Tutorial

1. Simple Sorting Algorithms

We saw the following sorting algorithms,

a. Selection Sort

b. Merge Sort

c. Quicksort (with Lomuto partitioning)

Answer the following questions about each 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[1]?

iv. Does the algorithm sort in-place[2]?

v. Is the algorithm input sensitive[3]?

If you get time, try to answer these questions for (d) Quicksort (with Hoare partitioning), and (e) Insertion Sort.

2. Longest Common Substring Problem

Last week we looked at one dynamic programming problem aligned with the Rod Cutting Problem. This week we will look at another problem called the Longest Common Substring Problem, this finds the longest string which is present in both strings.

Answer the following questions working towards an efficient solution for the Longest Common Substring Problem:

i. Discuss an exhaustive method of finding the longest common substring between

[U N D E R S I G N]

and

[D E S I G N]

ii. What is the time complexity of the algorithm?

iii. Discuss what subproblem which could be optimally solved using Dynamic Programming ideas.

iv. Derive and use an algorithm to find the longest common substring between the two strings.

v. What is the complexity of the Dynamic Programming approach?

3. (Homework) Quickselect

3. (Homework) Quickselect

–

Quicksort uses a Partition(A,$pivot$) function which partitions the array A to satisfy the constraint that all elements smaller than the *pivot* occur before it in the array, and those greater than the *pivot* occur after in. In quicksort we call Partition, keep track of the final index of the *pivot*, p and call Partition recursively on $A[0...p-1]$ and $A[p+1...n-1]$.

a. Design an algorithm based on Quicksort which uses the Partition algorithm to find the kth-smallest element in an array A.

   Hint: We know that once we partition the array, the index of the *pivot*, p, must be the correct position of *pivot* in the final sorted array. Can we use this fact to deduce where the kth smallest element must be?

b. Show how you can run your algorithm to find the kth-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?

- [1]a sorting algorithm is *stable* if the relative order of elements with the same value is preserved.↩
- [2]Sorting *in-place* is when only $O(1)$ additional space is required.↩
- [3]An algorithm is *input sensitive* if the runtime depends on properties of the input other than its size, for instance whether or not the input is already sorted.↩