School of Computing and Information Systems
COMP90038 Algorithms and Complexity Tutorial Week 7

Sample Answers

The exercises

42. 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.

43. 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.

44. Apply mergesort to the list S, O, R, T, X, A, M, P, L, E.

    Answer: Have fun :)

45. Apply quicksort (with Hoare partitioning) to the list S, O, R, T, X, A, M, P, L, E.

    Answer: Have fun :)

46. Use the Master Theorem to find the order of growth for the solutions to the following recurrences. In each case, assume $T(1) = 1$, and that the recurrence holds for all $n > 1$.

    (a) $T(n) = 4T(n/2) + n$

    (b) $T(n) = 4T(n/2) + n^2$

    (c) $T(n) = 4T(n/2) + n^3$

    Answer:

    (a) $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$.

    (b) $T(n) = \Theta(n^2 \log n)$.

    (c) $T(n) = \Theta(n^3)$.

47. When analysing quicksort in the lecture, we noticed that an already sorted array is a worst-case input. Is that still true if we use median-of three pivot selection?

    Answer: This is no longer a worst case; in fact it becomes a best case! In this case the median-of-three is in fact the array's median. Hence each of the two recursive calls will be given an array of length at most $n/2$, where n is the length of the whole array. And the arrays passed to the recursive calls are again already-sorted, so the phenomenon is invariant throughout the calls.

48. Let $A[0..n-1]$ be an array of n integers. A pair $(A[i], A[j])$ is an *inversion* if $i < j$ but $A[i] > A[j]$, that is, $A[i]$ and $A[j]$ are out of order. Design an efficient algorithm to count the number of inversions in A.

    Answer: We follow the hint that said to adapt mergesort. Let MERGESORT return the number of inversions that were in its input array (and as usual, have the side-effect of sorting it).

```
function MERGESORT(A[·], n) : Int
    if n > 1 then
        for i ← 0 to ⌊n/2⌋ − 1 do                ▷ Copy left half of A to B
            B[i] ← A[i]
        for i ← 0 to ⌈n/2⌉ − 1 do                ▷ Copy right half of A to C
            C[i] ← A[⌊n/2⌋ + i]
        b ← MERGESORT(B, ⌊n/2⌋)
        c ← MERGESORT(C, ⌈n/2⌉)
        a ← MERGE(B, ⌊n/2⌋, C, ⌈n/2⌉, A)
        return a + b + c
    else
        return 0
```

```
function MERGE(B[·], p, C[·], q, A[·]) : Int
    i ← 0; j ← 0; k ← 0
    res ← 0
    while i < p and j < q do
        if B[i] ≤ C[j] then
            A[k] ← B[i]
            i ← i + 1
        else
            res ← res + p − i
            A[k] ← C[j]
            j ← j + 1
        k ← k + 1
    if i = p then
        for m ← 0 to q − j − 1 do
            A[k + m] ← C[j + m]
    else
        for m ← 0 to p − i − 1 do
            A[k + m] ← B[i + m]
    return res
```

The idea is that, having split array A into B and C, the recursive calls to MERGESORT will count the inversions local to B and to C. It is the job of MERGE to count all cases (x, y) where x is an element from B which is greater than y, an element from C. MERGE does this at the point where it has identified such an x in $B[i]$ and such a y in $C[j]$. When $B[i]$ is the greater, then all of B's elements *after* position i are also greater than $C[j]$. So, before dismissing $C[j]$, we add $p - i$ to the count of inversions.

49. Let T be defined recursively as follows:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n-1) + n/2 \quad n > 1 \end{aligned}$$

The division is exact division, so $T(n)$ is a rational, but not necessarily natural, number. For example, $T(3) = 7/2$. Use telescoping to find a closed form definition of T.

Answer: Telescoping the recursive clause:

$$\begin{aligned} T(n) &= T(n-1) + n/2 \\ &= T(n-2) + (n-1)/2 + n/2 \\ &= T(n-3) + (n-2)/2 + (n-1)/2 + n/2 \\ &= T(2) + 3/2 + \ldots + (n-2)/2 + (n-1)/2 + n/2 \\ &= T(1) + 1 + 3/2 + \ldots + (n-2)/2 + (n-1)/2 + n/2 \\ &= 1 + 1 + 3/2 + \ldots + (n-2)/2 + (n-1)/2 + n/2 \\ &= 2 + \textstyle\sum_{i=3}^{n} i/2 \\ &= 2 + (\textstyle\sum_{i=3}^{n} i)/2 \\ &= 2 + ((n+3)(n-2)/2)/2 \\ &= 2 + \frac{(n+3)(n-2)}{4} \\ &= \frac{n^2+n+2}{4} \end{aligned}$$