

Informatics 2 – Introduction to Algorithms and Data Structures

Tutorial 2: Analysis of Algorithms

SOLUTIONS

1. (a) *Give an asymptotic upper bound for the number of arithmetic operations required to compute **ProbablePrime**(n) using Algorithm B for **Expmod**.*

When computing **Expmod** (a,n,m), each time round the for-loop we do 3 arithmetic operations (including the increment for i), so we do $\Theta(n)$ operations overall. Computing **ProbablePrime**(n) requires this plus one subtraction: still $\Theta(n)$.

- (b) *Do the same for Algorithm C.*

$\Theta(\lg n)$ arithmetic operations. Informally, this is because the computation of **Expmod** (a,n,m) recurses to depth $\lg n$ (give or take), since the value of n is halved with each recursive call — and at each level we perform either 4 or 5 arithmetic operations.

While a formal justification is not required, if one needed to provide one, you could argue as follows. Let $T(n)$ be the number of arithmetic operations required by Algorithm C to compute the solution on input n . By examining the pseudocode, we observe that this time is bounded by at most 5 arithmetic operations plus the number of arithmetic operations required by Algorithm C on input $\lfloor n/2 \rfloor$. We can write this as

$$T(n) \leq T(\lfloor n/2 \rfloor) + 5.$$

This is a *recurrence relation*, which is known to evaluate to $O(\lg n)$. We haven't learned how to solve recurrence relations yet, but we will see more about that later in the course; stay tuned!

2. (a) *Explain why after the first sweep through the array, the largest element will be in its correct place at position $n - 1$.*

Suppose the largest element starts at position k . If $k = n - 1$, then this element is already in the correct place, and clearly nothing will happen to move it. If $k < n - 1$, then when j reaches k , this largest element will be moved to position $k + 1$, and will then continue being moved to the right until j reaches $n - 2$, when the element will have reached position $n - 1$.

Develop this idea to show that after $n - 1$ sweeps, the array will be fully sorted.

Once the largest element is in position $n - 1$, exactly the same reasoning shows that after the second sweep, the second largest element will be in its correct

place at position $n - 2$. A simple induction (or an argument via a loop invariant) shows that after i sweeps, the largest i elements $x_1 \leq x_2 \leq \dots \leq x_i$ will be in their places at positions $n - 1, n - 2, \dots, n - i$ respectively. In particular, after $n - 1$ sweeps, the top $n - 1$ elements are in the right place, which can only mean that the remaining element (the smallest element) is also in the correct place at position 0.

- (b) *Asymptotic worst- and best-case number of comparisons for **BubbleSort**.*

The worst and best cases are the same: on *all* inputs of length n , the algorithm performs exactly $(n - 1)^2$ comparisons, which is $\Theta(n^2)$.

- (c) *Write some pseudocode for a new version, **BubbleSort2**, that incorporates both improvements.*

The first improvement is suggested by the answer to (a), when we come to sweep i , we know that the top $i - 1$ elements are already in their place, so we can stop at $j = n - 1 - i$. For the second improvement, we may use a boolean flag to record whether a swap has so far happened on the current sweep.

```

BubbleSort2(A):
    i = 1
    repeat
        i = i+1
        flg = false
        for j = 0 to |A|-i
            if A[j] > A[j+1]
                swap A[j] and A[j+1]
                flg = true
    until flg = false

```

- (d) *Asymptotic worst- and best-case number of comparisons for **BubbleSort2**.*

The worst case number of comparisons has roughly halved (now $n(n - 1)/2$), but is still $\Theta(n^2)$. The worst case occurs when the input A is reverse-sorted.

The best case is now just $n - 1 = \Theta(n)$: this occurs when A is already sorted. In this case, even the first sweep does not do any swaps, and we can stop immediately.

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

A rather pleasing argument. Suppose i, j is any inversion in the input A , i.e. we initially have $A[i] = x > y = A[j]$. Track the movements of x and y as the computation proceeds. At the start we have x before y , and at the end (when A is sorted) we must have x after y . But since both x and y can move by only one position at a time, there must be a time when these elements meet and are swapped; and at this point, they will be compared. So for any inversion i, j we have an associated comparison, and clearly no two inversions are associated with the same comparison in this way. So

$$\text{number of comparisons} \geq \text{number of inversions.}$$

3. *Write a version of MergeSort that uses just two arrays A and B of size n .*

We require two subroutines:

- **MergeAtoB**(m,p,n): merges the segment $A[m], \dots, A[p-1]$ with the segment $A[p], \dots, A[n-1]$ (assuming these segments are themselves already sorted), and writes the result to $B[m], \dots, B[n-1]$.
- **MergeBtoA**(m,p,n): merges the segment $B[m], \dots, B[p-1]$ with the segment $B[p], \dots, B[n-1]$, and writes the result to $A[m], \dots, A[n-1]$.

These are easy adaptations of the **Merge** procedure from lectures, except that they need not return a value. Note that these should work correctly even when one of the segments has length 0.

The following recursive procedure for MergeSort will then work:

```

MergeSort( $m,n$ ):
  if  $n-m > 1$ 
     $q = \lfloor (m+n)/2 \rfloor$ 
     $p = \lfloor (m+q)/2 \rfloor$ 
     $r = \lfloor (q+n)/2 \rfloor$ 
    MergeSort( $m,p$ )
    MergeSort( $p,q$ )
    MergeSort( $q,r$ )
    MergeSort( $r,n$ )
    MergeAtoB( $m,p,q$ )
    MergeAtoB( $q,r,n$ )
    MergeBtoA( $m,q,n$ )

```

What is the memory space use of this algorithm?

The arrays A and B (together) occupy $\Theta(n)$ of memory: this is the main space requirement.

However, we also need to keep track of certain information for each of the recursive calls to **MergeSort** currently in progress: specifically, the values of m,n,q,p,r , plus a record of which line of code we've got to in that call, so that we know where to return to. This is $\Theta(1)$ of information per call, and the maximum depth of recursion is $\lceil \log_4(n) \rceil$, so $\Theta(\lg n)$ of memory altogether. (In a typical programming language implementation, all this information will be stored on the *call stack*.)

While a call to **MergeAtoB** or **MergeBtoA** is in progress, there will also be the variables i,j,k associated with this call: just $\Theta(1)$ space.

So the total memory requirement is $\Theta(n) + \Theta(\lg n) + \Theta(1) = \Theta(n)$.