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

Sample answers

The exercises

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

   Answer: Have fun

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

   Answer: Have fun

3. Let T be defined recursively as follows:

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

   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:

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

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

5. 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.

6. 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).

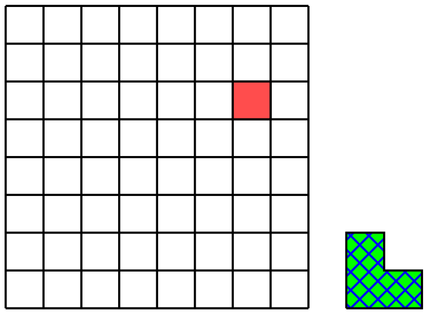
```
function MERGESORT(A[0..n−1]) : Int
    if n > 1 then
        copy A[0..⌊n/2⌋ − 1] to B[0..⌊n/2⌋ − 1]
        copy A[⌊n/2⌋..n − 1] to C[0..⌈n/2⌉ − 1]
        b ← MERGESORT(B[0..⌊n/2⌋ − 1])
        c ← MERGESORT(C[0..⌈n/2⌉ − 1])
        a ← MERGE(B, C, A)
        return a + b + c
    else
        return 0

function MERGE(B[0..p − 1], C[0..q − 1], A[0..p + q − 1]) : 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
        copy C[j..q − 1] to A[k..p + q − 1]
    else
        copy B[i..p − 1] to A[k..p + q − 1]
    return res
```

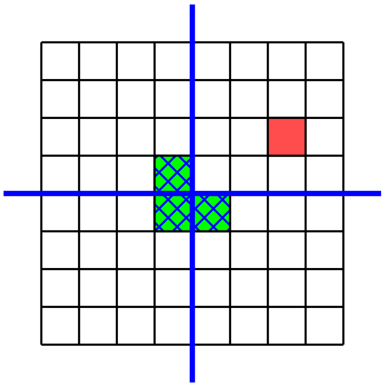
   The idea is that, having split array A into B and C, the recursive calls to MERGESORT will count the inversions local to B and to C. It is the job of MERGE to count all cases (x, y) where x is an element from B which is greater than y, an element from C. MERGE does this at the point where it has identified 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.

7. A *tromino* is an L-shaped tile made up of three 1×1 squares (green/hatched in the diagram below). You are given a $2^n \times 2^n$ chessboard with one missing square (red/grey in the diagram below). The task is to cover the remaining squares with trominos, without any overlap. Design a divide-and-conquer method for this. Express the cost of solving the problem as a recurrence relation and use the Master Theorem to find the order of growth of the cost.

   Hint: This is a nice example where it is useful to split the original problem into *four* instances to solve recursively.



   Answer: If $n = 0$ then we have a 1×1 board with a missing square, so there is nothing to cover. So let $n > 1$. Breaking the given board into four quarters corresponds to decrementing n by 1.



   One of the quarters will have the missing square, so place a tromino so that it borders that quarter, straddling the other three. Now we have four sub-problems of the same kind as the original, but each of size $2^{n-1} \times 2^{n-1}$, and we simply solve these recursively.

   Let us use m to denote the size of the problem, so $m = 2^{2n}$. The recurrence relation for the cost, in terms of m, is

$$T(m) = 4\ T(m/4) + 1 = 4\ T(m/4) + m^0$$

   with $T(1) = 1$. The Master Theorem tells us that $T(m) = \Theta(n^{\log_4 4}) = \Theta(n)$. That is, our method for solving the puzzle is linear in the size of the board.