Dov Kassai Discrete Final Question 2

dk778

1. Give a big-O bound on the number of bits needed to represent a number N. (2 points)
    1. The number of bits that are needed to represent the number N would be
        1. $\log_2(N)/\log_2(2)$
    2. The big oh for this would just be O(log n).
2. Give a big-O bound on the complexity of multiplying two integers between 1 and N? Hint: what is the worst case? Note, the bound should be simply in terms of N.
    1. If we say that there are n bits in the first number and k bits in the second number then the number of bits required would be $\log_2 n$ $\log_2 k$ and the time complexity of this would be:
    2. $O(\log_2 n * \log_2 k)$.
3. Given N, k, what is the big-O complexity of computing N^k? Hint: Consider fast exponentiation. Note, the bound should be in terms of N and k.
    1. the total time complexity would the final number of time state multiplication is done with the number of operations which are done during the multiplication, this would make the time complexity:
    2. $O(\sqrt{n} * \log_2(\sqrt{n}))$
4. Stop when either i) a2 = N and you have found the square root, or ii) a2 > N and N does not have an integer square root. Give a big-O bound on the overall complexity of this search in terms of N. (I couldn't properly copy the question here)
    1. We would have to sequentially search each number between 1 and n.
    2. First we would check if a^2=n, this means that we have found the square root and we will have to come compare it with n. If we have a^2>n this would mean that n does not have a integer square root and we will scan more than $\sqrt{n}$ elements as after that either a^2 will give greater than N
    3. So the overall time complexity would be $O(\sqrt{n})$.
5. As an alternate approach: we are effectively searching the set {1, 2, 3, . . . , N } for the value of $\sqrt{N}$ if it is there. For a given a, we can't compare a to $\sqrt{N}$ since $\sqrt{N}$ is not known. However, we can compute and compare a2 to N. Use this idea as the basis of a binary search-type algorithm. Give a big-O bound on the overall complexity of this search, in terms of N. How does it compare to the previous?
    1. We can set the upper bound to N and the lower bound to 1. Since it is binary search we would need to find the middle of the the lower and the upper bound. So we would take the average of the lower and the upper and square them. This would give us the time complexity of $\log_2(N)$. An d binary search would have the time complexity of $\log_2(n)$

2. the total time complexity would be $O(\log_2(N)*\log_2(N))=O((\log_2(N))^2)$.

6. Given that $1 \leq N^{(1/k)} \leq N$, consider sequentially taking each number $a \in \{1,2,3,...,N-1,N\}$ and computing a^k. Stop when either i)$a^k = N$ and you have found the k-th root ,or ii) $a^k > N$ andN does not have an integer k-th root. Give a big-O bound on the overall complexity of this search in terms of N and k.
    1. To find the big Oh of this we need to first establish what type of search we are using.
    2. Since we are using sequential search this would mean that it would take linear time.
    3. This also means that we will never have to check more than $n^{(1/k)}$ elements since it would stop when it is equal to K or it is greater than K.
    4. This would mean that the time complexity is $O(n^{(1/k)})$.
7. As an alternate approach: we are effectively searching the set {1, 2, 3, . . . , N } for the value of $N^{(1/k)}$ if it is there. For a given a, we can't compare a to $N^{(1/k)}$ since $N^{(1/k)}$ is not known. However, we can compute and compare a^k to N. Use this idea as the basis of a binary search-type algorithm. Give a big-O bound on the overall complexity of this search, in terms of N and k. How does it compare to the previous?
    1. well in this problem we are breaking it down the array into two parts and then looking for one of the elements in it
    2. Since we are breaking down the array into two and searching this would be almost identical to the time complexity of binary search which is O(log(N)).
    3. But when we have a sufficiently large N then $O(N^{(1/k)})$ would grow a lot more quickly than O(log(N)).
    4. So this shows that the second one is a lot better because it is a lot smaller than $O(N^{(1/k)})$ and thus would be more efficient.
8. For a given value of N what is the smallest value of K that may need to be considered? what is the largest value of K? Give a big O bound on the largest possible K
    1. We are looking for the equation for $A^K=N$ and we are trying to find the smallest value of K that should be considered
    2. The smallest case we can have is when k=1. This would mean that we have $a^k=N$. Meaning that it would be a would be 2 and N would be 2.
    3. Now to find the biggest value of k in this would be doing this the other way. We have $2^K=N$, which means we would need to just take the log of both side to get that k=Log(n).
    4. Since this would be the largest that it can be then it would be the big O as well so the big O is O(log(N)).
9. Consider extending the algorithm in question 7 in the following way:
    1. Using the binary search kind of algorithm that we had in 7.

2. If we start off by say ing that k=4 and n=64 we will have the low point=1 and the high point = 64, so the middle would be 32.
3. Next we will check that 32^4, it is clear that is is greater than 64 so we will need to reduce the upper limit to the high being 32.
4. The next step we need to find the new mid which would be 16 since the low is still 1 and and the high is 32 so the middle of those 2 is 16.
5. So now we check again and see that 16^4 is greater than 64 so we need to reduce this more and say that the high is now 16. this would be that the mid is now 8.
6. This would again be 8^3 is greater than 64 and now would reduce so the mid is 4
7. now we do it again and 4^4=64 so this would be our answer.
8. This can be seen that this is just a variation of binary search
9. The pseudo code would be as follows:
    1. int perfpow(int n, in k){
        1. int low=1, int high=n;
        2. while(low<high){
            1. int mid=(low+(high-low)/2;
            2. if (pow(mid,k)==n){
                1. return mid;}
            3. else if (pow(mid,k)>n){
                1. high= mid;}
            4. else{
                1. low=mid;}}
        3. return -1;}
10. The complexity of this would be $\log_2 n$, this would be much more efficient than linear search, since there it would have to go through n things to get to the right answer.
11. We are only only given n, and we are trying to find a^k=n, for any given n and k.
12. One way to do this would be to iterate through and look for the possible values of k. But the main point is trying to to find a value of k to get a greater number than the $\sqrt{n}$, and still be a prime number. The pseudo code then becomes
    1. int perfpow(int n, in k){
        1. for(int i=0;i<$\sqrt{n}$;i++){
            1. int low=1, int high=n;
            2. while (low<high){
                1. int mid=low+(high-low)/2;
                2. if (pow(mid,k)==n){
                    1. return mid;}

2. else if (pow(mid,k)>n){
    1. high = mid;}
3. else{
    1. low=mid;}}

3. return -1;}

13. the complexity will be $O(\sqrt{n} + \log_2(n))$.