Homework 1. Due April 16

CS180: Algorithms and Complexity
Spring 2018

GUIDELINES:

- Upload your assignments to Gradescope by 5:59 PM.
- Follow the instructions mentioned on the course webpage for uploading to Gradescope very carefully (including starting each problem on a new page and matching the pages with the assignments); this makes it easy and smooth for everyone. As the guidelines are simple enough, bad uploads will not be graded.
- You may use results proved in class without proofs as long as you state them clearly.
- Most importantly, make sure you adhere to the policies for academic honesty set out on the course webpage. The policies will be enforced strictly. Homework is a stepping stone for exams; trying the problems will help you a lot for the exams.

1. Arrange the following in increasing order of asymptotic growth rate (i.e., $O(\ )$). For full credit it is enough to just give the order. [.75 points]

   (a) $f_1(n) = n^3$
   (b) $f_2(n) = 1000n^5/2$
   (c) $f_4(n) = n(\log n)^{1000}$
   (d) $f_5(n) = 2^{n \log n}$
   (e) $f_3(n) = 2^{3\sqrt{n}}$
   (f) $f_6(n) = 2^{(\log n)^{0.9}}$

   Solution. The correct order is $f_6, f_4, f_1, f_2, f_3, f_5$.

2. Use Karatsuba's algorithm to multiply the following two binary integers: 10110100 and 10111101. Your entire calculations should be in binary and show all your work. [.75 points]
   Solution. Will be worked out in discussion sections.

3. Solve the following recurrences: [.75 points]

   (a) $T(n) = 4T(n/5) + n$.
   (b) $T(n) = 6T(n/3) + n$.
   (c) $T(n) = 16T(n/4) + n^2$.

Solution. (a) $T(n) = O(n)$: Master theorem case (3) with $a = 4, b = 5$, $f(n) = n$.

(b). $T(n) = O(n^{\log_3 6})$: Master theorem case (1) with $a = 6, b = 3$, $f(n) = n$ and $k = \log_3 6 > 1$.

(c). $T(n) = O(n^2 \log n)$: Master theorem case (2) with $a = 16, b = 4$, $f(n) = n^2$ and $k = \log_4 16 = 2$.

4. We have a list A of n integers, for some $n = 2^k - 1$, each written in binary. Every number in the range 0 to n is in the list exactly once, except for one. However, we cannot directly access the value of integer $A[i]$ (for any i); instead, we can only access the j'th bit of i: $A[i][j]$. Our goal is to find the missing number.

   Give an algorithm to find the missing integer that uses $\Theta(n)$ bit accesses. Prove that your algorithm is correct and that it has the stated runtime. Note that a brute force solution that accesses every bit will take time $\Theta(n \log n)$. [.75 points]

   Solution. We will iteratively figure out the bits of the missing integer from left to right. For instance, to figure out the first bit of the missing integer, we can do the following. We count the number of 1's in the first column and compare it to 2^{k-1}. If the number of 1's is less than 2^{k-1}, then the missing number has a 1 in the first bit and else a 0 in the first bit. Further, as we now know the first bit of the missing integer, say b, it suffices to only search in the rows whose first entry is b. We can now repeat the argument for the pruned list of numbers which gives us another instance of the same problem but now with only $k-1$ columns and $2^{k-1}-1$ rows. By iterating this approach we can find the missing integer. This is formalized in the following algorithm:

   (a) Initialize $L = \{0, 1, \ldots, n-1\}$ to be the list of all the *active* row indices.
   (b) Initialize an array $MissingInt$ of length k.
   (c) For $j = 0, \ldots, k-1$:
      i. Let $N = \sum_{i \in L} A[i][j]$ be the number of 1's in the j'th column of A when looking at rows in L.
      ii. If $N < 2^{k-j-1}$, set $MissingInt[j] = 1$, else set $MissingInt[j] = 0$.
      iii. Set $L = \{i : A[i][j] = MissingInt[j]\}$.

   For $m \in \{1, 2, 2^2, \ldots, 2^k = n\}$, let $T(m)$ denote the number of bit accesses made when the number of integers left is m. Then, we have the recurrence $T(m) = T(m/2) + m$: the first term comes as in each iteration, we halve the number of integers we search over; the second term accounts for the number of bit accesses we need to count the number of 1's in the corresponding column. By master theorem case(3), $T(n) = O(n)$.

   You can also solve the recurrence by just expanding it out:

$$T(n) = n + T(n/2) = n + n/2 + T(n/4) = \cdots = n(1 + 1/2 + 1/4 + 1/8 + \cdots +) = O(n).$$

Practice Problem.

1. Call an array of n numbers $B[0], B[1], \ldots, B[n-1]$ *L-regular* if for any index $i \in \{0, \ldots, n-1\}$, $B[i]$ occurs within L places of its actual location in the sorted ordering. For example, any array of size n is n-regular and the array $[1, 3, 2, 5, 4]$ is 1-regular.

   Give an algorithm which takes as input L and a L-regular array B of size n and sorts the elements of B in $O(n(\log L))$-time.

Solution We give two solutions for the problem. The second algorithm is a little simpler than the first, but its analysis is more involved. There are several wrong ways to prove the correctness of the algorithms—especially the second one! Try to compare the solution to what you had carefully.

First Solution. The first solution is a generalization of Mergesort:

1. Let $k = \lceil n/2L \rceil$. Divide the array into k blocks of length $2L$ each (except perhaps the last block as follows): for $1 \leq i < k$, $B^i = B[(i-1)2L, \ldots, i \cdot 2L - 1]$, $B^k = B[(k-1)2L, \ldots, n-1]$.
2. For $i = 1, \ldots, k$ sort block B^i (in-place) using, say, Mergesort.
3. Let $A^1 = B^1$.
4. For $i = 1, \ldots, k-1$, suppose we have merged the first i blocks to get an array A^i of length $i \cdot 2L$. Then, to merge-in the next block, merge the last $2L$ elements of A^i with the $i+1$'th block B^{i+1} as in the Mergesort algorithm to get A^{i+1}.
5. Return A^k.

The above algorithm uses $k \cdot O(L \log L) = O(n \log L)$ time for sorting the blocks $B^0, \ldots, B^k$. On top of that, merging the $i+1$'th block with the last $2L$ elements of A^i takes $O(L)$ time. Therefore, the total time is $O(n \log L) + O(k \cdot L) = O(n \log L)$.

For correctness, we use induction to prove that for $i = 1, \ldots k-1$,

1. A^i is sorted.
2. The least $(i-1) \cdot 2L$ elements of A^i are the least $(i-1) \cdot 2L$ elements of our input array B and hence are in their right place.

For $i = 1$, there is nothing to prove. Now, suppose that the statement is true for i. We shall now show the same holds for A^{i+1}. The induction hypothesis implies that all elements of B^{i+1} are larger than the first $(i-1) \cdot 2L$ elements of A^i. Therefore, A^{i+1} is also sorted and hence is essentially the numbers in $B[0, \ldots, i \cdot 2L - 1]$ sorted in the right order. On the other hand, note that the least $i \cdot 2L$ elements of B must all appear in the first $(i+1) \cdot 2L$ places; else, such an element would be off by more than L places. Hence, the least $i \cdot 2L$ elements of the array B must be in A^{i+1} and as A^{i+1} is sorted, must be the first $i \cdot 2L$ elements of A^{i+1} and are in their right place. This is what we set out to prove for $i+1$. The property of A now follows from induction.

It now follows that A^k is B sorted in the right-order as required.

Second solution. Another solution is to use a *sliding window* approach for sorting the array.

1. For $j = 0, \ldots \lfloor n/L \rfloor$

Sort the elements in positions $B[L \cdot j], B[L \cdot j + 1], \ldots, B[L \cdot j + 2L - 1]$ using, say, Mergesort. Here, we **rewrite** the elements of B in these positions in the sorted order.

2. Sort the elements in positions $L \cdot \lfloor n/L \rfloor$ to $n - 1$.

Note that each of the sort calls takes time $O(L \log L)$ as we are only sorting a list of at most $2L$ numbers. Thus, the total-time is $(n/L) \cdot O(L \log L) = O(n \log L)$ as we make only n/L sort calls.

While the description of the algorithm is simpler than the first one we saw, proving correctness is much trickier here. We shall prove correctness by induction on the number of elements that we are sorting.

If $n \leq 2L$, there is nothing to prove, the claim follows immediately as we are sorting the whole block. Next, consider the first iteration where we sort the numbers $B[0], \ldots, B[2L - 1]$. Note that by the definition of L-regularity, the least L elements of B must be among this set (each can only move by at most L places to the left). Therefore, after the first iteration the least L elements are in their right places. We next argue that after this sort operation, the sub-array of numbers $B[L, \ldots, n-1]$ is also L-regular. Once we show this, we are done by induction as we are essentially applying the same algorithm to the array $B[L, \ldots, n - 1]$ which is of smaller length than what we started with.

To prove this, note that as we didn't touch $B[i]$ for $i > 2L$, they are still only at most L places off from their right place as before. So, we just have to analyze the numbers $B[L, \ldots, 2L - 1]$.

To prove the last part, suppose that we sort the elements $B[0], \ldots, B[2L - 1]$ by first finding the minimum element, moving it to the head of the array and repeating for the second largest element and so forth. We claim that each of these operations where we move the least element to the front of the unsorted part preserves L-regularity. Suppose the unsorted part is $B[i, \ldots, 2L - 1]$ which is L-regular and we next move the least element of this sub-array, say $B[j]$ to the front of the unsorted part. Then, only the positions of the elements $B[i], \ldots, B[j - 1]$ could have changed by this move so we only have to analyze these elements. Further, $j - i \leq L$ since $B[j]$ can't be off by more than L places. We now do a case-analysis for each $\ell \in \{i, i + 1, \ldots, j - 1\}$:

Case 1: If the correct place of $B[\ell]$ is after $B[j]$, then the distance of $B[\ell]$ to its right place can only be decreased by moving $B[j]$ to the front.

Case 2: If the correct place of $B[\ell]$ is in the window $B[i, \ldots, j - 1]$, then also we are fine as the total length of the window itself is at most $j - i \leq L$.

Therefore, in either case, moving the least element of the unsorted part to the front of the unsorted part preserves L-regularity. Hence, sorting the array $B[0], \ldots, B[2L - 1]$ should preserve L-regularity as we wanted to show.