COMP20007 Workshop Week 9

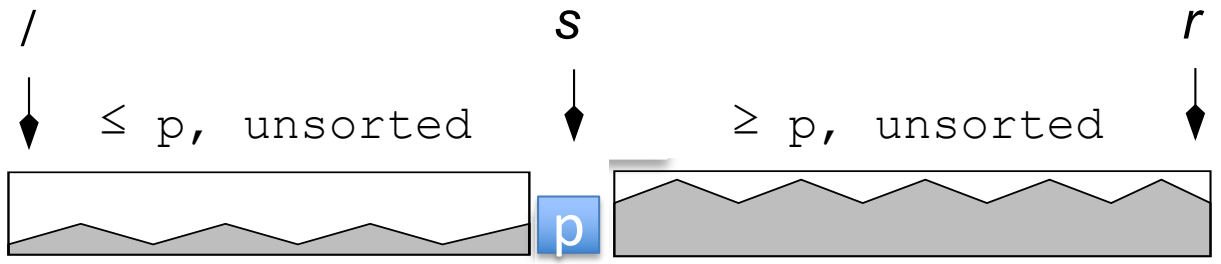
1	sorting algorithms & properties, focusing on • quicksort: Lomuto's & Hoare's Partitioning • mergesort ?
2	Dynamic Programming: • Problem 9.2 (Longest Common Substring) • DP: review
LAB	sorting & timing: probably as homework
Events	• A2: week 10+11, due week 12?

Review: Sorting algorithms for A[0..n-1]

	Selection Sort	Insertion Sort	Quick Sort	Merge
Basic Idea	for (i=0; i<n-1; i++): • find the smallest of A[i..n-1] • swap with A[i] • hence done with A[0..i]	for (i=1; i<n; i++) : • insert A[i] to the *sorted* A[0..i-1] to make A[0..i] sorted	Do recursively: • Choose a pivot P, partition array into a *lesser* and a *greater* (than P) halves, with P in between them	Do recursively • Split A into 2 equal sub-arrays • Sort them separately • Merge sorted sub-arrays
Complexity	$\theta(n^2)$	$O(n^2)$	$\theta/O(\ )$	$\theta/O(\ )$
Best case	$\theta(\ )$	$\theta(\ )$	$\theta(\ )$	$\theta(\ )$
Worst case	$\theta(\ )$	$\theta(\ )$	$\theta(\ )$	$\theta(\ )$
Average	$\theta(\ )$	$\theta(\ )$	$\theta(\ )$	$\theta(\ )$
Input-sensitive?				
In-place?				
Stable?				

Quicksort idea (recursive, usage: `Quicksort(A[0..n-1])`)

```
function QUICKSORT(A[l..r])
    if l < r then
        s ← PARTITION(A[l..r])
        QUICKSORT(A[l..s − 1])
        QUICKSORT(A[s + 1..r])
```



p = A[s] is called the ***pivot*** of this partitioning

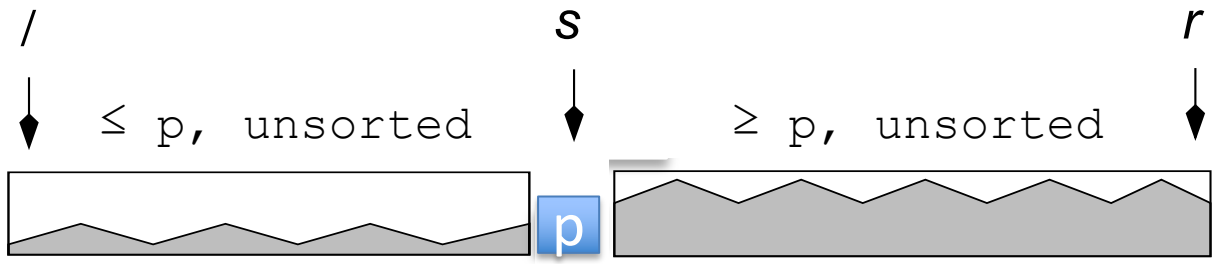
Note: a **PARTITION** of n elements has the complexity of $\theta(n)$

Questions:

- What is the (additional) space complexity of **QUICKSORT**?
- What is the time complexity?
- Is it input-sensitive?
- Is it in-place?
- Is it stable?

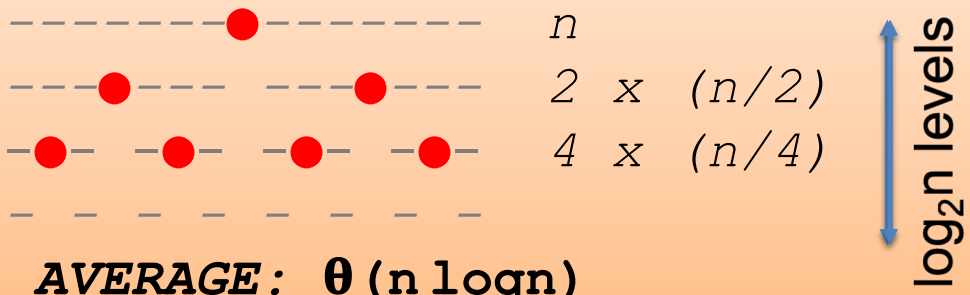
Quicksort Properties - Check your answers

```
function QUICKSORT(A[l..r])
    if l < r then
        s ← PARTITION(A[l..r])
        QUICKSORT(A[l..s − 1])
```



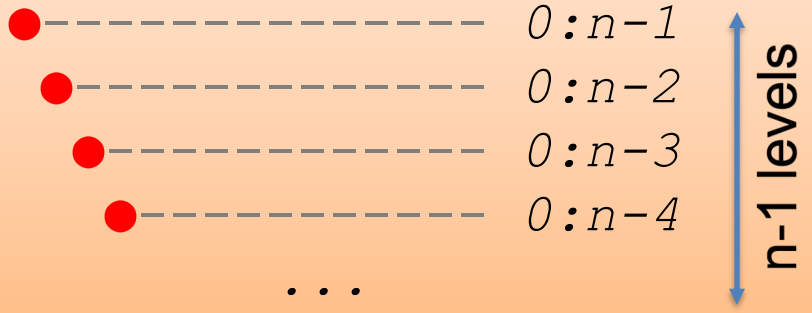
QUICKSORT complexity depends on the relative lengths of the left and the right parts in partitioning.

BEST: always balanced: $\theta(n \log n)$



AVERAGE: $\theta(n \log n)$

WORST: one half always empty: $\theta(n^2)$



What is the (additional) space complexity of **QUICKSORT**? for ecursion:

BEST/AVERAGE O(log n) WORST O(n)

- Is it input-sensitive? Y
- Is it in-place? Y
- Is it stable? N – but we need to check with Partitioning

Lomuto's Partitioning for A[l..r]: examining from the left

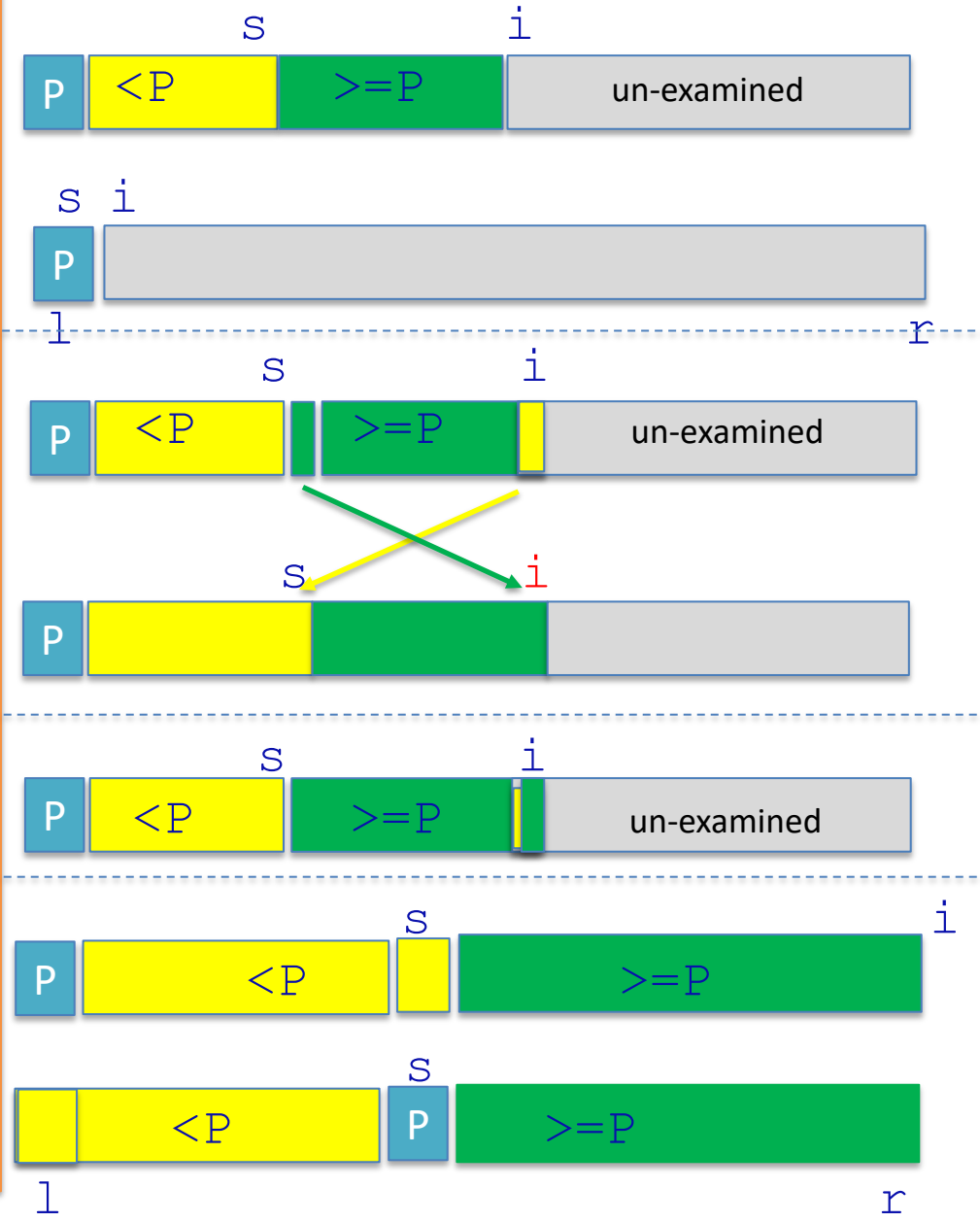
Pay attention on the loop invariant

```
# loop invariant:
l          s              i              r
PA[l+1..s]<P A[s+1..i-1]≥P A[i..r]un-examined

function LomutoPartition(A[l..r])
  P ← A[l]              #loop init
  s ← l,  i ← l+1
  for i ← l+1 to r do
    if A[i] < P then
      # it belongs to the left, the yellow area
      # ➔ swap it with the first green
      s ← s + 1
      Swap(A[s], A[i])
    else do
      # do nothing if A[i] >=P because
      # A[i] just joins the green area

  # at loop exit (i=r+1)
  # A[s] is the last yellow
  Swap(A[l], A[s])
  return s
```

```
function LomutoPartition(A[l..r])
  p ← A[l]
  s ← l
  for i ← l+1 to r do
    if A[i] < p then
      s ← s+1
      Swap(A[s], A[i])
  Swap(A[l], A[s])
  return s
```



Example: Run quicksort with Lomuto's for [2 4 1 3]

```
function LomutoPartition(A[l..r])
  P ← A[l]
  s ← l

  for i ← l+1 to r do
    if A[i] < P then
      #extend the yellow area
      #  by swapping
      s ← s + 1
      Swap(A[s], A[i])

  #  at loop exit (i=r+1)
  #  A[s]  is the last yellow, swap it with pivot A[l]
  Swap(A[l], A[s])
  return s
```

Start with:

$2_s \quad 4_i \quad 1 \quad 3$

To simplify, we will write out the sequence only:

- at the beginning and end of the loop, and
- after a swap.

loop invariant:

l i j r

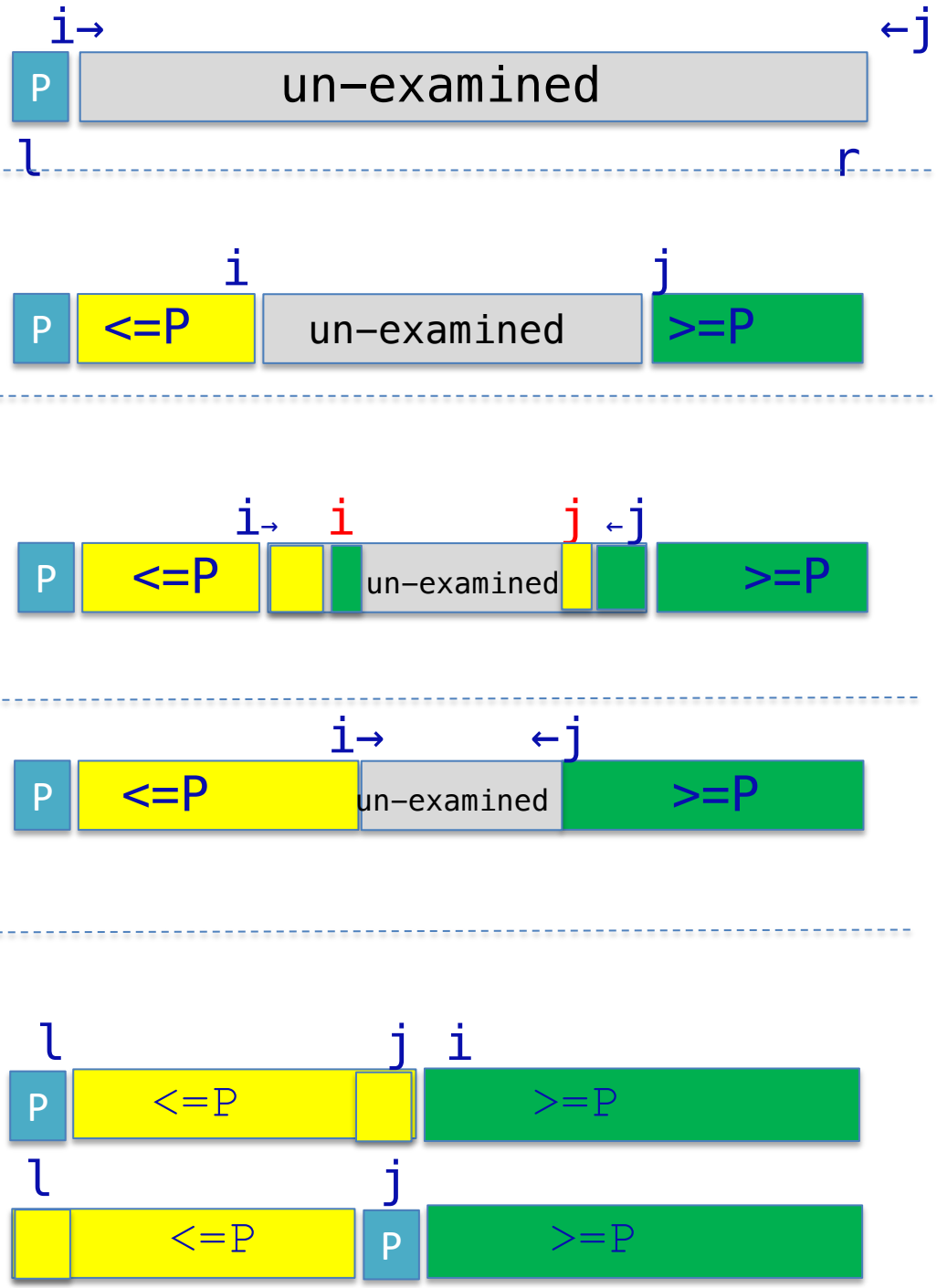
A[l+1..i]≤P A[i+1..j−1] un-examined A[j..r]≥P

```
function HoarePartition(A[l..r])
   i ← l; j ← r + 1
   P ← A[l]                          # loop init
   repeat
      # move i forward until A[i]>=P
      repeat i←i+1 until A[i] ≥ P
      # move j backward until A[j]<=P
      repeat j←j−1 until A[j] ≤ P

      # extend yellow and green area
      #    at the same time by swapping ----→
      if (i<j) then Swap(A[i], A[j])
until i ≥ j

# at loop's exit: i and j crossed
Swap(A[l], A[j])

return j
```



loop invariant:

```
l              i                                  j        r
 A[l+1..i]≤P  A[i+1..j-1]  un-examined  A[j..r]≥P
```

```
function HoarePartition(A[l..r])
  P ← A[l]                          # loop init
  i ← l; j ← r + 1

  repeat
     # move i forward until A[i]>=P
     repeat i←i+1 until A[i] ≥ P
     # move j backward until A[j]<=P
     repeat j←j-1 until A[j] ≤ P

     # extend yellow and green area
     #   at the same time by swapping ----→
    if (i<j) then Swap(A[i], A[j])
  until i ≥ j

   # at loop's exit: i and j crossed
  Swap(A[l], A[j])
return j
```

```
function HoarePartition(A[l..r])
    p ← A[l]
    i ← l; j ← r + 1
    repeat
        repeat i ← i + 1 until A[i] ≥ p
        repeat j ← j − 1 until A[j] ≤ p
        Swap(A[i], A[j])
    until i ≥ j
    Swap(A[i], A[j])
    Swap(A[l], A[j])
    return j
```

Example: Run quicksort with Hoare's for [2 4 1 3]

```
P ← A[l]                    # loop init
i ← l; j ← r + 1
# loop invariant:
l              i                              j        r
 A[l+1..i]≤P A[i+1..j−1] un-examined A[j..r]≥P

repeat
   # move i forward until A[i]>=P
   repeat i←i+1 until A[i] ≥ P
   # move j backward until A[j]<=P
   repeat j←j−1 until A[j] ≤ P

   # extend yellow and green area
   #   at the same time by swapping ----→
   if (i<j) then SWAP(A[i], A[j])
until i ≥ j

# at loop's exit: i and j crossed
SWAP(A[l], A[j])

return j
```

Start with:

$2_i \quad 4 \quad 1 \quad 3 \quad _j$

To simplify, we will write out the sequence only:

- at the beginning and end of the loop, and
- after a swap.

Question 9.1: Group Work

Know how to run by hand the following algorithms:

a) Selection Sort
b) Merge Sort
c) Quick Sort with Lomuto's Partitioning
d) Quick Sort with Hoare's Partitioning
e) Insertion Sort

For each algorithm: Run the algorithm on the array:

- [A N A L Y S I S]

Question 9.1

For each sorting algorithm:

i. Run the algorithm on the following input array:
[A N A L Y S I S]

ii. What is the time complexity of the algorithm?

iii. Is the sorting algorithm stable?

iv. Does the algorithm sort in-place?

v. Is the algorithm input sensitive?

- skip stuffs that, after a short discussion, your group agrees that it's easy!
- be careful with letter ordering, perhaps write down:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

CHECK: qsort with Lomuto's Partitioning

Q8. Run quicksort with Lomuto's for [A N A L Y S I S]. Here we rewrite the sequence after each swap and at the start/end of the loop.

Partitioning of
A N A L Y S I S

A_s N_i A L Y S I S
A_s N A L Y S I S_i
A_s [N A L Y S I S]

Partitioning of
N A L Y S I S

N_s A_i L Y S I S
N A_{si} L Y S I S
N A L_{si} Y S I S
N A L I_s S Y_i S
N A L I_s S Y S_i

[I A L] N_s [S Y S]

Partitioning of
I A L

I_s A_i L
I A_{si} L
I A_s L_i

[A] I_s [L]

Partitioning of
S Y S

S_s Y_i S
S_s Y S_i

S_s [Y S]

Partitioning of
Y S

Y_s S_i
Y S_{si}
Y S_s i

[S] Y_s

Final sorted sequence
A I L N S S Y

CHECK: qsort with Hoare's Partitioning

Q9.1: Run quicksort with Hoare's for [A N A L Y S I S].

Partitioning of
A N A L Y S I S

A_i N A L Y S I S_j
A A_{ij} N L Y S I S
A A_{ji} N L Y S I S

[A] A_j [N L Y S I S]

Partitioning of
N L Y S I S

N_i L Y S I S_j
N L I_j S_i Y S
N L I_j S_i Y S

[I L] N_j [S Y S]

Partitioning of
I L

I_i L_j
I_j L_i

I_j [L]

Partitioning of
S Y S

S_i Y S_j
S S_i Y_j
S S_{ij} Y_j

[S] S_j [Y]

Final sorted sequence
A I L N S S Y

Also... Mergesort

Make sure you can run (by hand) Merge Sort

```
[4 1 3 5 2]
[A N A L Y S I S]
```

And, review the lectures for the remaining questions of problem 1. For each sorting algorithm, think:
which is the best situations when we want to employ that algorithm?
in which situations when we definitely don't want that algorithm?

Sorting Algorithms

	Selection	Insertion	Quick	Merge
Basic Idea	Identify the smallest of the remaining and swap it with the first.	Insert next element to the *sorted* LHS to make the new, extended sorted LHS	Partition array into a *lesser* and a *greater* (than pivot) halves.	Split into 2 halves, sort them, then merge.
Complexity	$\theta(n^2)$	$O(n^2)$	$O(n^2)$	$\theta(n \log n)$
In-place?	✓	✓	✓	✗
Stable?	✗	✓	✗	✓
I- Sensitive				
The Good	• minimal number of swaps: $\theta(n)$, the choice when swaps are expensive	• Fast for nearly sorted arrays, or for small-size array	• expected: $\theta(n \log n)$ • fast in-memory sorting for big data	• always $\theta(n \log n)$ • fast external/internal sorting for big data • easy parallelism
The Bad	• always slow $\theta(n^2)$ especially when n big	• expected: slow $\theta(n^2)$, especially when n big	• could be slow, not a choice when n small	• not in-place: need extra memory for data

DP review: The Method

Problem 9.2: Longest Common Substring

The Task:

- Input: 2 strings: `a[0..n-1]` of length `n` and `b[0..m-1]` of length `m`
- Output: a longest string that appears in both `a` and `b`
- Example: `a= "UNDERSIGN"`, `b= "DESIGN"` → soln= `"SIGN"`
- Notes:
  - do not confuse with *finding a longest common subsequence* where soln is `"DESIGN"`, where "subsequence" has no requirement of being contiguous as in substring

Method 1: exhaustive search, discuss:

- what are all possible substrings?
- how to check if a substring is common, and how to find the longest one?

Problem 9.2.Method1.soln: Longest Common Substring – exhaustive soln

The Task: a[0..n–1], b[0..m–1] → LCS of a, b
Example: a= "UNDERSIGN", b= "DESIGN" → "SIGN"

a) [very bad exhaustive search]

- a has n(n+1)/2, b has m(m+1)/2 substrings
- we can compare them pair-wise to find the common, and hence the LCS,
- complexity: $O(m^2n^2m)=O(n^2m^3)$!

b) [exhaustive, but better]

- only consider the starting point of a substring in b, we have m starting points,
- for each such point i, do exhaustive pattern matching of b[i..m–1] in a, keep track of the longest match
  → total time complexity= $O(m^2n)$
- Worst case: when all character comparisons are matched (well, when all characters in both strings are just the same like AAAAAA and AAA) → $\theta(m^2n)$

Problem 9.2: Longest Common Substring – a DP solution

The Task: `a[0..n-1]`, `b[0..m-1]` → LCS of `a, b`

Example: `a= "UNDERSIGN"`, `b= "DESIGN"` → `"SIGN"`

any *known problem* which is similar or just a bit similar?

what does the soln look like?
DP(?)

parameters:
- how many?
- which?

sub-problems?
DP(?)
base cases?

what's the table's format

Problem 9.2: Longest Common Substring – a DP solution

The Task: `a[0..n-1]`, `b[0..m-1]` → LCS of `a, b`

Example: `a= "UNDERSIGN"`, `b= "DESIGN"` → `"SIGN"`

any known problem?
all palindromic
substrings?

what does the soln
look like?
DP(n,m)= LCS(a,b)

parameters: at least
- 1 for moving along a: value 1..n
- 1 for moving along b: value 1..m

sub-problems?
DP(i, j)= ?
base cases:

what's the table's
format

DP: longest common substrings

Store DP(i,j) in a table T →

- T[i][j] = DP(i,j)
- and T[n−1][m−1] should give solution

T[i][j]= something from comparing a_i with b_j

= 0 if $a_i \neq b_j$

= ? if = a_i = b_j

perhaps use a simple example "ABC" "XBCD"

we knew the soln is "BC"

Base cases: perhaps when i=0 or j=0 ?

Table: 2D, row i for a_i, with i= 0..n−1

col j for b_j, with j= 0..m−1

	X	B	C	D
A	0	0	0	0
B	0	?	0	0
C	0	0	?	0

we go "bottom-up" from small i, j to bigger i,j.

At i=2 ('C'), j = 2 ('C'), how can ? supply the information that the LCS-so-far is "BC", not just "C"?

DP-soln: longest common substrings

Homework:
- Complexity= ?
- Pseudocode

T[i][j]= something from comparing a_i with b_j

= 0 if $a_i \neq b_j$

= len of substring ended at (i,j)
T[i−1][j−1] + 1 if $a_i = b_j$

the table and take the diagonal

		D	E	S	I	G	N
	i \ j	0	1	2	3	4	5
U	0	0	0	0	0	0	0
N	1	0					
D	2	1					
E	3	0					
R	4	0					
S	5	0					
I	6	0					
G	7	0					
N	8	0					

Base cases: i=0 OR j=0

General case: building up the table for
(i= 1..n−1,
j= 1..m−1):
= 0 or
= T[i−1][j−1]+1
depending on
$a_i \neq b_j$ or not

Backtrack for soln:
find the largest value (4) in

		D	E	S	I	G	N
	i \ j	0	1	2	3	4	5
U	0	0	0	0	0	0	0
N	1	0	0	0	0	0	1
D	2	1	0	0	0	0	0
E	3	0	2	0	0	0	0
R	4	0	0	0	0	0	0
S	5	0	0	1	0	0	0
I	6	0	0	0	2	0	0
G	7	0	0	0	0	3	0
N	8	0	0	0	0	0	4

DP Review:

How to build DP sub-problems, parameters, relationship between subproblems? Review:

- Bean Bundles & Knapsack (re-using last week's slides)
- Coin-Row problem (last week)
- all-palindromic substrings (last week?)
- Longest Common Substring (this week)

How to build the pseudocode for the found sub-problems and relationships and how to run a sample? Review:

- Bean Bundles & Knapsack revisited (re-using last week's slides)
- Longest Common Substring (this week)

Important:

- There are some good examples with solutions in Levitin's chapter 8

DP: Learn examples in the Levitin's book

-

Most crucial step in DP: finding parameters & recurrences

Coin-row problem

There is a row of n coins whose values are some positive integers $c_1, c_2, \ldots, c_n$, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

Parameters:
Recurrence:
Base case:

Change-making problem

Give change for amount W using the minimum number of coins of denominations $d_1 < d_2 < \ldots < d_n$. Assume availability of unlimited quantities of coins for each of the n denominations d_i and $d_1 = 1$.

Parameters:
Recurrence:
Base case:

all palindromes

Given a string S of length n, construct a DP algorithm to find all nonempty substrings that are palindromes.

Parameters:
Recurrence:
Base case:

Some other problems

- Coin-collecting problem Levitin.8.1.example3
- Order of matrix multiplication: exercise L8.3.11
- Optimal binary search tree, described in L8.3
- …

- DP is typically used in optimization problems, where we are trying to find the best possible solution given a set of constraints. Examples: 3 of the above 4 boxes, knapsack, bean bundles...
- DP can also be used to solve problems that have overlapping sub-problems or can be divided into smaller, simpler sub-problems. Examples: fibonacci, Warshall, Floyd, all palindromes…

Additional Slides

Q 9.3 [homework]

a) Design an algorithm Quickselect based on Quicksort which uses the `Partition` algorithm to find the `k`-th smallest element in an array `A`.

b) Show how you can run your algorithm to find the `k`-th smallest element where `k = 4` and `A = [9,3,2,15,10,29,7]`.

c) What is the best-case time-complexity of your algorithm? What type of input will give this time-complexity?

d) What is the worst-case time-complexity of your algorithm? What type of input will give this time-complexity?

e) What is the expected-case (i.e., average) time-complexity of your algorithm?

f) When would we use this algorithm instead of the heap based algorithm from Question 3 [NA this year]

partitionning & qselect

```
   partition( A[lo..hi]):
     ...
     return m
```

```
11  # input:  lo ≤ k ≤ hi, A[lo..hi]  not ordered
12  # output: the k-th smallest in A[0..n-1]
13  function qselect( A[lo..hi], k)
14    m= partition(A[lo..hi])
15    # how can we use the value m ?
16
17
```

```
21  function ksmallest(A[0..n-1], k)
22    if (k>=0 && k<n)
23      return qselect(A[0..n-1], k)
```

Check your answer: partitioning & qselect

```
partition( A[lo..hi]):
  ...
  return m
```

```
# Note: this soln is a bit different from the ED's soln
# Here: we don't change k, only change lo and hi in the subsequent calls
11  function qselect( A[lo..hi], k)
12    m= partition(A[lo..hi])
13    if (k==m) then return A[m]
14    if (k<m) then
15      qselect(A[lo..m-1], k)
16    else
17      qselect(A[m+1..hi], k)
```

```
21  function ksmallest(A[0..n-1], k)
22    if (k>=0 && k<n)
23      return qselect(A[0..n-1], k)
```

Check your answer: partitioning & qselect

b) Show how to find the `k`-th smallest element where `k = 4` and `A = [9,3,2,15,10,29,7]`.

- as a standard way, take the leftmost as the pivot
- better to use Hoare's partitioning because it's (probably) easier to run by hand

Partition 1 - qselect(A[0..6], 4):

pivot=9, m= 3 < k, and the right half is A[4..6]= [10, 29, 15]

Partition 2 – qselect(A[4..6], 4):

pivot=10, m= 4 =k (and left half is empty), answer found A[4]= 10

c) What is the best-case time-complexity of your algorithm? What type of input will give this time-complexity?

Best case: only one partition, m=k. Simplest case of that: when A[] is sorted and k=0

c) What is the worst-case time-complexity of your algorithm? What type of input will give this time-complexity?

Worst case: do n partitioning, ie. m≠k in the first n-1 partitioning

Example input: A[] is sorted in increasing order, k= n-1. Each partitioning chose the the leftmost= smallest at the pivot, and results in empty left half.

d) What is the expected-case (i.e., average) time-complexity of your algorithm?

*Very i*nteresting. If you have time, try or read soln in ED

LAB: some interesting codes to understand and change a bit

- How to get the wall-clock running time of a function?