**More Details**

**Threads in java**

So now with this we can create threads in two ways

1. Extending the thread class
2. Implementing the Runnable interface

In first case when while extending thread class we just need to make the object of that extended class and we will be having all the methods from the thread class and upon using t.start() will run the RUN method present in the extended thread class, if t is the object of the extended class

In second case when we are implementing the interface, then while making the object for the thread we need to pass the object of this class as a reference to the Thread class constructor

MyClass obj1 = new MyClass();

Thread t1 = new Thread(obj1);

So with this two or { Thread t1 = new Thread(new MyClass()); } , this instruction will make the thread object and we can use all the methods such as

1. **public void run():**is used to perform action for a thread.
2. **public void start():**starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):**Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():**waits for a thread to die.
5. **public void join(long miliseconds):**waits for a thread to die for the specified miliseconds.
6. **public int getPriority():**returns the priority of the thread.
7. **public int setPriority(int priority):**changes the priority of the thread.
8. **public String getName():**returns the name of the thread.
9. **public void setName(String name):**changes the name of the thread.
10. **public Thread currentThread():**returns the reference of currently executing thread.
11. **public int getId():**returns the id of the thread.
12. **public Thread.State getState():**returns the state of the thread.
13. **public boolean isAlive():**tests if the thread is alive.
14. **public void yield():**causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():**is used to suspend the thread(depricated).
16. **public void resume():**is used to resume the suspended thread(depricated).
17. **public void stop():**is used to stop the thread(depricated).
18. **public boolean isDaemon():**tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):**marks the thread as daemon or user thread.
20. **public void interrupt():**interrupts the thread.
21. **public boolean isInterrupted():**tests if the thread has been interrupted.
22. **public static boolean interrupted():**tests if the current thread has been interrupted.

**Sleeping a thread**

So, if we want to stop the program for the moment then we can use Thread class static method for this purpose i.e Thread.sleep(TIME\_IN\_MILLISECONDS)

As run is just a method sitting in the implemented or extended class, so we can call it directly but that will not create a thread as in case of creation of thread a new call stack is generated for the use of that thread but when we just call the run method then it will simply be added onto the current call stack and will be run as the simple program but not as the thread.

If we want a thread to complete its run method and then we need to call some other method or thread thus this can be done with the help of t.join(); this will wait until the thread will stop.

We can use methods getName(), setName(), getId() methods on thread objects for our representation purposes.

And as for a moment number of threads could be running at the same time so we need to find which is running at that moment of time then we can use a static method

Thread.currentThread(); returns the reference to the thread running currently.

And there is priority present with the each thread If not mentioned NORM = 5 is given to threads and there are other as well THREAD.MIN\_PRIORITY = 1, THREAD.MAX\_PRIORITY = 10, THREAD.NORM\_PRIORITY = 5

So we can use method t.setPriority(THREAD.MAX\_PRIORITY) etc.

Problem of leet-code

<https://leetcode.com/problems/set-matrix-zeroes/description/>

So this is the problem in which we need to make the row and column of corresponding block to be zero where the value is zero initially.

So there are many options to complete this question.

1. O(mn) space, in this one we would simply create another matrix of m\*n size and just copy the content to other matrix then make row and column zero where the block is zero.
2. O(m+n) space, in this one we would make two arrays one of size m and other of size n, in which we will keep on storing the row and column which are to be made zero based on the value of initial array.
3. O(1) space, in this on we need to find the row and column in which there are no zeros at all and if we are unable to find either of the row or column then just simply make the all values as zero otherwise we need to use that row and column as the reference and make as if a[i][j] == 0 then we need to make a[row][j] = 0 and a[i][col] = 0 and later on use them to make all the required one’s zero.

Game of Life – Question from Leetcode

<https://leetcode.com/problems/game-of-life/description/>

So this is the question in which we need to change the living or dead status of the elements present in the matrix form and their future status depends upon the current scenario which is if

1. Live cell will die if more than three live neighbor(Overpopulation) and less than two live neighbor(under-population) are present.
2. Dead cell will become live if there are three live neighbor present.

So we need to do this question in place, for which we need to change the current scenario but also we need to take care of the change which we are causing to elements which would be neighbors to the other elements.

Upper\_bound – this is a function present in STL which can be used to find the first no which is greater than the no present as the third parameter.

upper\_bound(LOWER\_LIMIT, UPPER\_LIMIT, VALUE)

this contains lower limit which corresponds to the pointer to the lower limit and same for the upper bound.

**Fractional knapsack**

In this we are given with the weights and profit associated with them and we need to find the profit we can maximally make with the maximum fixed weight we can take!

So now we need to arrange the input weight and profit pair in a order which is arranged by profit / weight ratio in decreasing order.

And then we need to choose the weights if there is still space in the bag in this we can maximize the profit and utilize the constant space which we are having.

* Find the maximum path sum between two leaves of a binary tree

For this we need to take care and we need to run in the backward fashion that is from leaves to root and hence we can see if the sum of current node left and right is maximum change the maximum value otherwise move up and return the maximum of left, right and the current node value and in this way we can do this question.

# Find maximum of minimum for every window size in a given array

# In this one, if we consider one window of size k then we need to go to all the subarrays of that window size and keep on finding the minimum value of all the elements in these windows which will give us n/k elements and then we need to take the maximum value out of it .

# SO for this if we take a window size then we can keep on moving forward and removing the last element for this purpose we need to use deque and keep on adding the element from the front and removing from the last and this will get it done for a window in O(n) and we have n such windows so, O(n^2)

# Largest Rectangular Area in a Histogram

# So



So we need to work on this problem and we need to find the maximum area which can be there and we can do this as by taking the current height as the minimum and find the just greater that the current element towards its left and right and then find the value of maximum area for finding the greater element towards left and right we need to use stack and do this task in O(n) complexity.

* Check if it is a complete binary tree.

In this complete binary tree it has 2 children for each and it is as left as possible and in this we need to check that if we are having the tree and whenever we get the leaf node and after that we should be getting only the leaf nodes and not the other non-leaf nodes;

* Making a stack which can support O(1) deletions and finding of the middle element of the stack.

So as basically we need to implement the stack so we need to use some other datastructure such as array or LL but in case of arrays middle element deletion cannot be O(1) so we need to go with LL but in this case too we need to take this into consideration that we need to change the middle element back and forth but this steps of moving back and forth can be provided by doubly linked list and not by simple LL. So we will be using doubly LL and using other basic traits to judge whether to move forward or backward.

* Finding the majority Element from the unsorted array

1. Sort the array and then find the majority element in O(n) complexity but the total complexity would be O(nlog(n)) so we need to go with something interesting.
2. First check the number which is coming maximum number of times then find whether that count is exceeding the n/2 bar, so we can go for this in a much different way we first of all need to think it as we are having many that maximum number values and many others and we start by one on one cancelling between the maximum one and the others and eventually if we can cancel all others but still left with the maximum we found our maximum.

for(int i =0;i<n;i++){

if(count == 0){

last = a[i];

count++;

}else{

if(last == a[i]){

count++;

}else{

count--;

}

}

}

So this will let us find the value in last if count is greater than zero if zero it means that either the maximum value is equal to n/2 or some sought of pairing.

* Block Swap algorithm for array rotation.

Array

ArrayRotation1

So in this we need to convert this array from first image to the second image and in this we can use a little trick which ran as follows.

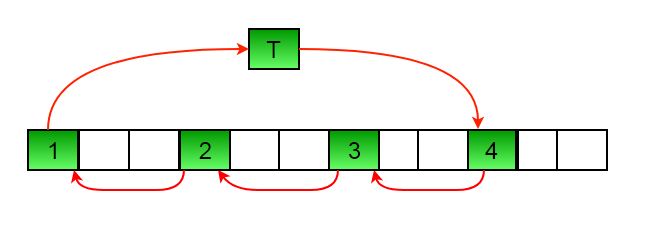
We first of all need to reverse the right part and then reverse the left part and then reverse the full part

1. Reverse(a, 0, k-1)
2. Reverse(a, k, n-1)
3. Reverse(a, 0, n-1)

These three reversals will get the job done!

<https://www.geeksforgeeks.org/block-swap-algorithm-for-array-rotation/>

There is another method present as well for this algorithm called juggling method in which first of all find the gcd of n and d and then use that value to form the blocks and in each block start swapping as shown.



So first of all first element of each block is to swapped then second and then at last the GCD element.

* Kth Smallest or Largest Element in the unsorted array

1. We can sort the array and then find the Kth element directly.
2. We can use the priority queue and then pop the first k-1 elements and get the kth element out of it.
3. We can use the quick select method in which we basically need to use quick sort based algo in which we will use quick sort partition method and check if the pivot element is kth or not.
4. We can improve the last algo with the help of using randomized quick select and thus linear expected time complexity and for this case we need to use random function which can be used in c++ as rand() which will give the value in any range so we need to apply modulus operation to find a random value.
5. So there is another method for this which involves worst case complexity as O(n) and this is the link to it: <https://www.geeksforgeeks.org/kth-smallestlargest-element-unsorted-array-set-3-worst-case-linear-time/>

* Maximum Sum Path in two arrays: In this we are given with two arrays and which are sorted and we need to start our path from one of the arrays and we can move from one array to other at the point of similarity and can end at any of the arrays and we need to maximize the sum which we encounter.

* Move all zeros to the end of the array in random array of numbers: In this we can apply the technique which is used in the partition function of the quick sort in this we need to move it to move the zero if it is coming to the next non-zero location and swap them so as to maintain the sequence of the original array elements.

while(i<n && j<n){

if(a[i] != 0){

i++;j++;

}else{

while(a[j]==0){

j++;

}

if(j<n){

swap(a[i], a[j]);

j++;i++;

}

}

}

* Arrange the given array to form the biggest number which can be formed

So for this we need to sort the array and this can be done by making the comparator and which will be basically combining two string in both the ways and see which one is bigger and return the true or false accordingly from the comparator.

* Rearranging the negatives and positives in an array in which first there will be all negative arrays and then there will be all positives but the catch is they should be In same sequence as they were given.

For this we can apply many things

* Quick Sort method partition function which can be applied here and we can’t just use it because if we want to get the same sequence in which it was earlier then we need to think of something else. As of now partition works as follows.

int j = 0, i = 0;

while(i<n){

if(a[i] < 0){

swap(a[i], a[j]);

j++;

}

i++;

}

Another method is to use the merge sort in which we have to use the same alogorithm as that of using while merge sort but in this there will be change while merging the arrays back in this we need not have to check the sorting procedure but we have to put the negatives of left half first and then negatives of second half and then positives of first half and then positives of second half this will led to the final solution.

* Finding a missing element out of the second smallest array which contains one less element and we need to use binary search in this problem and find it in O(log(n))
* Minimum length Subarray Sorting which makes the array sorted into full:

1. We can do the sorting stuff first and after that we can check whether the length of the array which was already in that state from the left and then from right and mark the corresponding start and ends.
2. Another way to find this is by moving from left and find the sorted path and then from right finding the sorted path and then mark the start and ends and then we need to find the minimum and maximum element in the part which is needed to be sorted and then we can find that whether we needed to increase the length the part towards left and part towards the right by looking such a way that the minimum should be the maximum of all values in the left and maximum should be the minimum of all the values in the right and then we are good to go.

vector<int> find(int a[], int n){

int s, e, i = 0;

vector<int> v(2, 0);

while(i<n-1 && a[i] < a[i+1]){

i++;

}

if(i < n){

s = i;

}else{

return v;

}

i = n-1;

while(i>0 && a[i] > a[i-1]){

i--;

}

if(i>=0){

e = i;

}else{

return v;

}

int min\_ele = INT\_MAX, max\_ele = INT\_MIN;

for(int i = s;i<=e;i++){

min\_ele = min(min\_ele, a[i]);

max\_ele = max(max\_ele, a[i]);

}

while(s-1>=0&&a[s-1]>min\_ele){

s--;

}

while(e+1<n&&a[e+1]<max\_ele){

e++;

}

v[0] = s;v[1] = e;

return v;

}

So this is the code for finding the answer which we were searching for!

* Selection sort containing minimum number of memory writes this is the ritual which people usually think about it as but as we can see the actual winner here is the cycle sort.
* Equilibrium point in the array: This is the index which is basically dividing the array into two parts and sum of the left part is equal to the sum of the right part. For this we can calculate using the sum of the full array and then we can just move towards the right by deleting the left elements from the sum and correspondingly increasing the left sum and decreasing the total sum hence the right sum.
* Consecutive Array Elements: in this we will be given an array and we need to tell that whether the given elements are contiguous or not! And they should be distict.

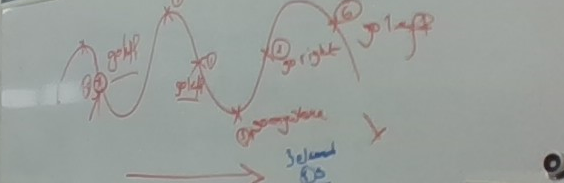
1. One way to do this is to use sorting and then check whether they are consecutive or not!
2. In this we need to check two conditions
   1. That max – min +1 == n, this by checking it in a contiguous fashion
   2. All the elements are distinct, using set data structure.

* Finding the number of occurrences of a number in the sorted array,
  + There could be number of ways to do this but the best one is to use the leftmost occurrence of the number and then the rightmost occurrence and hence find the right-left+1.
* Finding the minimum distance between the given x and y variables.
  + We can first of all use the all the x’s and y’s and find the value where it can lead to minimum.
  + We just keep a variable for last xpos and one for ypos and then we can use all of them as soon as we get both of them and if we haven’t got two of them in any case then we will simply return -1
* Finding the point where first positive arrives in the monotonic function
  + In this we need to apply binary search but we are not knowing before-hand that what is the upper limit so first of all we need to find the upper limit using some way which can be done using the 2^n use and move n until it is greater than the given value then apply binary search in n/2-n
* Given an array of size n and number k and find all elements that appear more than n/k times

1. Finding it one by one and check if the count is greater than n/k or not
2. Sort the array and then count whether the numbers appearing consecutive which would be same are greater than n/k or not.
3. Using hashmap we can calculate the elements which are having count greater than n/k in O(n) time complexity.
4. Last and the best way to understand things take O(nk) time complexity but also uses O(k-1) space for the solving.
   1. In this first of all we need to go and make an array of size k-1 which will store a pair containing element and the corresponding count and the elements can just be at most k-1 as in our answer we need to output if it’s occurrence is **greater than** n/k if something to exist so
      1. (>n/k)\*(x) = n so x is always <(k-1)
   2. Now if in any case the number of elements present in the array exceeds k-1 then we need to reduce the count of each element present in k-1 entries and then we can ignore the occurrence of the current element.
   3. Then at the end we will be having only k-1 elements or less in the array and then we can go and check whether there occurrence is more than n/k or not

It seems to be the dumbest way to solve the problem but it seems to be a different way to solve the problem.

But generally the variation of the problem involves the finding of the same in O(1) space and they usually ask for n/4 or n/3 so creating an array of 3 elements won’t be much space and it can be taken as O(1) and thus there hashing would fail.

* Peak Element in the array
  + In this we can go with the linear approach and that would take linear time complexity but we can do little better on this thing
  + In the other approach we need to use binary search it’s kind of a weird that with the unsorted data we can still use binary search as we can treat the same thing as the line graph
  + 
  + And then we can choose where to go by checking the position at which we are present and there would be number of peak elements present in the array and we can find any element which we would like to give.
* Stock buy and sell: So in this we can select multiple buy and sell options and then we can obtain the maximum profit and here we can take multiple buy sell pairs and hence can obtain the respective profit so we need to find the local minima and then local maxima and thus add it to the profit and in this way we can get the maximum profit which we can and this is basically applying the greedy algorithm here, otherwise if we were just allowed to pick one pair then we could have screwed it by just picking one maximum and one minimum to its left and find the maximum profit.
* Maximum Repeating number in array in O(1) space and O(n) time: so for this we need to take care that we just have O(1) space so we can do it by adding n to the location whatever ith number is and then at the last we can check by divinding with n that which is the number I with maximum value that will be the answer.
* Merge Overlapping Intervals: In this actually we are given with intervals and we need to make them output in such a way that each interval is mutually exclusive:
  + So for this we can use sorting in which we first of all need to sort the array based on the increasing order of the start time and then we just need to check that the end time of the ith entry is less than the start time of i+1 if not then split them otherwise combine them and hence it will reduce the amount of count of pairs.
* Tug Of War:
  + In this we will be having n elements and with this we need to take n/2 elements if n even and (n-1)/2 if odd and we have to minimize the sum’s difference and then find these floor(n/2) elements. We can do this with the help of using the backtracking in which we need to stop at the point when the temp vector contains the elements equal to required n/2 and it should have the minimum difference between each other and this would be the way.
* Pancake Sorting:
  + In this type of sorting we are just allowed to reverse the array from 0 to i and then we need to sort the array using this operation, so we can go with this as kind of a selection sort algorithm in this we need to sort according to like taking the maximum element at the bottom of the array and then keep on reducing the size of the array and keep on finding the maximum element and then putting at the end of the current array and then job is done,
    - How to rotate the array to get the maximum element to lowest position of the current size array, for this we need to first of find the maximum element index and then we need to rotate 0 to that index and which will led the maximum element to be at the top of the array and then we just need to rotate the array from top to the point which is the current size of the array and this will make the maximum array to the bottom of the array and then we can reduce the size of the array and we can do the same task till the size of array is not equal to 1