School of Computing and Information Systems
COMP90038 Algorithms and Complexity Tutorial Week 6

Sample answers

1. Apply the DFS-based topsort algorithm to linearize the following graph:

$$\begin{tikzcd}
a & b \arrow[l] \arrow[r] & c \\
d \arrow[u] \arrow[r] \arrow[d] & e \arrow[ul] \arrow[ur] \arrow[d] & \\
f & g \arrow[l] &
\end{tikzcd}$$

   Answer: Here is how the traversal stack develops:

$$\begin{array}{ccc}
 & & f_{7,4} \\
 & & g_{6,5} \\
 & c_{3,2} & e_{5,6} \\
a_{1,1} & b_{2,3} & d_{4,7}
\end{array}$$

   The reverse of the popping order is: d, e, g, f, b, c, a.

2. For what kind of array is the time complexity of insertion sort linear?

   Answer: Insertion sort is linear-time for already-sorted arrays, and arrays that are "almost-sorted". The definition of this is that the number of "inversions" is $O(n)$ where n is the size of the array. In next week's exercises we define inversions properly and ask for an efficient algorithm to count their number.

3. Trace how interpolation search proceeds when searching for 42 in an array containing (in index positions 0..21)

$$20, 20, 21, 23, 25, 26, 26, 27, 29, 29, 29, 30, 32, 33, 34, 36, 38, 40, 41, 43, 43, 45$$

   Answer: The first probe will be at position $x = 0 + \lfloor \frac{42-20}{45-20}(21-0) \rfloor = \lfloor \frac{22}{25} \cdot 21 \rfloor = 18$. At index 18 we find 41, so the entire segment from 0 to 18 is discarded. Interpolation sort is now ready to repeat the process for 43, 43, 45. However, it is essential to first check the extreme elements. Since $43 > 42$, the process must stop straight away, reporting failure.

4. Trace how QuickSelect finds the median of 39, 23, 12, 77, 48, 61, 55.

   Answer: We are after the median, and for an array of length 7, that means the fourth smallest element. Hence the algorithm's k is 4. QuickSelect will keep probing until it it manages to place the pivot in position $4 - 1 = 3$.

   First recursive call: Content is 39, 23, 12, 77, 48, 61, 55, and k is 4. Partitioning proceeds as follows: 39 is the pivot, and a (useless) swap happens for each of the next two elements, as they are smaller than the pivot. After that, all elements are greater than the pivot, so we end up with $s = 2$ (pointing to 12 in 39, 23, 12, 77, 48, 61, 55). Swapping 12 and the pivot, we get 12, 23, 39, 77, 48, 61, 55.

Since $s = 2 < 3 = 0+4-1 = lo+k-1$, the recursive call focuses on the sequence from index 3 onwards, and the new k in the recursive call is $k-(s+1) = 1$. That is, the process repeats, now looking for the smallest element in the sequence 77, 48, 61, 55. This time, partitioning proceeds as follows: With 77 being pivot, all other elements are smaller, so three useless swaps take place, changing nothing. Finally item 77 is swapped with the item in position $s = 6$, leaving us with 55, 48, 61, 77.

Since $s > 3+1-1 = 3$, the next recursive call focuses on 55, 48, 61 (that is, $lo = 3$), still with $k = 1$. This time partitioning swaps 55 and 48, and returns $s = 4$.

So a final recursive call happens, focusing very narrowly on item 48 only, with $k = 1$, and of course this call returns 48 as the final result.

5. We can use QUICKSELECT to find the smallest element of an unsorted array. How does it compare to the more obvious way of solving the problem, namely scanning through the array and maintaining a variable *min* that holds the smallest element found so far?

   Answer: The straight-forward way of scanning through the array is better. It will require $n-1$ comparisons. QUICKSELECT will require that number of comparisons just to do the first round of partitioning, and of course one round may not be enough. In fact, in the worst case we end up doing $\Theta(n^2)$ comparisons.

6. **Just for fun ... a revision question**

   Consider the *subset-sum problem*: Given a set S of positive integers, and a positive integer t, find a subset $S' \subseteq S$ such that $\sum S' = t$, or determine that there is no such subset. Design an exhaustive-search algorithm to solve this problem. Assuming that addition is a constant-time operation, what is the complexity of your algorithm?

   Answer:

```
function SUBSETSUM($S, t$)
    for each $S'$ in POWERSET($S$) do
        if $\sum S' = t$ then
            return *True*
    return *False*
```

How can we systematically generate all subsets of the given set S? For each element $x \in S$, we need to generate all the subset of S that happen to contain x, as well as those that do not. This gives us a natural recursive algorithm:

```
function POWERSET($S$)
    if $s = \emptyset$ then
        return $\{\emptyset\}$
    else
        $x \leftarrow$ some element of $S$
        $S' \leftarrow S \setminus \{x\}$
        $P \leftarrow$ POWERSET($S'$)
        return $P \cup \{s \cup \{x\} \mid s \in P\}$
function ADDTOEACHOF($P, x$)
    $result \leftarrow \emptyset$
    for each $s \in P$ do
        $result \leftarrow result \cup \{x\}$
    return $result$
```

There are 2^n subsets of S. $\sum S'$ calculates the sum of the elements in S', which requires $O(n)$ time. Hence the brute-force solution is $O(n \cdot 2^n)$.