Applications of Binary Search

(Prepared by the UCF Programming Team Coaches for the Developmental Teams.)

- When table is sorted, we can do binary search.
- When table is not sorted, can't take advantage of it.
- Can do binary search when the search space is sorted.
- This is especially useful in situations where you can calculate an increasing function forwards easily, but have difficulty calculating its inverse directly.

Problem #1: Big Boxes (2020 UBC Programming Contest)

Brandon has n items that he needs to pack into k big boxes. The n items are currently arranged in a row, and Brandon doesn't want to bother reordering them, so he will partition the n items into k groups of consecutive items and put each of the k groups into their own box. For convenience when moving, Brandon wants to minimize the weight of the heaviest box.

Input: Two integers n and k ($1 \le k \le n \le 10^5$); followed by n integers w_i ($1 \le w_i \le 10^4$), representing the weight of each item in order.

Output: A single integer denoting the minimum possible weight of the heaviest box.

Do a Binary Search on the weight of the box.

Problem #2: Crystal Etching (2006 UCF Locals)

$$\frac{f_2 - f_1}{f_1 f_2} = at + b(1 - e^{-ct})$$

Calculate how many seconds a crystal should be "etched" until it arrives at a given frequency.

initial frequency f_1, target frequency f_2, constants a, b, c

The only unknown in this formula is t, the number of seconds for which the crystal must be etched.

The difficulty with this problem is solving the equation for t. No matter what you try, it's difficult to only get one copy of t in the equation, since t appears in both an exponent and a linear term.

As *t* rises, the value on the right-hand side of the equation also rises. In particular, since the constants a, b and c are always positive, that function on the right is a strictly increasing function in terms of *t*.

Do a Binary Search on *t*.

Problem #3: A Careful Approach (2009 World Finals)

Given 2 to 8 planes;
Each plane has a time window to land;
Schedule the landings so that the minimum time gap between successive landings is as large as possible.
Example:
$Plane_1$: 0 to 10, $Plane_2$: 5 to 15, $Plane_3$: 10 to 15,
$Plane_1$ could land at t = 0, $Plane_2$ at t = 7.5, $Plane_3$ at t = 15.
If $Plane_2$ moves its time any earlier, then the gap between $Plane_1$ and $Plane_2$ gets below 7.5 and if it moves its time later, then the gap between $Plane_2$ and $Plane_3$ goes below 7.5. Thus, 7.5 is the largest gap we can guarantee between each of the planes.

Two Problem Simplifications

First, assume we know the order for landing:

Try all orderings of the planes landing!

Second, write a function that, given an ordering of the planes and a gap value, returns true if that gap is achievable.

The function (greedy algorithm):
(1) Make the first plane land as early as possible.
(2) Make the next plane land exactly gap minutes later (if this time is within its range); if it is not, then make it land after that time, as soon as possible. If this can't be done, then the arrangement is impossible. If it can, then continue landing planes.
Now, using this function, we can do a Binary Search on gap.

Problem #4: Bones's Battery (2013 Pacific Northwest)

(Binary Search with APSP, All Pairs Shortest Paths)

Electric shuttle going from school to school;

Can charge at schools only;

Can charge at most K times;

Find the necessary range R for the shuttle.

$2 \le N \le 100$ and $1 \le K \le 100$ and $1 \le d \le 10^9$

Example:

Let's say we have a graph with 4 nodes, 4 edges, and K = 2

A↔B, d = 100

B↔C, d = 200

C↔D, d = 300

D↔A, d = 400

So, we have a graph that goes in a circle.

If we leave from node A we need to use a charge so we only have 1 more charge left and we can only use a charge at a node.

If R = 300, we can go from node A to B, C without having to use another charge and then at node C we can use a charge to go to C↔D since that only cost 300;

therefore A ↔ D in 2 charges.

If R = 200, we would only be able to go to node A↔B before we had to use a charge to go to node C. But then we don't have enough gas or charges to go to node D from C.

Solution:

1. Run "modified" Floyd-Warshall to remember the shortest path for every pair of vertices (i.e., remember the path for each pair and not just the min distance).
2. Do a Binary Search on R (with the given K) to find R.

Problem #5: Reconnaissance (2014 NAIPC)
(Ternary Search)

Given *n* cars with their starting x position, speed, and direction, find the minimum distance that will cover all of the cars at some time.
The first observation you can make is that the min distance is clearly the car on the far left and the far right at the given moment of time regardless of how many cars you have.

Let's consider just two cars; there are five cases:

(1) Both cars are going in the same direction with the same speed which means their distance will always stay the same.
(2) Both cars going in the same direction, the car in front going faster.
(3) Both cars going in the same direction, the car in back going faster.
(4) One car is on the left going left and the other car is on the right going right. As time progresses their distance also increases. Therefore, the minimum answer is at time 0 before they started to move.
(5) The cars' directions are towards each other. This means for some time the distance gap is going down and eventually reaches 0 and then as they pass each other this case reverts to the fourth case.

If the distance was graphed as a time/distance graph we will notice that the fifth case resembles a parabola. Here the answer would be the bottom point of a parabola, which in the case of 2 cars would just be zero.

But, of course, the problem is not as simple and there are *n* cars. What is important though is that regardless of how many cars there are if we were to graph the time/distance it would still resemble a parabola.

Using ternary search, we can find the bottom of this parabola and our answer:

```
for (int i = 0; i < 100; i++) {
      double lowThird  = (2 * low + high) / 3;
      double highThird = (low + 2 * high) / 3;
      // Update the appropriate bound
      if (range(lowThird) > range(highThird))
            low = lowThird;
      else
            high = highThird;
}
```

Do a Ternary Search on t:

Find location of each car for t = lowThird; then

range(lowThird) = maxLocation – minLocation

Find location of each car for t = highThird; then

range(highThird) = maxLocation – minLocation

low =

high =