COMP20007 Workshop Week 8: Counting Sort and ...

ATTN

Interesting, Important, & Unfamiliar Topic Next Workshop: **Dynamic Programming**.

Remember at least to attend/watch lectures before the workshop.

1	**Counting Sort** **Radix Sort**
2	**The kth-smallest Problem, Q8.1, Q8.2**
3	**Interesting Lab questions that you should at least do in the algorithmic level:** **- W8.4 on Dijkstra: W8.4 []** **- W8.2 adaptive merge sort** **and/or any other questions from previous weeks**
LAB	**Order: W8.1, W8.3**

A Special Case for Sorting, when keys have limited range

Example 1
The Task: Sort arrays like: A[0..6]= {5,5,4,6,4,6,5} in the increasing order, in a time-efficient way. Supposing that the array might be large, but the elements are integers in between min=4, max= 6.
Solution:

A Special Case for Sorting, when keys have limited range

Example 1	Example 2
The Task: Sort arrays like: A[0..6]= {5,5,4,6,4,6,5} in the increasing order, in a time-efficient way. Supposing that the array might be large, but the elements are integers in between min=4, max= 6.	The Task: Sort array of records like:, A[0..6]= {(5,80),(5,70),(4,60),(6,90),(4,70),(6,70),(5,80)} in the increasing order of the first component, in a time-efficient way. Supposing that the array might be large, but the first components have min=4, max= 6
Solution:	Solution:

A Special Case for Sorting, when keys have limited range

Example 1	Example 2
The Task: Sort arrays like: A[0..6]= {5,5,4,6,4,6,5} in the increasing order, in a time-efficient way. Supposing that the array might be large, but the elements are integers in between min=4, max= 6.	The Task: Sort array of records like:, A[0..6]= {(5,80),(5,70),(4,60),(6,90),(4,70),(6,70),(5,80)} in the increasing order of the first component (key), in a time-efficient way. Supposing that the array might be large, but the first components have min=4, max= 6
Solution: **Step 1**: Build the table D[] of frequencies of each possible values Values: 4 5 6 D[]= { 2, 3, 2 } **Step 2**: Use D[] to populate the values to A[], overwriting A[]	Solution: Step 1: the same, build table of frequencies of the keys Step 2: ???

A Special Case for Sorting, when keys have limited range

Example 1	Example 2
The Task: Sort arrays like: A[0..6]= {5,5,4,6,4,6,5} in the increasing order, in a time-efficient way. Supposing that the array might be large, but the elements are integers in between min=4, max= 6.	The Task: Sort array of records like:, A[0..6]= {(5,80),(5,70),(4,60),(6,90),(4,70),(6,70),(5,80)} in the increasing order of the first component (key), in a time-efficient way. Supposing that the array might be large, but the first components have min=4, max= 6
Solution: **Step 1**: Build the table D[] of frequencies of each possible values Values: 4 5 6 D[]= { 2, 3, 2 } **Step 2**: Use D[] to populate the values to A[], overwriting A[] **this algorithm is *in-place***	Solution: Step 1: the same, build table of frequencies of the keys Step 2: need to copy elements of A[] to a new (sorted) array **In general, Distribution Counting Sort is *not in-place***

Distribution Counting Sort: how

Conditions: keys are integers in a small range (small in comparison with n), for example: array of positive integers, each ≤ 2:

`input keys:` A[0..6]= {5,5,4,6,4,6,5}   min=4, max=6, range=3

frequency of A[i] is stored in D[A[i]-min]

index/value	**0**	**1**	**2**
frequency table D[ ] =	2	3	2
index range in the sorted array	**0..1**	**2..4**	**5..6**

	Option 1	**Option 2**
transform D to position array	storing *end-position* of keys: D= {1,4,6} (or D= {2,5,7} as in lecture) *pros*: simple transition from frequency table	storing *start-position* of keys: D= {0,2,5}
how to build sorted array B to ensure ***stability***?	scanning input array A ***right-to-left*** for i from n-1 to 0 do B[D[A[i]-min]] = A[i] D[A[i]-min]= D[A[i]-min]-1	scanning input array A ***left-to-right*** for i from 0 to n-1 do B[D[A[i]-min]] = A[i] D[A[i]-min]= D[A[i]-min]+1

Exercise: Counting Sort for sorting array A[0..n-1], l=3, u= 5 [using the lecture's algorithm]

Input: A[0..6]= {5,5,4,6,4,6,5}

l= 4, u= 6

Output: B[0..6] which is the sorted version of A[]

Distribution Counting summary

Unlike comparison-based sorting algorithms, **Distribution Counting Sort:**

- Sorts integers by counting their frequencies, not by comparison.
- Fast, sorting in linear time, but:
  - efficient only when the range of integer keys (r= max - min + 1) is O(n). Otherwise, generally not applicable.

Should we apply Counting Sort to sort n integers?

Time complexity, supposing r= max-min+1*:*

- ***P***(n+r), or
- ***P***(n) if r ∈ O(n)

Special properties:

- *not in-place*, ie. requiring additional arrays for data records
- additional memory: ***P***(n+r), or ***P***(n) if r ∈ O(n)
- with *careful implementation*, the sorting is *stable*

Radix Sort: some jargons

- **Alphabet:** A finite set of symbols. Examples include the letters 'a' through 'z', the digits '0' through '9', or the hexadecimal digits '0' through '9' and ‘A' through ‘F’.
- **String:** A sequence of characters formed by selecting zero or more symbols from a given alphabet. For instance, "cat" is a string from the English alphabet, and "1A3" is a string from the hexadecimal alphabet.
- **Radix (or Base):** The number of distinct symbols within the alphabet being used. For the decimal alphabet (0-9), the radix is 10. For the lowercase English alphabet (a-z), the radix is 26.
- **Least Significant Position:** The rightmost position in a string. For "cat", it's the position of 't'. For number 123, it’s the position of 3.
- **Most Significant Position:** The leftmost position in a string. For "cat", it's the position of 'c'. For number 123, it’s the position of 1.

Radix Sort= sort strings by sorting each character at a time, from right to left

Applied when all keys can be represented as *same-size strings* over a small-size alphabet σ. Examples:

{22,17,167,28,173,…} where $0 \leq x_i < 256$

= { 022, 017, 167, 028, 173,…} σ= {0,1,…,9} using 3-digit

= { 16, 11, A7, 1C, AD,…} σ= {0,1,…,9,A,B,..F} using 2-digit

= {0001 0110, 0001 0001, 1010 0111, 0001 1100, 1010 1110,…} σ= {0,1} using 8-digit

Radix Sort: Iteratively sort the strings based on the characters at each position, from the **rightmost** (least significant) position to the **leftmost** (most significant).

- At each position, use a **stable sort** (typically Counting Sort).

Example: for { 001, 110, 001, 010, 100, 101}

Complexity for n strings of length m: $\Theta(n \times m)$

Radix Sort can be very fast (faster than comparison sorting) if keys are short (e.g. m is small)

Should we apply radix sort for an array of positive integers?

Q8.1 - Counting Sort: Use counting sort to sort the following array of characters:

[a, b, a, a, c, d, a, a, f, c, b]

How much space is required if the array has n characters and our alphabet has k possible letters.

Q8.2 - Radix Sort: Use radix sort to sort the following strings:

`abc bab cba ccc bbb aac abb bac bcc cab aba`

As a reminder radix sort works on strings of length k by doing k passes of some other (stable) sorting algorithm, each pass sorting by the next most significant element in the string. For example in this case you would first sort by the 3rd character, then the 2nd character and then the 1st character.

Q8.3: Which property is required to use counting sort to sort an array of tuples by only the first element, leaving the original order for tuples with the same first element. For example, the input may be:

(8, campbell), (6, tal), (3, keir), . . . (6, gus), (0, nick), (8, tom)

Discuss how you would ensure that counting sort satisfies this property. Can you achieve this using only arrays? How about using auxiliarry linked data structures?

Check your answer: Q10.2

Radix Sort: Use radix sort to sort the following strings:

`abc bab cba ccc bbb aac abb bac bcc cab aba`

First, sort (using *stable* counting sort) by the last letters:

ab**c** ba**b** cb**a** cc**c** bb**b** aa**c** ab**b** ba**c** bc**c** ca**b** ab**a**

→cb**a** ab**a** ba**b** bb**b** ab**b** ca**b** ab**c** cc**c** aa**c** ba**c** bc**c**

Next, do the same for the middle letter :

c**b**a a**b**a b**a**b b**b**b a**b**b c**a**b a**b**c c**c**c a**a**c b**a**c b**c**c

→bab cab aac bac cba aba bbb abb abc ccc bcc

Last, do the same for the first letter:

bab **c**ab **a**ac **b**ac **c**ba **a**ba **b**bb **a**bb **a**bc **c**cc **b**cc

→ aac aba abb abc bab bac bbb bcc cab cba ccc

Note: the sorting method is required to be stable, why?

Previous Weeks' Exercises: quicksort and k-smallest

Group Work on Algorithm Design: The k-th smallest problem

Problem: Given an array, find its k-th smallest value.

Task: Design and compare the complexity of 5 different algorithms.

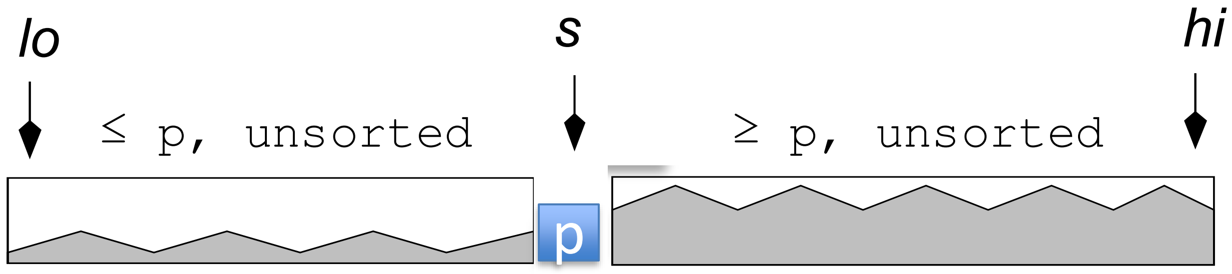
A reasonable plan:

- Discuss and briefly describe algorithms with complexity analysis in class.
- Choose two distinct optimal methods for pseudocode implementation in class.
- Complete remaining work at home, if desired.

Related Questions: 8.1, 8.2,

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

```
function QuickSort(A[lo..hi])
  if lo < hi then
    s := Partition(A[lo..hi])
    QuickSort(A[lo..s-1])
    QuickSort(A[s+1..hi])
```



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

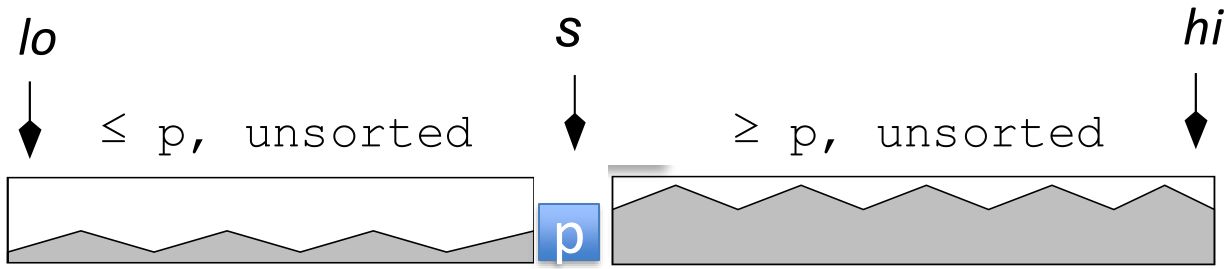
Note: a Partition of n elements has the complexity of $\theta(n)$

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[lo..hi])
  if lo < hi then
    s := Partition(A[lo..hi])
    QuickSort(A[lo..s-1]
    QuickSort(A[s+1..hi]
```



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

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

n

2 x (n/2)

4 x (n/4)

$\log_2 n$ levels

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

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

0:n-1

0:n-2

0:n-3

0:n-4

...

n-1 levels

What is the (additional) space complexity of **Quicksort**? for ecursion:
BEST/AVERAGE O(log n) WORST O(n)

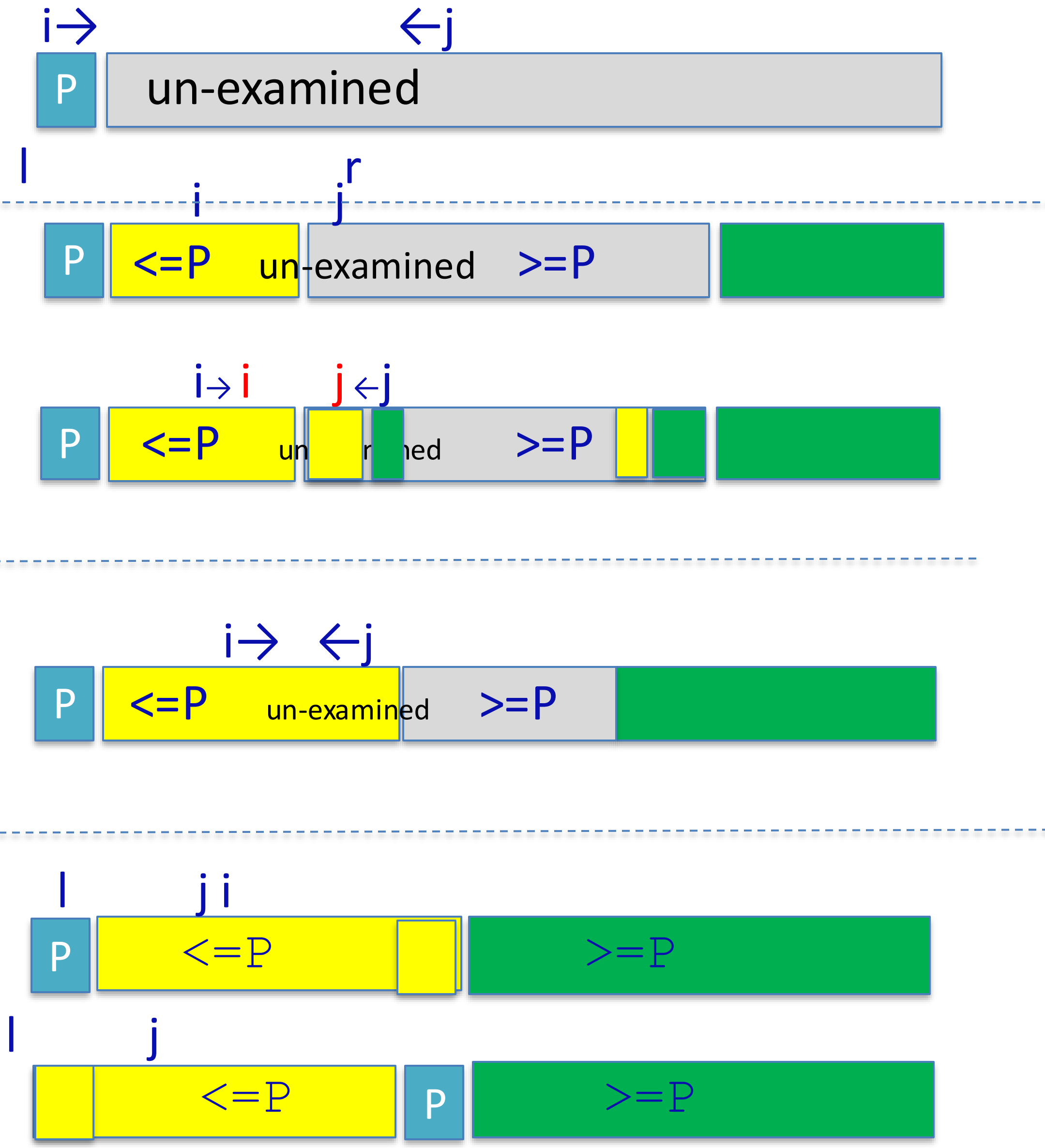
- Is it input-sensitive? Y
- Is it in-place? Y
- Is it stable? N – but we need to check with Partitioning

loop invariant	l i j h A[l+1..i-1]≤P A[i..j] un-examined A[j+1..h]≥P

```
function Partition(A[l..h])
 i ← l; j ← h
 P ← A[l]          # loop init
   repeat
       # move i forward until A[i]>P
       while i<h and A[i]≤P do i ← i+1
       # move j backward until A[j]<=P
       while A[j]>P do j ← j-1

       # 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
`
```



Note	This slide shows that the algorithm presented here and the one in the lectures are basically the same. They are just one variation of implementation of Hoare's Partitioning

```
function Partition(A[l..h])

 P ← A[l]; i ← l; j ← h

   repeat
       # move i forward until A[i]>P
       while i<h and A[i]≤P do i ← i+1
       # move j backward until A[j]<=P
       while A[j]>P do j ← j-1
       # extend yellow and green area
     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
`
```

Note: there are a number of different ways to implement the Hoare's partitioning

function $\text{PARTITION}(A[lo..hi])$

$\quad p \leftarrow A[lo]$; $i \leftarrow lo$; $j \leftarrow hi$

$\quad$ **repeat**

$\quad\quad$ **while** $i < hi$ **and** $A[i] \leq p$ **do** $i \leftarrow i + 1$

$\quad\quad$ **while** $j \geq lo$ **and** $A[j] > p$ **do** $j \leftarrow j - 1$

$\quad\quad$ *swap*$(A[i], A[j])$

$\quad$ **until** $i \geq j$

$\quad$ *swap*$(A[i], A[j])$ — undo the last swap

$\quad$ *swap*$(A[lo], A[j])$ — bring pivot to its correct

$\quad$ **return** j

end function

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

```
function HoarePartition(A[l..h])

 P ← A[l]; i ← l+1; j ← h

   repeat
      # move i forward until A[i]>P
      while i<h and A[i]≤P do i ← i+1
      # move j backward until A[j]<=P
      while A[j]>P do j ← j-1
      # extend yellow and green area
     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:

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

To simplify, we will write out the sequence only:

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

Question 7.1: Group Work

Know how to run by hand the following algorithms:

a) Selection Sort
b) Merge Sort
c) Quick Sort with Hoare's Partitioning

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

LAB: Some lab questions are interesting for discussion

W8.4 Update Dijkstra's solution after a single change of (u,v,w)

W8.5 Suppose that an input array already contains some "ordered runs", like

{ 9, 2, 4, 8, 20, 7, 3, 15, 18, 21, 6}

How would you adapt mergesort for a better running time? Would you start with top-down or bottom-up mergesort?

Check: The k-th smallest problem

Problem: Given an array, find its k-th smallest value (supposing that k is zero-origin).

Task: Design and compare the complexity of 5 different algorithms.

Method	Description	Complexity
Repeated Minimum Finding	Iteratively find the minimum of the remaining unsorted portion A[i...n-1] (for i from 0 to k) and swap it with A[i].	O(nk)
Sorting	Sort the entire array A using an O(nlogn) algorithm, then return the element at index k (assuming 0-based indexing)	O(nlogn)
Quickselect	Use the partitioning step of Quicksort to recursively narrow down the search to the subarray containing the k-th smallest element	O(n) (average case), $O(n^2)$ (worst case)
Min-Heap	Build a min-heap from the array and extract the minimum element k+1 times. The last extracted element is the k-th smallest	O(n + klogn)
Max-Heap of k+1 Elements	Build a max-heap of the first k+1 elements. Then, for each remaining element in the array, if it's smaller than the root of the max-heap, replace the root with the current element and heapify. The root of the final max-heap is the k-th smallest	O(k + (n−k)logk)

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 N_i A_j L Y S I S
A A_j N_i 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 Y_i S_j
S S_j Y_i

[S] S_j [Y]

Final sorted sequence
A I L N S S Y