Binary Search

Consider a bit string s which is of the form- some (possibly zero) 0s followed by some (possibly zero) 1s. (i.e. “001”,”00” but not “010”)
Now you are given the size of this string (say,n) and are asked to find the position of rightmost ‘0’ in s (if it exists). The string s is hidden from you, however you can ask queries of the type “What is the bit at the ith index” for any 0<=i<n.
Now here one possible approach can be to iterate from i=0 to n and ask what the ith bit is, and whenever you encounter a ‘1’ bit for the first time at some index i, you return i-1. (the cases when there is no ‘0’ or no ‘1’ are special and are expected from the reader to be able to handle separately). This would need O(ans) queries where ans is what you finally return. However this is O(n) in the worst case and even in average case this is O(n/2).
A better approach is to keep a range of indices where our ans would lie and update it as follows:-
Initially the range is l=0,r=n-1 and it holds trivially that our answer lies between l and r. Now, let mid=(l+r)/2 be the index in the middle of this range, we ask for the bit at index=mid, if it is ‘1’, then surely the range [mid,r] is useless and we can reduce our search to [l,mid-1]. If however you have a ‘0’ at index=mid then the range [l,mid-1] is useless and we can reduce our search to [mid,r]. We can repeat this till our range finally converges to the answer.
Note that here after every query the size of our range reduces roughly by half. And hence after O(log(n)) queries our range will converge to the answer! This is essentially what binary search is….halving the search space till you converge to the answer.
There is however a small issue in our above approach when it comes to implementation, when r=l+1 , then mid=l (if you take floor of (l+r)/2) and find that ‘0’ is at the lth position and your search space still remains [mid,r]=[l,r] and isn’t reduced. Hence it can lead to infinite loops if this issue isn’t handled carefully. One possible way of doing that is to stop when r-l=1 and then ask values at indices l and r and answer accordingly. This however makes the implementation a bit messy and often leads to bugs, here is a cleaner implementation which avoids this issue altogether:

```
int ans=-1,l=0,r=n-1;
while(l<=r){
   int mid=(l+r)/2;
   if(s[mid]=='0'){
      ans=mid;
      l=mid+1;
   }
   else r=mid-1;
}
//ans is the final answer
```

Take some time to analyse the above snippet and convince yourself that the search space is always reduced and finally the variable ans stores the answer…you can also determine if there actually exists a ‘0’ or not by checking whether ans is -1 or not in the end.

So, this was all about what binary search is. Any task involving binary search is nothing but finding a function f which is 0 till some point and 1 afterwards and then solving for the leftmost 0 or rightmost 1. In most problems however, this function is not explicitly given and most of the creativity involved in binary search is in figuring out an appropriate f (whose

rightmost 0 or leftmost 1 is the answer to the problem), or even in deciding that the given problem requires binary search! (atleast in the beginning, when you aren't experienced enough) Consider for example the following problem-
There are n machines ,ith of which takes time ti seconds to produce one product. You have to produce k products and you are allowed to use multiple machines simultaneously. Find the minimum time you would require to produce k products. (n upto 1e5, ti upto 1e9, k upto 1e9)
Here if I want to binary search on the answer, I would want to be able to (sufficiently efficiently) find out, for any time t, whether t seconds are enough to produce k products. Call this function f(t). Clearly f is monotonic, (i.e. false till some point and then true everywhere after that point). Now to evaluate f(t), one possible approach can be to find the maximum number of products that can be produced in time t(which is sort of an inverse problem of our original problem) and if that number is >=k return true else return false. And this number can be found out by making every machine work to its full potential in t seconds and thus producing floor(t/ti) products for all 1<=i<=n. Hence f(t) is true if summation(floor(t/ti))>=k and false otherwise. This summation can be evaluated in O(n) and hence f(t) can be evaluated in O(n). Rest is straight forward, just find crude lower and upper bounds for the minimum time required (for example, l=1 and r=k*max(ti) or even r=1e18 works)... note that here r=floor(k/n)*max(ti), l=min(ti) are better bounds but since log is a pretty slowly growing function, even the former choices would pass comfortably. Remaining is just binary searching over the space [l,r] for the answer which can be done in O(log(r-l)) steps and since each step does O(n) computation, overall time complexity becomes O(nlog(r-l)) which is enough to pass in a few hundred milliseconds.
Note that here the fact that we are able to evaluate f(t) sufficiently fastly was also important since if otherwise we were evaluating f(t) in say, O(n^2), our solution would still not have passed.
In conclusion, binary search problems are mostly either of interactive kind (like our first example, where it seems directly from the constraints on number of queries that can be asked,that some sort of binary search is required) or of the non-interactive kind (like our second example) where observing that binary search can be helpful is not that direct. In such problems binary search is helpful if the inverse yes/no question (like whether k products can be made in t seconds) is monotonic and is easy and efficient enough to answer!
Problems of varying difficulty are frequently asked for both kinds!

Problems:

1. Factory Machines
2. Array Division
3. Magic Powder - 2
4. Cellular Network
5. Schedule Management
6. Fixed Point Guessing
7. Guess the K-th Zero (Easy version) (we leave the hard version for one of the later weeks)
8. Guessing the Greatest (easy version) (try the hard version too! or atleast look at its editorial)

9. Binary Search
10. Bear and House Queries
11. Kayaks
12. Multiplication Table
13. Doremy's Perfect DS Class (Medium Version) (do not let the problem rating deter you, you actually know enough to solve this problem!, also if you are able to solve this, try the hard version….or atleast look at its editorial!)