Informatics 2 – Introduction to Algorithms and Data Structures

Tutorial 2: Analysis of Algorithms

1. In Lecture 2 we considered three algorithms (A, B, C) for computing $\mathbf{Expmod}(a, n, m) = a^n \bmod m$, and mentioned how this operation can be used to test for 'probable primes'. For example:

$$\mathbf{ProbablePrime}(n) \;=\; (\mathbf{Expmod}(2, n-1, n) == 1)$$

We can gain insight into the time complexity of this procedure by analysing the *number of arithmetic operations performed* $(+, -, \times, \mathrm{div}, \mathrm{mod})$.

(a) Give an asymptotic upper bound – i.e. O(some function of n) – for the number of arithmetic operations required to compute $\mathbf{ProbablePrime}(n)$ using Algorithm B for **Expmod**:

```
Expmod (a,n,m):
    b=1
    for i = 1 to n
        b = (b × a) mod m
    return b
```

(b) Do the same for Algorithm C:

```
Expmod (a,n,m):
    if n=0 then return 1
    else
        d = Expmod (a,⌊n/2⌋,m)
        if n is even
            return (d × d) mod m
        else return (d × d × a) mod m
```

Give informal justifications for your answers. (We are not expecting rigorous proofs here, though you're welcome to think about what they'd look like.)

2. The sorting algorithm known as **BubbleSort** works by repeatedly sweeping through an array, swapping any pairs of adjacent elements that are out of order. Here is a very crude version of **BubbleSort**. (NB. This may be simpler than other versions you might have come across, so pay close attention to the details of how the loops are set up.)

```
BubbleSort (A):
    for i = 1 to |A| − 1
        for j = 0 to |A| − 2
            if A[j] > A[j+1]
                swap A[j] and A[j+1]
```

(a) First, let's see why this algorithm works. Explain why after the *first* sweep through the array (i.e. after the completion of the j-loop when i=1), the largest element will be in its correct place at position $n-1$, where $n = |\mathrm{A}|$. Develop this idea to show that after $n - 1$ sweeps, the array will be fully sorted.

(b) What is the asymptotic worst-case number of comparisons performed by this algorithm for inputs of size n (as Θ(something))? What about the asymptotic best case?

(c) The above version of **BubbleSort** can be made more efficient in two ways. One of these might be suggested by your answer to part (a) above: perhaps we don't need to sweep through the *whole* of A each time? Another comes from noting that if we ever happen to complete a sweep of the array without doing any swaps, the array must be fully sorted and we can stop.

Write some pseudocode for a new version, **BubbleSort2**, that incorporates both these improvements. (It is this version, or something very close, that is most often referred to as 'BubbleSort'.)

(d) What are the asymptotic worst- and best-case runtimes for **BubbleSort2**? For what inputs do these worst and best cases arise?

(e) $\star$ (Optional) A useful measure of the *unsortedness* of an array A is the number of pairs of indices i,j $<$ |A| such that i $<$ j but A[i] $>$ A[j]. (Such a pair i, j is often called an *inversion*.) For example, a fully sorted array has unsortedness 0. A reverse-sorted array of size n has unsortedness $n(n-1)/2$, since here *every* pair i,j with i<j satisfies A[i] $>$ A[j].

Argue that the number of comparisons performed by **BubbleSort2** on input A is at least the unsortedness of A.

3. $\star$ The version of **MergeSort** given in lectures is rather wasteful on space, as it allocates a fresh array D for every merge performed. Give pseudocode for an alternative version of **MergeSort** that sorts a given array A of size n, returning the sorted result within A itself, and using another array B of size n as workspace, but not creating any other arrays.

[Here's one way to approach it: Write a function that sorts the portion of A from position m to n−1, by splitting this portion into four roughly equal parts, recursively sorting each of these, then merging the results with the help of B.]

Does your algorithm require any memory space beyond that used to store A and B? Give a Θ-estimate for the total space usage of your algorithm, informally justifying your answer.