

1. Proceeding along the lines of the problem done in class -

Probability of success in one attempt is $= \left(\frac{\sqrt{n}}{n}\right) * \left(\frac{\sqrt{n-1}}{n}\right) \approx \left(\frac{1}{n}\right)$

Probability of failure in one attempt is $\approx \left(1 - \frac{1}{n}\right)$

Hence, the probability of failure in k successive attempts is $\approx \left(1 - \frac{1}{n}\right)^k$

We want this probability to be no more than $n^{-\alpha}$. I.e., we want $\left(1 - \frac{1}{n}\right)^k \leq n^{-\alpha}$

Using the inequality $\left(1 - \frac{1}{x}\right)^x \leq \frac{1}{e}$, $\left(1 - \frac{1}{n}\right)^k \leq e^{-k/n}$. Equating $e^{-k/n}$ and $n^{-\alpha}$ we get: $k = \frac{\alpha n \log n}{\log e}$. Thus the number of attempts needed is no more than $\frac{\alpha n \log n}{\log e}$ with high probability.

In other words, the run time is $\tilde{O}(n \log n)$. If we only make $O(\log n)$ attempts, the probability of success can be verified to be not as high as what we want.

2. Algorithm \mathcal{A} is called $d\alpha \log_e n$ (d being a constant) times in the new algorithm. A count of the number of YESs and the number of NOs output by \mathcal{A} is kept. Let u be the total number of YESs and v the total number of NOs. If $u > v$, the new algorithm (call it \mathcal{B}) outputs YES; otherwise it will output NO.

\mathcal{B} 's output will be correct if there are more correct answers than incorrect answers in the $d\alpha \log_e n$ answers obtained from \mathcal{A} . Let X be the number of times \mathcal{A} outputs the correct answer and let Y be the number of times \mathcal{A} outputs the incorrect answer. We want to show that $\text{Prob.}[X > Y]$ is $\geq (1 - n^{-\alpha})$.

Note that X is a binomial random variable with parameters $(d\alpha \log_e n, c)$. Similarly, Y is a random variable with parameters $(d\alpha \log_e n, (1 - c))$. Since c is greater than $\frac{1}{2}$, let $c = \frac{1}{2} + \delta$ for some some constant $\delta > 0$. The expected value of X is $d\alpha \log_e n \left(\frac{1}{2} + \delta\right)$ and the expected value of Y is $d\alpha \log_e n \left(\frac{1}{2} - \delta\right)$.

Chernoff's bounds are typically used to show that if the mean of a binomial random variable is μ , then, with high probability, the actual value of the random variable can not be significantly greater (or less) than μ . This is what we are going to use to prove our result.

Using Chernoff's inequality,

$$\text{Prob.} \left[X \leq \left(1 - \frac{\delta}{2}\right) cd\alpha \log_e n \right] \leq e^{-\frac{\delta^2}{8} cd\alpha \log_e n}$$

The RHS of the above inequality will be $\leq n^{-\alpha}$ if $d \geq \frac{8}{c\delta^2}$.

Thus we have

$$\text{Prob.} \left[X \leq \left(\frac{1}{2} + \frac{3}{4}\delta - \frac{\delta^2}{2}\right) d\alpha \log_e n \right] \leq n^{-\alpha} \tag{1}$$

if $d \geq \frac{8}{c\delta^2}$.

From equation 1 we see that X will be more than Y with high probability if we pick $d \geq \frac{8}{c\delta^2}$.

3. To begin with we identify the first non-infinity cell by examining $a[1], a[2], a[4], a[8], \dots$. Let k be the smallest integer such that $a[2^k] = \infty$. Note that 2^k is no more than $2n$. Finding k takes $\Theta(\log n)$ time. Now the element x could be located by using binary search in the interval $a[1 : 2^k]$. This takes $O(\log(2n)) = O(\log n)$ time. Total run time is $\Theta(\log n) + O(\log n) = \Theta(\log n)$.

4. Maintain an array $a[]$ of n bits and a linked list L of all the elements in the set S . Bit $a[i]$ is set to 1 if integer i exists in the set S , otherwise $a[i] = 0$.

INSERT(i) If $a[i]$ is 0 then set $a[i]$ to 1 and append the element i to L .

DELETE Delete first element of L and set the bit for this element in $a[]$ to 0.

MEMBER(i) If $a[i]$ is 1 then i is a member of the set else it is not a member.

5. We can make use of any data structure that supports the following operations: 1) **SEARCH** for an arbitrary element; 2) **INSERT** an arbitrary element; and 3) **DELETE** the minimum. One such data structure is a 2-3 tree. We can use a modified version of any such data structure. Each node in this data structure will be a linked list of identical keys. Sorting, as pointed out in class, amounts to inserting the given keys into the empty data structure and deleting the minimum one by one. When inserting a key, we first check if this key is present in the data structure. If it is, the new key will be inserted as the head of the corresponding list. If it is not, a new node will be created (using the **INSERT** algorithm of the data structure) and a list with one key will be stored in the node.

Note that the data structure will contain at most d nodes at any time. Time for inserting a single element into the data structure is $\log d + c$, where c is a constant time needed to insert an element into linked list. Time for inserting all the elements into the data structure is $O(n \log d)$. Total time for all the deletes is $O(n + d \log d)$. Therefore, the run time of the algorithm is $O(n \log d + n + d \log d) = O(n \log d)$.

6. (a) Similar to the **WeightedUnion** algorithm in book;
- (b) Similar to proof in book; Let T be any tree with m nodes created using **HeightUnion**. Let $Union(k, j)$ be the last union performed. Let the sizes of k and j be $m - a$ and a , respectively. Without loss of generality assume that $1 \leq a \leq \frac{m}{2}$. By induction hypothesis, $height(k) \geq height(j)$. Thus k becomes the root. There are two possibilities.
- $height(k) > height(j)$. In this case, the height of the new tree is the same as $height(k)$. Thus the height of the new tree is $\leq \lfloor \log(m - a) \rfloor + 1 \leq \lfloor \log m \rfloor + 1$.
 - $height(k) = height(j)$. Applying the induction hypothesis, this can happen only if $\log \lfloor a \rfloor = \log \lfloor m - a \rfloor$. This in turn means, $\frac{m}{3} < a \leq \frac{m}{2}$. The height of the new tree is then $\leq \lfloor \log a \rfloor + 1 + 1 \leq \lfloor \log \frac{m}{2} \rfloor + 2 = \lfloor \log m \rfloor + 1$.
- (c) Consider the singleton sets S_1, \dots, S_n . Union S_1 with S_2, S_3 and $S_4 \dots$, giving rise to trees of height 2 each. Now perform union on a pair of these at a time. The result will be trees of height 3 each and so on. (See Figure 2.23 in the text).
7. Find the median M of the n given keys in $O(n)$ time. If X is the given input sequence partition X into X_1 and X_2 such that $X_1 = \{q \in X : q < M\}$ and $X_2 = \{q \in X : q > M\}$. If $|X_1|$ and $|X_2|$ are both even, then M is the unique element. In this case output M and quit. If not, if $|X_1|$ is odd, recursively look for the unique element in X_1 . If none of the above cases applies, recursively look for the unique element in X_2 . Let $T(n)$ be the run time of this algorithm on any input of size n . Then, it takes $O(n)$ time for the initial selection and the partitioning. The time for the recursive call is $T(n/2)$. Therefore, it follows that $T(n) = T(n/2) + O(n)$ which solves to: $T(n) = O(n)$.
8. We can see that the smallest element appears $2n - 1$ times in C ($\min\{k_1, k_1\}, \min\{k_1, k_2\}, \dots, \min\{k_1, k_n\}, \min\{k_2, k_1\}, \min\{k_3, k_1\}, \dots, \min\{k_n, k_1\}$). Similarly, the second smallest element appears $2n - 3$ times in C and the i th smallest element appears $2n - 2i + 1$ times in C . Now the median of C is the j th smallest element such that

$$\sum_{i=1}^j 2n - 2i + 1 = \frac{n^2}{2}$$

j can be obtained by solving the above equation, i.e., $j^2 - 2nj - \frac{n^2}{2} = 0$.

Now, the median of C can be obtained by finding the j th smallest element in S .

Complexity = $O(n)$.

9. Do a radix sort on the elements in A and B separately. Complexity = $O(n)$.
Merge the two sets to check whether the sets are disjoint. Complexity = $O(n)$.
Total complexity = $O(n)$.