# Binary Search[1]

In this lecture we look at an extremely powerful idea of speeding up algorithms, and also use it to introduce **time analysis of recursive algorithms**. The idea is called "binary search".

- The canonical problem solved by this technique is the following. The input to this problem is a sorted list $A$ and a "search" item $x$. We assume the entries of $A$ and $x$ are both integers[2]. Our goal is simple: decide if $x \in A$ or not. One could consider variants where in the case $x \in A$ you also have to return the location, but let's focus on this simpler "decision" problem. For the time being, let's still assume we are interested in the *number* of comparisons we make (everything else is free).

  If the list $A$ was not sorted, then $x$ needs to be compared with every element of the list leading to $n$ comparisons. Why? The main idea is if you didn't compare $x$ with some $A[i]$ then perhaps the "adversary" put in $x$ there. This is not a proof by any means, but hopefully gives you an idea why $n$ may not be avoidable. We now see if $A$ is sorted we can do *way better*.

- The main idea of binary search is to "quickly halve the search space". What do we mean? In the beginning, if $x \in A$, then it has $n$ possible locations. We want to do very few comparisons (as it turns out, it will be 1 comparison), and reduce the possibilities by a *constant fraction* (as it turns out, this will be $1/2$). Which comparison does this? This part is creative, but it's one of those things which is hard to "unforget". We compare $x$ with the mid-point $A[n/2]$. If $x = A[n/2]$, then we are done, and can say YES. If $x < A[n/2]$, then[3] ***using the fact that $A$ is sorted***, we can say $x < A[j]$ for $j \geq n/2$. And so, if $x \in A$ it must be in the first "half", that is, in $A[1 : n/2-1]$. So we reach the same problem except the new list is much smaller. We **recurse**. If $x > A[n/2]$, then similar reasoning: we recurse on the other half. Done!

  Well, we need to take care of the base case. If $A$ had no elements, we say NO. If $A$ had only one element, then we just check if $x$ is that element and say YES or NO accordingly. One can be smarter with these base cases. Doesn't matter much.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 5th Jan, 2024*
*These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!*

[2] but all we need is that there is some total order among them

[3] and here we are using that there is a "total ordering", that is, any two elements can be compared

```
1:  procedure RecBinS(A, x):
2:      n ← len(A). ▷ note, A is indexed 1 to n
3:      if n = 0 then:
4:          return NO
5:      if n = 1 then:
6:          if x = A[1] then:
7:              return YES
8:          else:
9:              return NO
10:     else:
11:         m ← ⌈n/2⌉.
12:         ▷ Compare A[m] and x.
13:         if A[m] = x then:
14:             return YES
15:         else if x < A[m] then:
16:             return RecBinS(A[1 : m − 1], x)
17:         else:▷ That is, x > A[m]
18:             return RecBinS(A[m + 1 : n], x)
```

- **Correctness.** I hope all of you can see why it is a correct algorithm. Clearly we say YES only if we find it. All we need to show is that if $x \in A$, we do say YES. This can be proved by induction on the size of $A$. See here, for instance. Let's understand the running time of this algorithm.

- **Running Time.** Let's set up the definition of the function we are interested in. Here it is

$$T(n) \quad := \quad \max_{A, |A| \leq n, x} \text{Number of comparisons made by RecBinS on } (A, x) \tag{1}$$

  That is, $T(n)$ is defined to be the maximum number of comparisons RecBinS makes when run on a list $A$ and target $x$, where the list has $\leq n$ elements. Note that we moved from $= n$ elements to $\leq n$ elements; this is just something which makes life easier, because we can say $T(m) \leq T(n)$ for all $m \leq n$. Comeback to this later if you are confused.

  Let us see what can we say about this function. First we note the following.

$$T(0) = 0 \quad \text{and} \quad T(1) = 1 \tag{Base Cases}$$

  This is because when $n = 0$, we satisfy Line 3 and say NO without making any comps, while if $n = 1$, the code in Line 5 makes 1 comparison. What if $n \geq 2$?

- Then things get more interesting. The if statement at Line 13 makes 1 comparison. If this comparison is a equality, then we end. But recall we are in worst-case land, so we assume this is not a equality. Well then, we either run Line 16 or Line 18. But how long do they take? They are *recursive* calls!

  This is where *worst-case* analysis will be our friend. Let's consider Line 16. It's a recursive call of RecBinS on an array of length $m - 1$ (with $x$). We can't say *exactly* how long it takes, but we can say this time is ***at most*** $T(m - 1)$. This is because of how $T(m - 1)$ is defined: it already has the "worst-case" baked in, and we cannot take more time than the worst case. Let this sink in if you are

seeing this for the first time. Similarly, Line 18 takes $T(n-m)$ time. One of the two cases will occur, we don't know which, but we can definitely say the time taken is at most $\max(T(m-1), T(n-m))$. Since $m = \lceil n/2 \rceil$, both of these are $\leq \lfloor n/2 \rfloor$. And thus, using the fact that $T$ was defined to have monotonicity, we see that the time taken for either of the cases is at most $T(\lfloor n/2 \rfloor)$.

In summary, we can say that:

$$\text{For all } n \geq 2, \quad T(n) \leq 1 + T(\lfloor n/2 \rfloor) \qquad \text{(Recursive Ineq)}$$

The Equation (Base Cases) and Equation (Recursive Ineq) together form what is called a **recurrence** inequality, and the solution to this will lead us to our answer.

- **Solving Recurrence Inequalities.** We will see a very general theorem which solves recurrence inequalities, but let's use the one above to give the general idea of how it is solved. I like to call this the "kitty" method; this is not a universal terminology. The reasoning goes something like this.

  We want to figure out $T(n)$. We imagine a ball of "size" $n$. If this ball's size is 1 or 0, then we immediately know how much it "costs" using Equation (Base Cases); in this recurrence above, $n = 0$ costs 0 and $n = 1$ costs 1.

  We interpret (Recursive Ineq) as follows. We think of the "1" in the RHS as a cost you can pay and put it in a collection (or a kitty), and once you do this, the ball of size $n$ "shatters" and becomes a ball of size $\lfloor n/2 \rfloor$. Then we put another 1 in the kitty, and off it shatters and becomes $\lfloor \lfloor n/2 \rfloor / 2 \rfloor$. And so on, till we get to a ball of size which can be covered by the base case. And when we reach that, we pay all the money in the kitty plus money for the base case. All this is just an evocative way of describing the following inequalities:

$$\begin{aligned}
T(n) &\leq T(\lfloor n/2 \rfloor) + 1 \\
&\leq (T(\lfloor \lfloor n/2 \rfloor / 2 \rfloor) + 1) + 1 \\
&= T(\lfloor \lfloor n/2 \rfloor / 2 \rfloor) + \underbrace{(1+1)}_{\text{kitty}} \\
&\leq (T(\lfloor \lfloor \lfloor n/2 \rfloor / 2 \rfloor / 2 \rfloor) \ + \ 1) + (1+1) \\
&\ \ \vdots \\
&\leq T(1) + \underbrace{(1 + \cdots + 1)}_{\text{stuff in kitty at the end}}
\end{aligned}$$

- For simplicity, let us assume $n$ was a power of 2 so that all flooring business doesn't bother us. Say $n = 2^k$. Then, let's see how many times we need to "shatter" this ball till it gets to size 1: a little thought shows you the answer is $k$ times. And so, the value of $T(2^k) = k + 1$, where the $+1$ is due to the fact that the size 1 ball costs 1 unit.

- What about a general $n$? Let's do an example here; say $n = 23$. What happens to the balls in our evocative picture. Well, a size 23 ball becomes size 11 then it becomes size 5 then size 2 and then size 1. So it puts 4 in the kitty, and then another 1 for the base case; gives $T(23) \leq 5$.

  Let us now write these numbers in binary[4] notation. We see that $23 = (10111)_2$ and $11 = (1011)_2$ and $5 = (101)_2$ and $2 = (10)_2$ and $1 = (1)_2$. Do you see what's going on? As we go left to right, the last bit is just lopped off; this is the *right-shift* operation.

[4] perhaps explaining why it's called *binary* search

So, we conclude that for any $n$, the number of smashings till we get to 1 is precisely the number of bits required to describe $n$ in binary minus 1. When we add the $+1$ from the basecase, we get

$$T(n) \leq \text{Number of bits to write } n \text{ in binary } \quad = \lceil \log_2(n+1) \rceil$$

- **Binary Search is more than just Searching in a Sorted Array.** Let me end this lecture by saying that although "binary search" is used to imply the algorithm that we saw above, it also often used to denote the idea of "quickly halving ones search space". Here is another example which is solved using binary search which has no "sorted arrays" explicitly given.

- *Group Testing.* You have collected $n$ blood samples from soldiers and you know one of these is contaminated. You have a very expensive test which can take in a mixture of blood and detect contamination. You need to find the contaminated sample with as few tests, where in each test you will take some portions of each sample and feed the mixture to the test. How few samples do you need? The above problem is a special case of the "real life" problem from the late 1940's, and is now called the *group testing* problem.

  Here is the "binary search" algorithm. We take the first half of samples. That is, $S := \{1, 2, \ldots, n/2\}$ and run the test on it. If the test succeeds, that is, tells there is a contaminated sample in $S$, then we recurse on this set $S$. Otherwise, we know that the contaminated sample lies in $T := [n] \setminus S$, and so we recurse on this set $T$. Each of these sets are of size (roughly) $n/2$, and thus we halved our search space with one test. Of course when $n = 1$, we know that this must be the contaminated sample, and we can just return it. Thus, all in all, the number of tests follows almost[5] the same recurrence as the one above for binary search, implying, roughly $\log_2 n$ samples are all we need.

  **Exercise:** *Suppose you knew that there are $k$ contaminated samples. How many tests can you use to find* **all** *of these? I hope with a little thought you see you can do this in roughly $k \log_2 n$ tests. Can you do better?*

[5] the difference is that instead of $\lfloor n/2 \rfloor$ we will have $\lceil n/2 \rceil$ in the RHS of Equation (Recursive Ineq). Can you solve this recurrence exactly for general $n$?