ON A MISSION TO MAKE YOU LOVE DSA

CRASH CAMP

BINARY SEARCH

Episode 03

ADDED FEW MORE PROBLEMS

AND
MANY MORE...

Index

Starting few slides are beginner oriented but will definitely give some good insights even if you already know 'Binary Search'

Let's begin the journey

Linear Search

Introduction

- You are a given a sorted array nums[] & an int k,

  nums[] = {1,2,4,6,8,10,15} k = 15
- return true if k exists else false.

Now, how would you approach above problem...

A "linear search" (a loop)

- iterating from i = 0 to i = n-1 (n = size)
- if(nums[i] == k) return true
- if(i == n) return false K doesn't exist & you checked all values

Using 'linear search', in worst case you would scan all 'n' (7) elements.

Can we do something better

let's jump to idx 2

0 1 2 3 4 5 6

[1 2 4 6 8 10 15]   nums[2] < 15

these value idx[0,1] are also less than 15, so need to check here

now this is our potential search space

By jumping to idx = 2, we avoided 2 elements, so instead of all **7** elements we have only **5** , which is better than linear search

Sorted Search Space

- We saw if search space is sorted, jumping to some idx is better than linear search.
- What should be that idx value.. let's see.

say your search space had 100 sorted values, check if k = 100 exist

1 2 .. 10 .... 99 100

assume you jump to idx 10, which divides array into 2 parts.

left array (10%) [1..10]
right array (90%) [10..100]

although jumping to idx = 10 is better than 'linear search' but still in 'worst case' you have 90% space

can we reduce this search space in worst case even more ?

if we jump to 'mid' value it divides

left array (50%) [1..50]
right array (50%) [50..100]

now in worst case only 50% space is left

now we can conclude, if our search space is 'sorted' we will always jump to 'mid' index

getting something new, leave a like

Dividing Search Space

- You are given a sorted nums[] & an integer k
- return largest value less than k.

i/p

nums[] = {0,1,2,6,7,8,9} k = 6

o/p

2

```
        0 1 2  3 4 5 6         this is our ans
       [0 1 2  6 7 8 9]
```

If you are at any of these indices you will move to 'right' — values < 6

If you are at any of these indices you will move to 'left' — values >= 6

- Given condition, we can divide search space in 2 parts
- Depending on which part we are, we move 'left' or 'right'.

Let's name these 2 space as

1) Favourable space (F) where your ans may lie
2) Unfavourable space (U) where ans will never lie

We want larget value 'less' than k

so values < K are (F)
values >= K are (U)

```
    ↓ ans
[0 1 2 6 7 8 9]
[F F F U U U U]
```

If you are at F move to 'right'
else move to 'left'

we conclude, if space is sorted, we can divide search space in F & U, depending on which space we are, we either move 'right' or 'left' .

Few Conclusions

- Till now we concluded, if given space is sorted
  1) jump to 'mid' idx (reducing no. of comparisons)
  2) divide search space in Favorable (F) & Unfavorable (U) space to choose which part of space you would move (right or left)

- You are given a sorted nums[] & an integer k
- return largest value less than k.

i/p

nums[] = {0,1,2,6,7,8,9} k = 6

o/p

2

l = 0 (lower limit)
h = 6 (higher limit)

our search space will lie b/w 'l' & 'h'

we calculate mid as

```
mid = (l+h) / 2
```

1

[0 1 2 6 7 8 9]
[F F F U U U U]

we divide search space with following

```
if(nums[mid] < k) {
    l = mid + 1;
} else {
    h = mid - 1;
}
```

2

we are at 'F', move 'right'
to move 'right' just push l to right of 'mid'

we are at 'U', move 'left'
to move 'left' just push h to left of 'mid'

'l' & 'h' are moving towards each-other, till what point they will chase (initially l < h)...

```
while(l <= h)
```

3

when l becomes > h, it indicates we exhausted our search space

note-> division of space in 'F' & 'U' will always depend on the problem

The Algorithm

- Using 3 points given in prev. slide let's construct an algo

```
int l = 0;
int h = n-1;
while(l <= h) {
  int mid = (l+h) / 2;
  if(nums[mid] < k) {
      l = mid + 1;
  } else {
      h = mid - 1;
  }
}
return h;
```

What is significance of 'return h' ?
will reveal the secret behind it... (my secret trick)

Now this algo is what we call 'Binary Search'

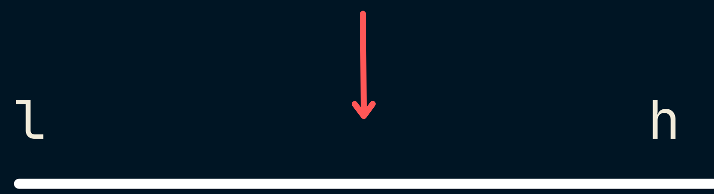
From now on, will you be able to write 'binary search' easily ?

let me know in comments...

Ways to calulate mid

- There are many ways-

1) mid = (l+h) / 2

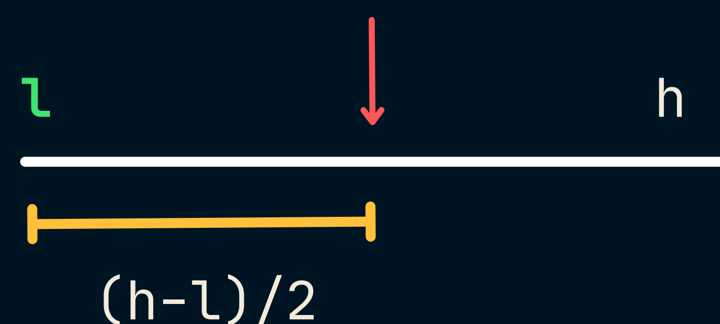


many lang. have varibale limits c/c++ has 2147483647 (int)

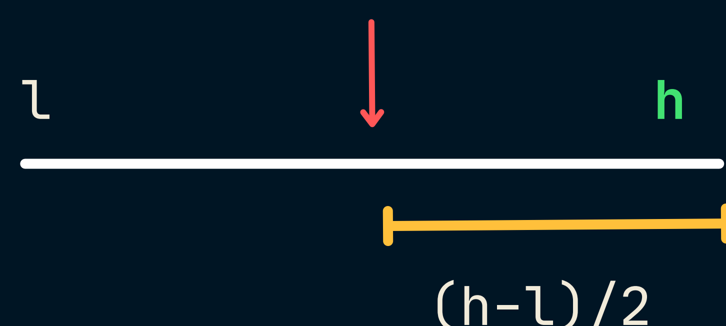
now, if both l & h are INT_MAX so l+h will cause 'overflow', so only use this way to get mid, if constraint are small.

we have other ways which also take care of 'overflow'

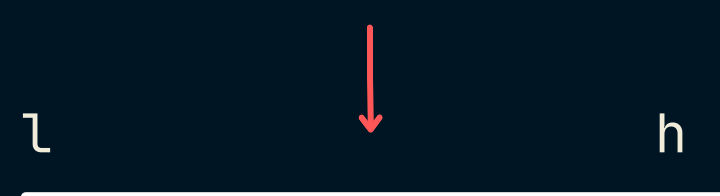
2) mid = l + (h-l)/2



3) mid = h - (h-l)/2



4) mid = (l + h) >> 1



>> is a right shift operator which is equivalent of (divide by 2)

Secret Trick of l & h

- Earlier we decided if search space is sorted we will divide it in 2 parts 'F' & 'U'

  Favorable   (F)
  Unfavorable (U)

- Assume for some problem 1st part is 'F' & 2nd is 'U' (trick will work even if it's vice-versa).

```
        [F F F F U U U U U]
      l                      h
```

- all 'F's form 1 space & all 'U's form another

- When we start 'Binary Search'
  - l is at 1st value of 1st space
  - h is at last value of 2nd space

- 'l' will always move towards 2nd space
  (till it doesn't crosses 1st space)
- 'h' will always move towards 1st space
  (till it doesn't crosses 2nd space)

```
        [F F F F U U U U U]
             h     l
```

here l > h,so Binary Search terminates

- When 'Binary Search' ends (while loop terminates)
  - l is at 1st value of 2nd space
  - h is at last value of 1st space

more than 90% times our answer will be given by either 'l' or 'h' when 'Binary Search ends'

getting some hint why we only wrote 'return h' someslides back ?

Problem Types

This will be our generic template & more than 90% problems will be solved just with minor tweaks in it.

```
int l = 0;
int h = n-1;
while(l <= h) {
  int mid = (l+h) / 2;
  if( nums[mid] < k ) {
      l = mid + 1;
  } else {
      h = mid - 1;
  }
}
return h;
```

This 'if()' decides whether we are in 'F' or 'U' & accordingly move to 'left' or 'right'

(refer point 2 in slide 5)

Now depending on what goes inside that 'if()' we will categorize Binary Search in 2 types

- Type 1
- Type 2

Type 1

simple values will decide whether to enter if or else.
ex- nums[mid] > k, mid*mid < k etc...

Type 2

Inside if() we will call a function whose result will evaluate to either 'true' or 'false'

Type 2 is sometimes referred to as Binary Search on answer

More Insights

When to use Binary Search ?

If your search space is sorted & you can apply a linear search this is the intuition to go for Binary Search

How to use Binary Search ?

1) Divide the search space into 2 parts 'F' & 'U'
   ->To divide search space you need to figure out what goes inside that if()
2) Figure out who gives you answer 'l' or 'h'

I bet almost 90% of Binary Search problems will be solved using above 2 steps + that basic template

now let's solve some problems ...

Problems

Problem List

1. UpperBound
2. LowerBound
3. Sqrt(x)
4. Valid Perfect Square
5. Find the smallest letter greater than target
6. Search Insert Position
7. Valid Triangle Number
8. Arranging Coins
9. Split Array Largest Sum
10. Capacity to ship packages within D days
11. Koko eating bananas
12. Allocate minimum number of pages
13. Aggressive Cows
14. Nth magical number
15. Minimum Time to Complete Trips

1st 6 are type 1 problems which don't require much observations while remaining are type 2, so they are a bit challenging

UpperBound

Description

Given a sorted array nums & an integer k, return index of smallest value greater than k

i/p	o/p
nums = [1,2,3,3,4,5] k = 3	4

Why to use Binary Search ?

Space is sorted + you can apply 'linear search'

To use 'Binary Search' we need to divide the search space in 2 parts 'F' & 'U'

We are asked smallest value greater than k

```
so nums[i] >  k -> 'F'
   nums[i] <= k -> 'U'
```

```
       0 1 2 3 4 5
      [1 2 3 3 4 5]
l ->   U U U U F F  <- h
               ^
               this is our answer
```

1 Our ans is 1st element of 2nd space, so whenever your mid is at 'U' move 'right' & at 'F' move 'left'.

2 When Binary Search ends who points to 1st element of 2nd space ... l (refer slide 9)

UpperBound

```
int upperBound(vector<int>& nums, int k) {

    int l = 0;
    int h = nums.size() - 1;

    while(l <= h) {

        int mid = l + (h-l) / 2;

        if(nums[mid] > k) {
            h = mid - 1;      <-----  we are in 'F' so
                                      move left
        } else {
            l = mid + 1;      <-----  we are in 'U' so
                                      move right
        }
    }

    return l;  <-----  return l

}
```

Getting the idea...

we just took care of 2 things

1) Which is of 'Favorable' or 'Unfavorable' space.
2) Who gives us ans 'l' or 'h'

LowerBound

Description

Given a sorted array nums & an integer k, return index of first element not less than k

i/p

nums = [1,2,3,3,4,5]
k = 3

o/p

2

You know why to use Binary Search... right ?

Let's divide search space in 2 parts 'F' & 'U'

We are asked 1st value not less than or greater than equal to k

```
so nums[i] >= k -> 'F'
   nums[i] <  k -> 'U'
```

```
     0 1 2 3 4 5
    [1 2 3 3 4 5]
l    U U F F F F    h
```

this is our answer

1 ans is 1st ele of 2nd space,so if mid is at 'U' move 'right' (l = mid+1) & at 'F' move 'left' (h = mid - 1)

2 When Binary Search ends who points to 1st element of 2nd space ... l

(refer slide 9)

LowerBound

```
int lowerBound(vector<int>& nums, int k) {

    int l = 0;
    int h = nums.size() - 1;

    while(l <= h) {

        int mid = l + (h-l) / 2;

        if(nums[mid] >= k) {
            h = mid - 1;
        } else {
            l = mid + 1;
        }
    }

    return l;

}
```

we are in 'F' so move left

we are in 'U' so move right

return l

Getting the idea...

we just took care of 2 things

1) Which is of 'Favorable' or 'Unfavorable' space.
2) Who gives us ans 'l' or 'h'

Sqrt(x)

Description

Given a non-negative integer x, compute and return the square root of x.(return only integer part)

i/p	o/p
x = 4	2
x = 8	2 it should be 2.82 but only int part, so 2

note-> your are not allowed to use any inbuilt method for calculating power or sqrt

Brute Force

consider below number line-

x = 15
o/p = 3

0 1 2 3 4 5 6 7 8 ... 15

- iterate from i = 0 till i*i <= x
- keep updating variable ans
- return ans

i	i*i	comment	ans	x
0	0*0=0	0<15,i++	0	15
1	1*1=1	1<15,i++	1	15
2	2*2=4	4<15,i++	2	15
3	3*3=9	9<15,i++	3	15
4	4*4=16	16>15,stop & return ans = 3		

can you see, we are iterating over the number line & the number line is sorted

sorted space + linear search,
go for Binary Search

Sqrt(x)

x   = 15
o/p = 3   0 1 2 3 4 5 6 7 8 ... 15

- We concluded to go for Binary Search, but we need to divide
  1) Search space in 2 parts ('F' & 'U')
  2) Decide whether 'l' or 'h' gives ans.

Dividing 'Search Space'

- we were iterating till i*i <= k
  so all i for which i*i <= k are 'F'
  i*i > k are 'U'

x   = 15
o/p = 3

l 0 1 2 3 4 5 6 7 8 ... 15 h

F F F F U U U U U... U

this is our ans

1 • ans is last value of 1st space, so when mid is at 'F' move 'right' (l=mid+1) else move 'left' (h=mid-1)

2 • when 'Binary Search' ends who points to last value of 1st space...h (refer slide 9)

Sqrt(x)

```
int mySqrt(int x) {

        long long l = 0;
        long long h = x;

        while(l <= h)
        {
            long long mid = l + (h-l)/2;

             if(mid*mid > x) {
                h = mid-1;
             } else {
                l = mid+1;
             }
        }
        return h;
    }
```

Valid Perfect Squares

Description

Given a positive integer num, write a function which returns True if num is a perfect square else False.

i/p	o/p
x = 4	true
x = 8	false

note-> your are not allowed to use any inbuilt method
for calculating power or sqrt

Brute Force

consider below number line-

x = 10
o/p = false

0 1 2 3 4 5 6 7 8 ... 10

- iterate from i = 0 till i*i <= x
- for some i, if(i*i == x) -> return true
- else if i*i > x return false

i	i*i	comment	x
0	0*0=0	0<10,i++	10
1	1*1=1	1<10,i++	10
2	2*2=4	4<10,i++	10
3	3*3=9	9<10,i++	10
4	4*4=16	16>10,stop & return false	

because if we further increase i, i*i will only increase than x, so no i*i==x hence 10 is not a valid square

can you see, we are iterating over the number line & the number line is sorted

sorted space + linear search,
go for Binary Search

Valid Perfect Squares

```
x   = 10
o/p = false   0  1  2  3  4  5  6  7  8 ... 10
```

- We concluded to go for Binary Search, but we need to divide
  1) Search space in 2 parts ('F' & 'U')
  2) Decide whether 'l' or 'h' gives ans.

Dividing 'Search Space'

- we were iterating till i*i <= k
  so all i for which i*i <= k are 'F'
  i*i > k are 'U'

```
x   = 10
o/p = false
```

```
l   0 1  2  3  4  5  6  7  8 ... 15   h
    F F  F  F  U  U  U  U  U... U
```

this is last value for which we will check i*i <= 10 as after this all i*i are > 10, so if this final i*i of 1st space = 10 we return true else false

1 • ans is given by last value of 1st space, so when mid is at 'F' move 'right' (l=mid+1) else move 'left' (h=mid-1)

2 • when 'Binary Search' ends who points to last value of 1st space... h (refer slide 9)

3 • if h*h == x -> return true, else return false

Valid Perfect Squares

```
class Solution {
public:
    bool isPerfectSquare(int num) {


        int l = 0;
        int h = num;

         while(l <= h){

             long long mid = l + (h-l)/2;

             if(mid*mid > num) {
                 h = mid-1;
             }
             else {
                 l = mid+1;
             }
         }

         return h*h == num;

    }
};
```

Find Smallest Letter Greater Than Target

Description

Given a characters array letters that is sorted in non-decreasing order and a character target, return the smallest character in the array that is larger than target.

Note that the letters wrap around.

- For example, if target == 'z' and letters == ['a', 'b'], the answer is 'a'.
  or we can say, if we there is no greater element then return first element.

i/p	o/p
letters = ["a","c","f","j","k","l"], target = "c"	"f"

Brute Force

idea is simple, iterate from i = 0 to last idx, as soon as letters[i] > target, return letters[i]

i	letters[i]	comment
0	'a'	'a' <= 'c', i++
1	'c'	'c' <= 'c', i++
2	'f'	'f' > 'c' stop & return 'f'

can you see, we are iterating over the letters[] which is sorted

sorted space + linear search,
go for Binary Search

Find Smallest Letter Greater Than Target

- We concluded to go for Binary Search, but we need to divide
  1) Search space in 2 parts ('F' & 'U')
  2) Decide whether 'l' or 'h' gives ans.

Dividing 'Search Space'

- we were iterating till letters[i] <= target & we are returning letters[i+1] (as we want smallest greater element)

  so all i for which
  letters[i] <= target are 'U'
  letters[i] > target are 'F'

- (as our ans won't lie in this space so we called it 'Unfavorable')

target = 'c'

```
l   a  c  f  j  k  l  h
    U  U  F  F  F  F
```

this is the smallest greater value than c & it is given by 1st value of 2nd space.

1 • ans is given by first value of 2nd space, so when mid is at 'U' move 'right' (l=mid+1) else move 'left' (h=mid-1)

2 • when 'Binary Search' ends who points to first value of 2nd space... l (refer slide 9)

3 • there may be chance that l = lastIdx+1 (when there is no greater ele than target in that case return letters[0] as stated in problem statement)

Find Smallest Letter Greater Than Target

Let's elaborate point #3 further

i/p   o/p

letters = ["a","b","c","d"],   "a"
target = "e"

as there is no ele. greater than 'e',
so our ans is letters[0] i.e 'a'

target = "e"

```
l     a  b  c  d     h
      U  U  U  U
```

as our ans will be given by 1st ele of 2nd space(which is pointed by l when B.S ends) bt in above problem there is no 2nd space, so finally l will overtake h, thus l = lastIdx+1(size of letters)

so we will check finally
if(l < letters.size()) -> return letters[l]
else return letters[0]

Find Smallest Letter Greater Than Target

```
int nextGre(vector<char>& letters,char target){

    int n = letters.size();
    int l = 0;
    int h = n-1;

    while(l <= h){
        int mid = l + (h-l)/2;
        if(letters[mid] > target) h = mid-1;
        else l = mid+1;
    }
    return l;
}

class Solution {
public:
    char nextGreatestLetter(vector<char>& letters, char target) {

    int t = nextGre(letters, target);


    if(t < letters.size()) {
        return letters[t];
    } else{
        return letters[0];
    }
 }

};
```

if tells us we are in 'F'

else tells us we are in 'U'

there is no ele greater than target, return letters[0]

Now it's Hero Time

Till now you should be able to realize on some easy-medium problems whether to go for Binary Search or not

Now it's **HERO TIME**

Let's solve some of the toughest Leetcode problems having lowest accuracy (many have tried but very few solutions got accepted).

I bet, with the tricks of l & h, these problems should be cake-walk for you guys.

✔	878	Nth Magical Number	35.8%	Hard
✔	2187	Minimum Time to Complete Trips	29.1%	Medium

Minimum Time to Complete Trips

Description

- You are given an array time where time[i] denotes the time taken by the ith bus to complete one trip.
- Each bus can make multiple trips successively; that is, the next trip can start immediately after completing the current trip.
- You are also given an integer totalTrips, which denotes the number of trips all buses should make in total. Return the minimum time required for all buses to complete at least totalTrips trips.

i/p	o/p
time = [1,2,3], totalTrips = 5	3

Brute Force

- So you want min. time in which total of 5 trips can be made (combining all the buses)
- Let's say that min. time is t, so bus1 may have done 3 trips, bus2 has 1 (in time t) & so on...
- So by combining total trips of all the buses we want the min. time in which totalTrips = 5 can be made

Since we want to know that is there a time 't' for which all buses(combined) can make atleast given totalTrips so obviously we need to make a function for that.

Brute Force

assume you made a function canCompleteTrips()
where you pass a time t (1,2,3 etc) & it returns 'true'
if in given time t totalTrips = 5 can be made else false

i/p	o/p
time = [1,2,3], totalTrips = 5	3

In above test case

- bus1 takes 1 sec for a trip
- bus2 takes 2 sec bus3 takes 3 sec

So in time = t, how many trips each bus will make...

totalTripsBus1 = givenTime(t) / bus1TripTime

let's take t = 5, so in 5 sec. how many trips each bus will make...

```
totalTripsBusN = givenTime(t) / busNTripTime

totalTripsBus1 = 5 / 1
               = 5
totalTripsBus2 = 5 / 2
               = 2
totalTripsBus3 = 5 / 3
               = 1
totalTripsAllBuses = totalTripsBus1 + totalTripsBus2 +
                     totalTripsBus3

totalTripsAllBuses = 5 + 2 + 1
                   = 8 trips
```

Brute Force

With prev. ellaboration will look like...

canCompleteTrips()

```
bool canCompleteTrips(vector<int>& time, int
                      givenTime, int totalTrips) {

        int totalPossibleTrips = 0;

        for(int i = 0;i < time.size();i++) {

            totalPossibleTrips += (totalTime/time[i]);

        }

        return  totalPossibleTrips >= totalTrips;

    }
```

If sum of trips that all buses (combined) can make is more than totalTrips, we return true.

ex-> totalTrips = 5, & if totalPossibleTrips = 7, which means combined trips made by buses is more than totalTrips we were asked i.e. 7 > 5, we return true, as in givenTime we are able to complete given totalTrips.

Brute Force

since you are clear with the implementation of 'canCompleteTrips()' let's start with Brute Force.

So idea is, we start with time t = 1 & see if canCompleteTrips(t) return true of false, as we want min. time in which totalTrips can be made so as soon as for some time t our canCompleteTrips(t) returns true, we return that time t.

i/p	o/p
time = [1,2,3], totalTrips = 5	3

t	canCompleteTrips	comment
1	false	can't complete trips, i++
2	false	can't complete trips, i++
3	true	can complete trips
4	true	if in 3 sec. we can complete 5 trips so obviously in t = 4,5,6 .. we would be able to complete

we want min. time so return 3

can you see, we are iterating over a number line from 1,2,3,4,5.. which is sorted

sorted space + linear search,
go for Binary Search

Minimum Time to Complete Trips

- We concluded to go for Binary Search, but we need to divide
  1) Search space in 2 parts ('F' & 'U')
  2) Decide whether 'l' or 'h' gives ans.

Dividing 'Search Space'

- let's say our time t is ans so till t-1 we can't completeTrips() & from t onwards we are able to completeTrips()
- starting from time t = 1...

  canCompleteTrips(t) = false, are 'U'
  canCompleteTrips(t) = true, are 'F'

- (as our ans won't lie in this space so we called it 'Unfavorable')

l	1	2	3	4	5	6	h
	U	U	F	F	F	F	

this is the smallest time in which we canCompleteTrips

1 • ans is given by first value of 2nd space, so when mid is at 'U' move 'right' (l=mid+1) else move 'left' (h=mid-1)

2 • when 'Binary Search' ends who points to first value of 2nd space... l (refer slide 9)

Minimum Time to Complete Trips

```
class Solution {
public:


    bool canCompleteTrips(vector<int>& time, int givenTime, int totalTrips) {

        int totalPossibleTrips = 0;

        for(int i = 0;i < time.size();i++) {

            totalPossibleTrips += (totalTime/time[i]);

        }

        return  totalPossibleTrips >= totalTrips;

    }

    int minimumTime(vector<int>& time, int totalTrips) {


        int l = 1;
        int h = time[0] * totalTrips;

        while(l <= h) {

            int mid = l + (h-l) / 2;

            if(canComplete(time, mid, totalTrips)) {
                h = mid-1;
            } else {
                l = mid+1;
            }


        }
        return l;
    }
};
```

- Try to figure out what should be the value for h (higher limit)
- We can also sort the time[](reverse) so canCompleteTrips() performs better...Why? tell me in comments (I'll reveal in next episode)

Split Array Largest Sum

Description

- Given an array nums which consists of non-negative integers and an integer m, you can split the array into m non-empty continuous subarrays.
- Write an algorithm to minimize the largest sum among these m subarrays.

i/p	o/p
nums = [7,2,5,10,8], m = 2	18

explanation

- Above nums[] can be split in 2 parts in many ways

split 1	split 2	max sum
[7]	[2,5,10,8]	25 (split 2)
[7,2]	[5,10,8]	23 (split 2)
[7,2,5]	[10,8]	18 (split 2)
[7,2,5,8]	[8]	22 (split 1)

- out of all combinations 3rd combination has smallest max sum (18) & that's our answer

Brute Force

- Approach #1
  - As we are considering all possible combinations & out of that we are concerned with one optimal one.
  - So what all ideas you got ...

All possible combinations

Recursion

Optimal combination

D.P. (dynamic programming)

- Approach #2

- Idea is, we would start with some 'low' limit & check if it is possible to split nums[] in 'm' split such that maxSum of any split in not more than 'low'

i/p	o/p
nums = [7,2,5,10,8], m = 2	18

- We need to divide nums[] in 2 splits
- 10 is largest element so our smallest max. element can't be less than 10
- so let's say low = 10
- Now the largest possible sum for a split can be sum of nums
- so let's keep high = sum(nums) = 32

- Our answer will lie b/w low & high limits only

Brute Force

But what do these low to high limits tell ?

- Remember, we wanted to minimize the sum of largest split
- Now we start iterating from i = low to i <= high
- for every i we check if we canSplit() our nums[] in m parts such that no part has sum > i
- The first i that gives 'true' for canSplit(), that's our answer (as we want the smallest sum with which we were able to split nums in m part)

- So let's 1st implement the canSplit() , it takes 3 parameters
  1) nums[]
  2) maxSum (i value for which we check if no split's sum > i)
  3) m (no. of splits we want)

```
bool canSplit(vector<int>& nums, int maxSum, int m) {

        int totalPart = 0;
        int currSum = 0;

        for(int i = 0;i < nums.size();i++) {
            if(currSum + nums[i] <= maxSum) {
                currSum += nums[i];
            } else{
                currSum = nums[i];
                totalPart++;
            }
        }

        return (totalPart + 1) <= m;
    }
```

Brute Force

i/p

nums = [7,2,5,10,8],
m = 2

o/p

18

low = 10
high = 32

- let's start with i = 10 to i <= 32

i = 10

can we split nums[] in m parts such that no part has sum > 10 ?
No

[7,2] , [5,10,8]

we kept 1st split sum(9) <= 10 but 2nd split sum(23) > 10
so if you pass nums[], i, m in canSplit() i.e.

canSplit(nums, i, m)   it returns false

if we keep traversing linearly from i = 10,11,12.. till 18 we won't be able to split nums, as our canSplit() returns 'false'

i = 18

can we split nums[] in m parts such that no part has sum > 18 ?
[7,2,5] , [10,8]

yepp, we are able to split in 2 parts & sum of both parts sum(7+2+5)=14 or sum(10,8)=18 is <= 18

so canSplit() return 'true' this time

so 18 is our ans (remember we wanted to minimize the largest sum a split can have)

with i = 18 we are able to split nums such that the max sum is <= 18, so obviously for i > 18 (19,20,100...) also we would be able to split the nums[] such that the sum of any split is not > i

so from 18 onwards our canSplit() will return true

Intuition

i/p

nums = [7,2,5,10,8],
m = 2

o/p

18

low = 10 high = 32

	low													high
	10	11	12	...	15	16	17	18	19	20	21	....	32
canSplit()	F	F	F		F	F	F	T	T	T	T		T

our answer

can you observe what we are doing, linear search on a sorted space so what's better than

Binary Search

Solving a Binary Search problem requires 2 steps

1) Divide the search space in 2 parts

2) After 'Binary Search' ends check whether low or high , who gives your answer.

1) Divide the search space in 2 parts

1 We already divided the space in 2 parts, part 1 where canSplit() return 'false' & part 2, where canSplit() returns 'true'

Intuition

low high

	10	11	12	...	15	16	17	18	19	20	21	....	32
canSplit()	F	F	F		F	F	F	T	T	T	T		T

our answer

When B.S started

2) After 'Binary Search' ends check whether low or high , who gives your answer.

- See in above no. line our answer is 1st element of 2nd space(true or favorable space)
- When B.S. started low point to 1st ele. of 1st space & high points to last ele of 2nd space

high low

	10	11	12	...	15	16	17	18	19	20	21	....	32
canSplit()	F	F	F		F	F	F	T	T	T	T		T

our answer

When B.S ends

2 • When B.S ends (while loop terminates) i.e. (low becomes > high), who points to 1st ele. of 2nd space...

low

- So finally low gives your answer

our answer

when your 'mid' is at 'F' as you want to 1st 'T' move low to right of mid i.e

3 low = mid + 1

when your 'mid' is at 'T' as you want to 1st 'T' move high to left of mid i.e

high = mid - 1

And who tells whether you are at 'F' or 'T' ?

canSplit()

```cpp
class Solution {
public:


    bool canSplit(vector<int>& nums, int maxSum, int m) {

        int totalPart = 0;
        int currSum = 0;

        for(int i = 0;i < nums.size();i++) {
            if(currSum + nums[i] <= maxSum) {
                currSum += nums[i];
            } else{
                currSum = nums[i];
                totalPart++;
            }
        }

        return (totalPart + 1) <= m;
    }


    int splitArray(vector<int>& nums, int m) {

        int low = 0,high = 0;
        int sum = 0;
        for(int i = 0;i < nums.size();i++) {
            high += nums[i];
            low = max(low, nums[i]);
        }

        while(low <= high) {

            int mid = low + (high - low) / 2;

            if(canSplit(nums, mid, m)) {
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }

        return low;


    }
};
```

Leave a Like

Comment if you love posts like this, will motivate me to make posts like these

Share, with friends

will be continuing this series, see it takes hell lot of efforts & these are Slides so there might be some 'Typos' or I would have missed something so try to help me make it correct & avoid negatively criticizing things.