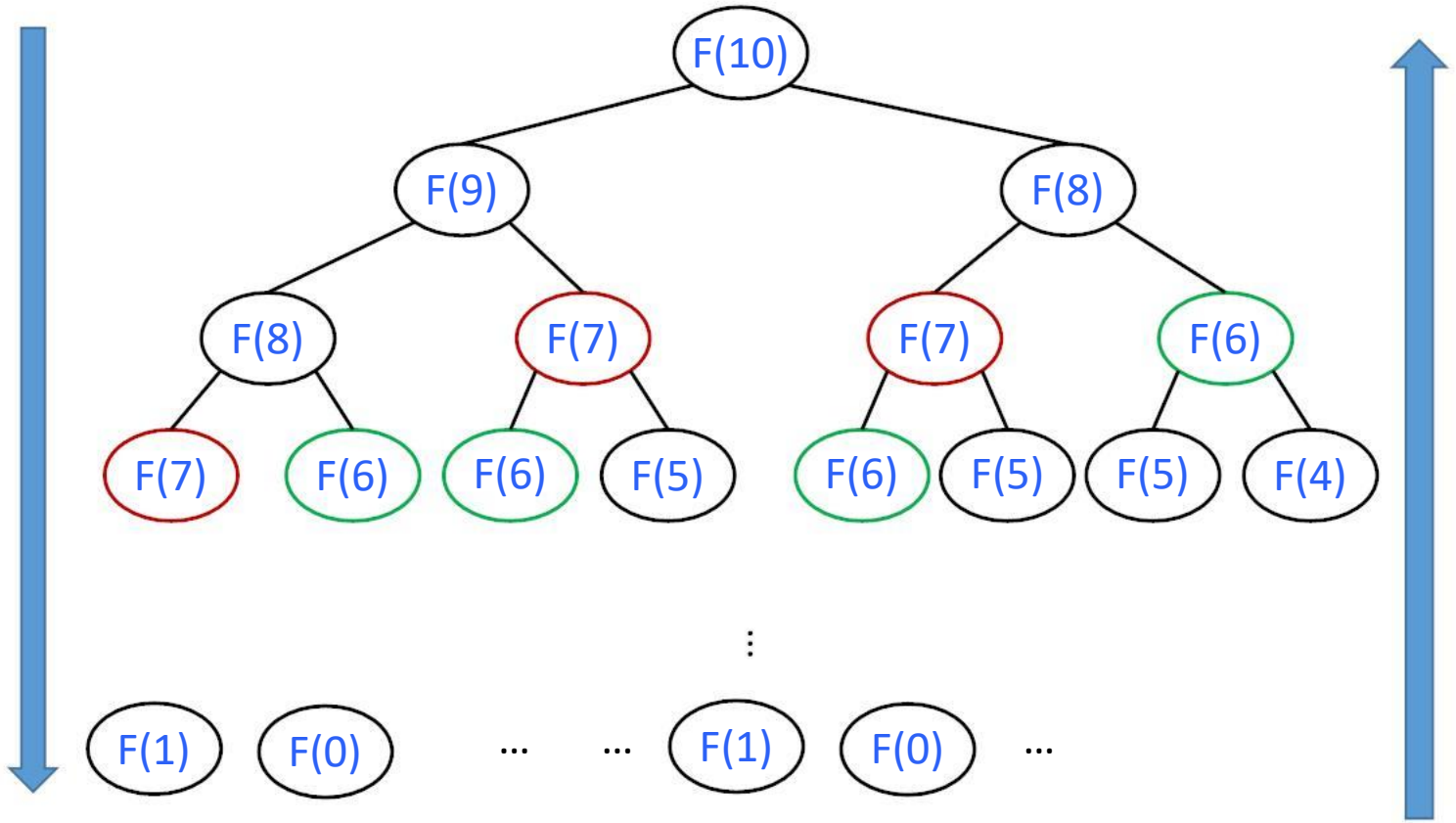
Dynamic programming:
- Find Fibonacci numbers in increasing order
- Store solutions to small sub-problems in a table A

Algorithm DPFibonacci(k):
Input: A nonnegative integer k
Output: Fibonacci number for k
A[0] ← 0
A[1] ← 1
for i ← 2 **to** k **do**
A[i] ← A[i – 1] + A[i – 2]
return A[k]

- Runs in $O(k)$ time… i.e., linear time…
- while recursive algorithm runs in exponential time!

Fibonacci numbers – recursion vs dynamic programming

Recursive:
Top-down



DP:
Bottom-up

Longest common subsequence – LCS

- Given two strings X and Y, the longest common subsequence (LCS) problem is to find the longest subsequence(s) common to both X and Y
- Subsequence is a substring of characters not necessarily contiguous
- Generalizations of this problem have applications in text processing and bioinformatics:
  - DNA or protein sequence alignment
  - Find index terms in web pages
- Example:
  - X = AGCCTAGT and Y = GACTCAT
  - More than one solution:
  - E.g., 1) ACTAT 2) GCTAT

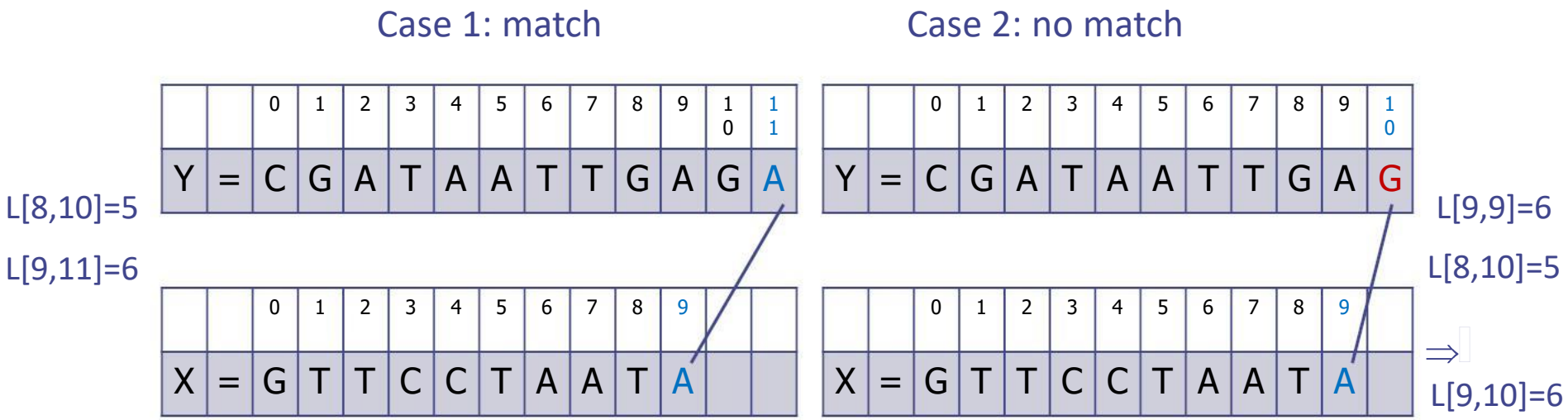
Solution formulated as:

- Define L[i,j] to be the length of the longest common subsequence of X[0..i] and Y[0..j]
- Let -1 be an index, so L[-1,k] = 0 and L[k,-1]=0, to indicate that the null part of X or Y has no match with the other
- Then we can define L[i,j] in the general case as follows:
  1. If $x_i=y_j$, then L[i,j] = L[i-1,j-1] + 1 (we can add this match)
  2. If $x_i \neq y_j$, then L[i,j] = max{L[i-1,j], L[i,j-1]} (we have no match here)
- Recursive solution is trivial…
  - But algorithm runs in exponential time!

Longest common subsequence – LCS

When we define L[i,j], we have two cases:

1. If $x_i=y_j$, then L[i,j] = L[i-1,j-1] + 1 (we can add this match)
2. If $x_i \neq y_j$, then L[i,j] = max{L[i-1,j], L[i,j-1]} (we have no match here)



LCS – dynamic programming solution

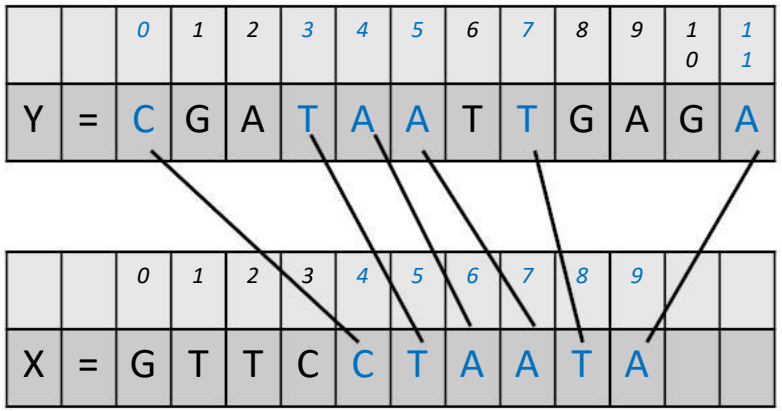
L			C	G	A	T	A	A	T	T	G	A	G	A
		-1	0	1	2	3	4	5	6	7	8	9	10	11
	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	1	1	1	1	1	1	1	1	1	1	1
T	1	0	0	1	1	2	2	2	2	2	2	2	2	2
T	2	0	0	1	1	2	2	2	3	3	3	3	3	3
C	3	0	1	1	1	2	2	2	3	3	3	3	3	3
C	4	0	1	1	1	2	2	2	3	3	3	3	3	3
T	5	0	1	1	1	2	2	2	3	4	4	4	4	4
A	6	0	1	1	2	2	3	3	3	4	4	5	5	5
A	7	0	1	1	2	2	3	4	4	4	4	5	5	6
T	8	0	1	1	2	3	3	4	5	5	5	5	5	6
A	9	0	1	1	2	3	4	4	5	5	5	6	6	6

- Solutions can be found by tracing back the table from L(m,n)
- More than one solution can be found by using backtracking
- One of these solutions is in the path colored red

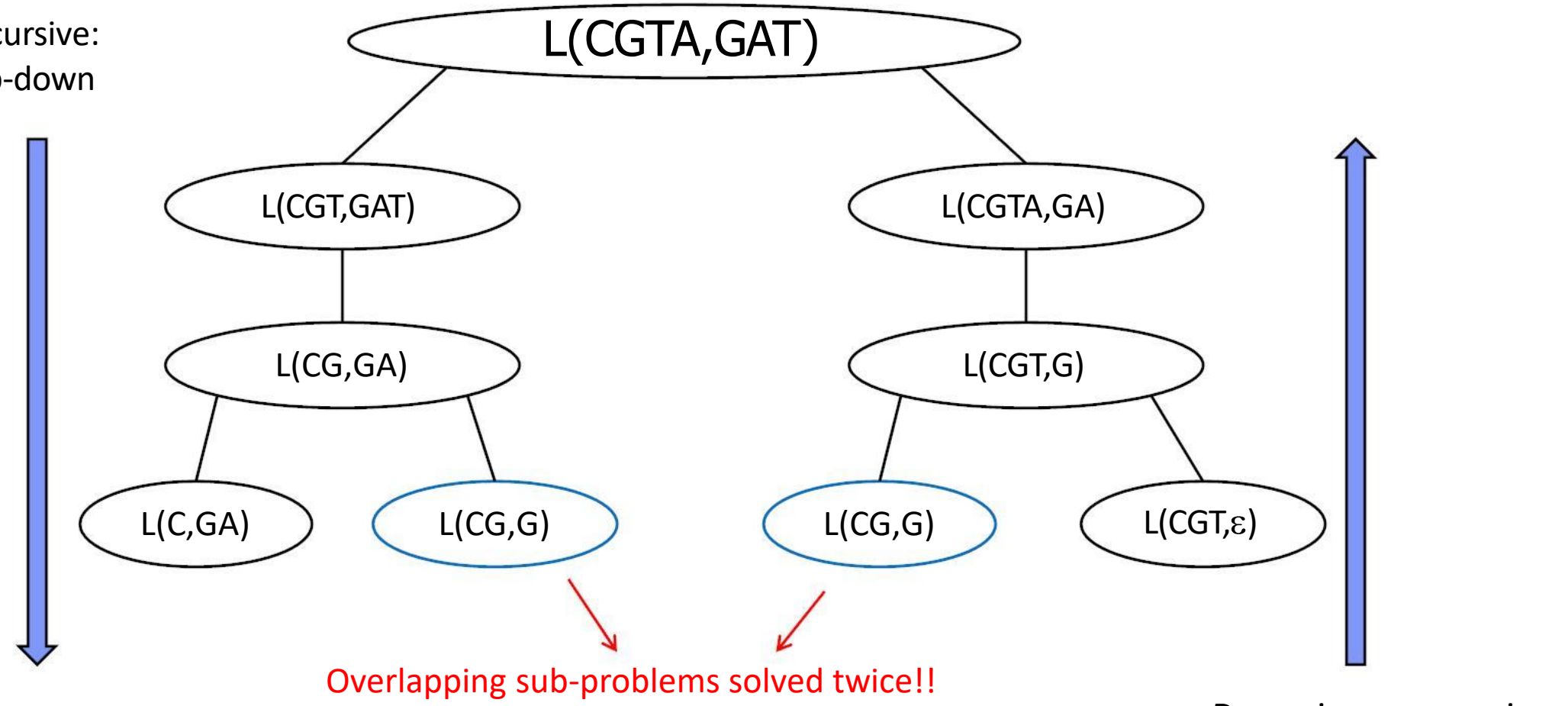
Algorithm LCS(*X*,*Y*)
Input: Strings *X* and *Y* of length *n* and *m* respectively
Output: Array L containing the lengths of all sub-problems

for *i* =-1 **to** *n*-1 **do**
  L[*i*,-1] ← 0
for *j* =0 **to** *m*-1 **do**
  L[-1,*j*] ← 0
for *i* =0 **to** *n*-1 **do**
  for *j*=0 **to** *m*-1 **do**
    if $x_i = y_j$ **then**
      L[*i*, *j*] ← *L*[*i*-1, *j*-1] + 1 *// match*
    else
      L[*i*, *j*] ← max{*L*[*i*-1, *j*] , *L*[*i*, *j*-1]} *//no match*
return array *L*

Running time: $O(mn)$
If both sequences have same length n: $O(n^2)$



LCS: Recursive vs Dynamic Programming



$L(X_i, Y_j) \equiv \text{RecursiveLSC}(X_i, Y_j)$

Dynamic programming vs Recursion

University of Windsor

Dynamic Programming	Recursion
Bottom-up: • Solve small sub-problems first • Use these solutions to solve larger sub-problems, and so on	Top-down strategy: • Attempt to solve the whole problem • Call for solutions to smaller sub-problems, • which will call for solutions to even smaller sub-sub-problems, and so on
Each sub-problem is solved once	Each sub-problem may be solved many times
Solution is difficult to be formulated	Solution is rather easy and "natural" to the problem
Polynomial-time complexity of algorithm	Algorithm may run in exponential time

Application 1 – Document Comparison

- Documents can be compared to find similarities
- Examples:
  - Finding that two programs are the same or similar
  - Given a document (text or PDF), find similar ones on the Web
  - Comparing two assignments to see how similar they are
- Concrete example:
  - What is the difference between Programs 1 and 2?
  - How do we compare these two?

Program 1:

```
public class Maximum {
   public static void main(String[] args) {
      int currentMax = …;
      …
   }
}
```

Program 2:

```
/* my program */
public class Maximum {
   /* Calculate the maximum */
   public static void main(String[] args) {
      int currentMax = …;
      …
   }
}
```

Edit distance

- Can be seen as a generalization of LCS
- Given two strings *X* and *Y*
- Consider *X* as the initial string and Y as the target string
- Find the minimum number of changes to transform *X* into *Y*
- Consider three operations:
  - Substitution: A character in *X* is replaced by the corresponding character in *Y*
  - Insertion: A character in *Y* is inserted into *X*
  - Deletion: A character in *X* is deleted
  - It uses a matrix of costs, or "distances", and $\delta(X_i, Y_j)$, which is 1 if $X_i = Y_j$, and 0 otherwise

- **Example:**

  X = *excused* is transformed into

  Y = *exhausted*

- Steps:
  - First, substitute *h* for *c*, yielding
    X = *exhused*
  - Second, insert *a*, obtaining
    X = *exhaused*
  - Third, insert *t*, obtaining
    X = *exhausted*
  - Since the cost of each operation is 1, the distance between these two strings is 3

Edit distance

- Can be solved using dynamic programming
- Algorithm EditDistance runs in O(*nm*)
- Function δ can be generalized to:
  - Other than the 0-1 cost function
  - Can consider different costs for different symbols

C		e	x	h	a	u	s	t	e	d
0		*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
e	*1*	0	1	2	3	4	5	6	7	8
x	*2*	1	0	1	2	3	4	5	6	7
c	*3*	2	1	1	2	3	4	5	6	7
u	*4*	3	2	2	2	2	3	4	5	6
s	*5*	4	3	3	3	3	2	3	4	5
e	*6*	5	4	4	4	4	3	3	3	4
d	*7*	6	5	5	5	5	4	4	4	3

Algorithm *EditDistance*(*X*,*Y*)

Input: Strings *X* and *Y* of length *n* and *m* respectively

Output: Array C containing prefix edit distances

$C[0,0] \leftarrow 0$

for $i \leftarrow 0$ **to** *m* // length of X

  $C[i,0] \leftarrow i$

for $i = 0$ to *n* // length of X

  $C[0,j] \leftarrow j$

for $i = 1$ **to** *m*

  for $j = 1$ **to** *n*

    $C[i,j] = \min\{C[i\text{-}1,j]+1, C[i, j\text{-}1]+1,$

    $C[i\text{-}1,j\text{-}1]+1 - \delta(X_i,Y_j)\}$

return $C[m,n]$

Applications:

- DNA/protein sequence alignment
- Spellchecking
- Document comparison

Dynamic programming – problems

- Several problems can be solved using dynamic programming
- Some examples are:
  - Longest common subsequence
  - Edit distance
  - Bio-sequence alignment
    - Global and local
    - Pairwise and multiple
  - All-pairs shortest path in a graph
  - Traveling salesman problem
  - Matrix chain multiplication
  - Optimal binary search trees
  - 0-1 knapsack problem
  - Assembly-line scheduling
  - Multi-level thresholding
  - Learning parameters of a hidden Markov model
    - Applications to speech recognition and multiple bio-sequence alignment

Edit Distance – Application 2: Spellchecking

- Consider a dictionary of n words:
  $D = \{w_1, w_2, \ldots, w_n\}$
- Let w be a word typed by the user

Spellchecking:

- Correct spelling:
  - There is at least one word w_i in D such that $w_i = w$
- Incorrect spelling:
  - Find the edit distance between w and all words in D
  - Rank all the words in D by distance
  - List the top k words as suggestions
  - Complexity?

Example:

- D = English dictionary: 171,476 words
- D = {a, ab, …, horse, …, hose, ..., hot, …, house, … , zyzzyva}
- User types: w = hoyse
- Suggestions:

Top k words	Distance
house	1
hose	1
horse	1
…	…
hope	2
…	…

Review and Further Reading

- Recurrences – Master method:
  - Sec. 11.1 of [1], Ch. 4 of [4]
- Recursion
  - Ch. 5 of [2]
- Divide and conquer
  - Ch. 11 of [1], Sec. 10.2 of [3], Ch. 5 of [5]
- Dynamic programming
  - Ch. 11 of [1], Sec. 13.4 of [2], Sec. 10.3 of [3], Ch. 15 of [4], Ch. 6 of [5]

References

1. Algorithm Design and Applications by M. Goodrich and R. Tamassia, Wiley, 2015.
2. Data Structures and Algorithms in Java, 6th Edition, by M. Goodrich and R. Tamassia, Wiley, 2014.
3. Data Structures and Algorithm Analysis in Java, 3rd Edition, by M. Weiss, Addison-Wesley, 2012.
4. Introduction to Algorithms, 3rd Edition, by T. Cormen et al., McGraw-Hill, 2009.
5. Algorithm Design by J. Kleinberg and E. Tardos, Addison-Wesley, 2006.

Lab – Practice

1. Use Edit Distance (class Sequences.java) implementation provided in the source code.
   a) Generate 1,000 pairs of random words with lengths 10, 20, 50 and 100.
   b) Compute the edit distance for all words and record the average CPU time for each pair.

2. Use Edit Distance and web page provided.
   a) Write a program that takes an index term (a word, e.g. Airbus) as well as a web page (.txt) as input and finds out the 10 most similar words in the file (smallest distance between the index term and a word in the file).

Exercises

1. Briefly explain the meaning of the Master theorem. *How would you use iterative substitution in a general recurrence T(n) = aT(n/b) + f(n)?
2. Write an algorithm that merges three sorted lists of size n/3 in O(n) time. Show that the algorithm runs in O(n) worst-case time. Can you do the same for 4 and 5 lists?
3. Create new sorting algorithms on the basis of Merge sort. Design 4 algorithms for dividing the list into 2, 3, 4 and 5 sub-lists. Assume that merge takes linear time for all cases. Find the worst-case running times for the four algorithms using the Master theorem.
4. Do the same as #9 with binary search.
5. Suppose that Merge sort divides the list into two lists of size 1/4n and 3/4n. What is the worst case running time of this algorithm? Use the Master method. Compare the running time of this algorithm with that of the standard Mergesort.
6. Give six examples of recurrences that can be solved using the Master method (two examples for each case). When possible, find existing algorithms whose running time corresponds to those recurrences.
7. Find the asymptotic notation for $T(n) = 3T(n/4)+n^2$
8. Find the asymptotic notation for $T(n) = 4T(2n/3)+\log^2 n$
9. Find the asymptotic notation for $T(n) = T(n/4)+\log n$
10. Write a recursive algorithm for the LCS problem.
11. Write a recursive algorithm that finds the edit distance between two words. Implement the algorithm in Java. Run this algorithm on several words and compare its performance with the dynamic programming algorithm.
1. Using the Master theorem, solve $T(n) = 8T(n/2) + O(n^2)$
12. Find the LCS between “transposition” and “transponder”. What is the edit distance between these two words?
13. For the first example seen in class, how many different LCSs are there? Show four of them.
14. Repeat the same for the second example.
15. For the Fibonacci of 10, how many recursive calls will be made?
16. Mention the key differences between dynamic programming and recursion.