HW08 - Mixedupness

Prasun Surana
CS5008, Summer 1

1. The maximum mixed-up-ness score would occur when the array is maximally unsorted, i.e. in its original state, it is sorted in reverse/descending order. To find the maximum possible mixed-up-ness score, we can take a few examples and then extrapolate for larger numbers.

   If we have an array of size 1, sorting it is trivial, so our score is 0. If we have an array of size 2, for instance {2,1}, the score would be 1. For an array of size 3, e.g. {3,2,1}, the score would be 3, since 3 is greater than 2 and 1, and 2 is greater than 1. For an array of size 4, e.g. {4,3,2,1}, the score would be 6. We can summarise this further below:

Array Size	Maximum Mixed-up-ness Score
1	0
2	1
3	3
4	6
5	10
6	15
7	21
8	28
9	36

We have just looked at the first 9 array sizes for brevity. We notice that with each increase in array size, the new mixed-up-ness score is just the previous mixed-up-ness score added to the previous array size. For instance, the maximum mixed-up-ness score for an array of size 7 is 21, which is 15 (the previous mixed-up-ness score) + 6 (the previous array size). To illustrate, consider {7,6,5,4,3,2,1}. 7, the first element, is larger than 6 of the elements. Now, if we truncated the array to {6,5,4,3,2,1}, we already know the mixed-up-ness score of this array of size 6 is 15, so for an array of size 7, we just add 6 to 15, since the element 7 is now larger than the other 6 elements.

In the same way, a reverse-sorted array of size 16 would be 15 + the mixed-up-ness score of array[15]. The score for array[15] would be 14 + the mixed-up-ness score of array[14], and so on. This is essentially the triangle number sequence, a very common sequence in mathematics. We can find the 16th mixed-up-ness score by doing it manually, i.e. 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 = 120. There is also a formula to calculate the 16th term, which is n(n-1)/2 = (16*15)/2 = 240/2 = 120. Hence, the maximum mixed-up-ness score of an array of size 16 is 120.

2. In the brute-force algorithm, the worst case occurs when the array is maximally unsorted, i.e. initially, it is sorted in reverse order, for instance {10,9,8,7,6,5,4,3,2,1}. When counting the number of unordered pairs, the code will start from the first element and compare it to each successive element. It will then do this for each element. For 10, it will compare it to number from 9 to 1, and then increment the number of times where 10 is found to be greater than that number. 10 will have to be compared 9 times. However, 9 will have to be compared 8 times, 8 will have to be compared 7 times, and so on. So for each iteration of the outer loop, the number of operations in the inner loop is reduced, so it is not exactly $O(n^2)$. However, we must remember that Big-O denotes the upper bound, i.e. a bound beyond which our algorithm will not grow in it's worst case as the input size increases, so the Big-O time complexity can still be said to be $O(n^2)$ in the worst case.

3. $T(n) = 2T(n/2) + cn$. This is because during the 'divide' part of the algorithm, each iteration divides the problem into 2 problems, each of half the size. Combining them takes cn because it is a linear process.

4.

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

$T(n) = 2[2T(n/4) + cn/2] + cn$
$= \mathbf{4T(n/4) + 2cn}$

$T(n/4) = 2T(n/8) + cn/4$

$T(n) = 4[2T(n/8) + cn/4] + 2cn$
$= \mathbf{8T(n/8) + 3cn}$

$T(n/8) = 2T(n/16) + cn/8$

$T(n) = 8[2T(n/16) + cn/8] + 3cn$
$= \mathbf{16T(n/16) + 4cn}$

We can see that the general pattern is $\mathbf{T(n) = 2^kT(n/2^k) + kcn}$

Now we let $n = 2^k$
$n/2^k = 1$
So $T(n/2^k) = T(1)$

Plugging this in into the equation above, we get $\mathbf{T(n) = 2^kT(1) + kcn}$

Next, from $n = 2^k$, we can use $\log_2 n = k$.

Plugging this into the equation above, we get $\mathbf{T(n) = 2^{\log_2 n} + \log_2(n)*cn}$
$\mathbf{= n + c*n\log(n)}$

We can see that from the final equation above, the time complexity of the divide and conquer algorithm is nlog(n).

5. The Master Theorem is:

If $T(n) = aT(n/b) + O(n^d)$, where $a > 0$, $b > 1$, and $d >= 0$:

$T(n) = O(n^d)$ if $d > \log_b a$
$O(n^d \log n)$ if $d = \log_b a$
$O(n^{\log_b a})$ if $d < \log_b a$

For $T(n) = 2T(n/2) + cn$:
$a = 2$
$b = 2$
$d = 1$

$\log_b a = \log_2 2 = 1$

Therefore, since $d = 1$, and $\log_b a = 1$, by the Master Theorem, **$T(n) = O(n\log n)$.**