Algorithms

Compiled by - Sagar Udasi

(Version 0.0.1)

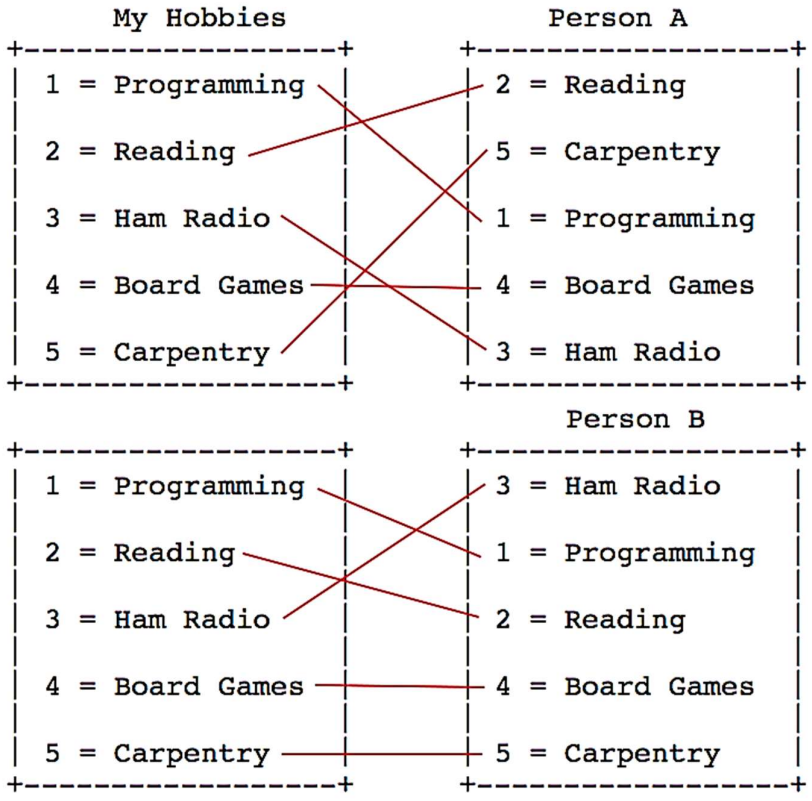
Chapter 1 – Sorting Based Problems

1.1 Inversions

Let's start by defining the problem of inversions. In an array, two elements form an inversion if a[i] > a[j] and i < j. In simple words, a bigger element appears before smaller one. In the following list: [1, 3, 5, 2, 4, 6], there are 3 inversions: (3, 2), (5, 2), and (5, 4).

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted, then inversion count is 0. If array is sorted in descending order, then inversion count is the maximum.

Inversion count is usually used by social media networks to find the people sharing common preferences. For example, if I wanted to find who in a group I most share the same hobbies with, I could have ranked everyone's hobbies, map them to values based on my hobbies to establish an order, and run them through an inversion counting algorithm to see who's hobbies most match mine. We would probably have to do a little extra clean-up work to get matching sets of data but after that it could look like this:



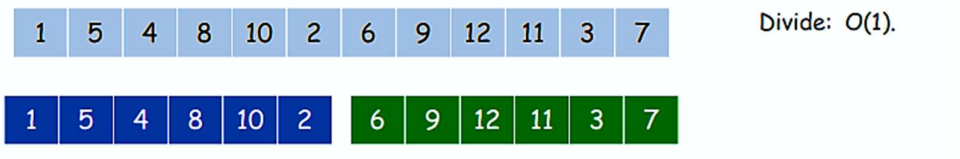
I have 4 inversions with person A and 2 inversions with person B. So, it looks like I should hang out with person B.

Similarly, music sites try to match your song preferences with others. The more you listen a song, that song gets a higher rank in your list. Music site consults database to find people with similar tastes. And on the basis of what they listen, you get music recommendations.

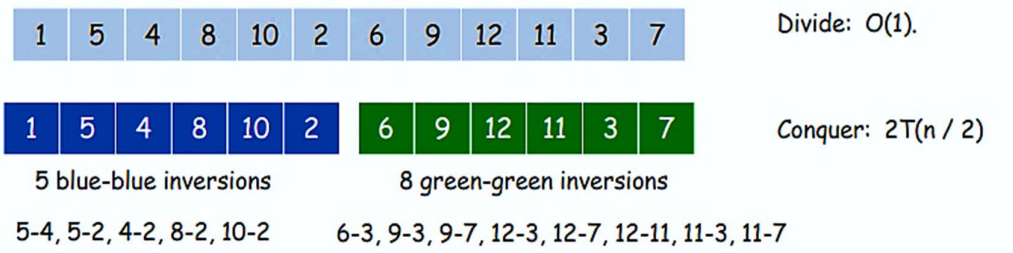
Now the question is – *how do we calculate the inversion count?*

The **naïve algorithm** is – traverse through the array from start to end; for every element find the count of elements smaller than the current in front of it; and finally sum up the count of inversion for every index. This is an $O(n^2)$ time complex approach.

The **efficient algorithm** is – to use *modified merge sort.* Let's see what modification we need to make. Like merge sort, we first divide the array into almost two halves. This is the *divide step.*

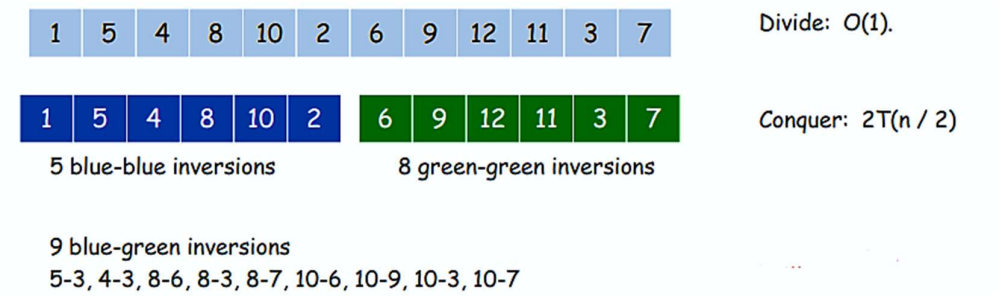


We will pass these halved arrays to recursive function. Now assume that the recursive function gives you back the answer of inversion count for these halved arrays. This is the *conquer step.*



In the final step, we need to merge these halved arrays. *How do we combine these arrays?*

Assume each half is sorted. Now count the number of inversions of a_i and a_j, where a_i and a_j belong to different halves. The merge step is similar to traditional merge of merge sort. Only additional thing you have to do is – keep a count in one half for every element, that how many elements bigger than that element are present in another half. Since the halves are sorted, this could be done in $O(n)$ time.



So, total number of inversions = 5 + 8 + 9 = 22.

This is an $O(n * \log n)$ time complex approach. However, it takes $O(n)$ extra space.

```
// Merge Function returns the number of inversions
// caused when left and right parts of array are merged
long merge (int arr[], int left, int mid, int right) {
    int i = left, j = mid, k = 0;
    int temp[right-left+1];
    long count = 0;
    while (i < mid && j <= right) {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else {
            temp[k++] = arr[j++];
            count += mid-i;          // Counting statement
        }
    }
    while (i < mid)
        temp[k++] = arr[i++];
    while (j <= right)
        temp[k++] = arr[j++];
    for (i = left, k = 0; i <= right; i++, k++)
        arr[i] = temp[k];
    return count;
}

// Modified mergeSort() which returns inversion count
long mergeSort (int arr[], int left, int right) {
    long inversion_count = 0;
    if (right > left) {
        int mid = left + (right-left)/2;
        inversion_count  = mergeSort(arr, left, mid);
        inversion_count += mergeSort(arr, mid+1, right);
        inversion_count += merge(arr, left, mid+1, right);
    }
    return inversion_count;
}
```

Problem

Q. 1 Given an array A, for every element, find the sum of all the previous elements which are smaller than the current element. Print the total sum of all such individual sums.

For eg. A = { 1, 5, 3, 6, 4 }.

```
For i = 0: sum = 0
For i = 1: sum = 1 (1 is smaller than 5)
For i = 2: sum = 1 (only 1 is smaller than 3)
For i = 3: sum = 9 (1, 5, 3 all are smaller than 6)
For i = 4: sum = 4 (1, 3 are smaller than 4)

Total sum = 0 + 1 + 1 + 9 + 4 = 15
Output = 15.
```

Solution: Here we are not interested in inversions, instead we are interested in the pairs which are in order. So in the merge() of inversion count code, instead of writing 'counting statement' in else block, we will write a 'summing statement' in if block.

```
long merge (int A[], int left, int mid, int right) {
    int i = left, j = mid, k = 0;
    int temp[right-left+1]; long sum = 0;

    while ( i < mid && j <= right) {
        if (A[i] < A[j]) {
            sum += (right-j+1)*A[i];  //Summing statement
            temp[k++] = A[i++];
        }
        else temp[k++] = A[j++];
    }
    while (i < mid)     temp[k++] = A[i++];
    while (j <= right)  temp[k++] = A[j++];
    for (i = left, k = 0; i <= right; i++, k++)
        A[i] = temp[k];
    return count;
}

long merge_sort (int A[], int left, int right) {
    long sum = 0;
    if (right > left) {
        int mid = (left+right)/2;
        long sum  = merge_sort (A, left, mid);
        long sum += merge_sort (A, mid+1, right);
        long sum += merge(A, left, mid+1, right);
    }
    return sum;
}
```

1.2 Partition method of Quicksort

Partition method of Quicksort can solve many re-arrangement types of problems which may not be obvious at first look. Let's see few of them.

Suppose you have an array A = { -5, -2, 5, 2, 4, 7, 1, 8, 0, -8 } and you want to rearrange the elements such that array has alternate positive and negative terms.

The **naïve way** will be to sort the elements in $O(n * \log(n))$ quicksort way and then swap alternate negative numbers with positive numbers. Here, by sorting the elements, we are overdoing. We just need to arrange positive and negative elements alternatively. We can achieve this in $O(n)$ complexity.

The **efficient way** is to use Quicksort's partition algorithm by considering pivot point as 0. After partition process, all the negative numbers will come at the beginning of the array and all the positive elements will be pushed to the end of the array. The negative elements and positive elements will not be in sorted order, but are just separated. We then swap the positive and negative elements to get the desired result.

```
void rearrange (int arr[], int n) {
    // Partition algorithm with pivot as 0
    int i = -1;
    for (int j = 0; j < n; j++)
        if (arr[j] < 0)
            swap (arr[++i], arr[j]);
    // Now swap +ve and -ve elements to make them alternating
    int neg = 0, pos = i + 1;
    while (pos < n && neg < n) {
        swap (arr[neg], arr[pos]);
        pos++; neg += 2;
    }
}
```

For A = { -5, -2, 5, 2, 4, 7, 1, 8, 0, -8 }, we have the following output:
⇨ 2 -2 4 -5 7 -8 1 0 8 5

One famous follow up question is – look at the output carefully. You will see the order of negative (as well as positive) numbers has changed in the output. The negative numbers in original array were in order: -5, -2, -8. But in output, order is -2, -5, -8. The order has changed because quicksort is not a stable sorting algorithm.

So, the follow up question is – *Can you maintain the order of the appearance of the elements and still get the alternate positive-negative pattern?*

The above problem becomes very easy if extra $O(n)$ space is allowed. You can just keep negative elements in order in some separate array and then replace the negative elements in the answer using this array. However, the problem becomes interesting (and it is not possible to solve it using Quicksort's partition method) if $O(1)$ space constraint is kept.

This is the simple idea which solves this problem. Find the element which is out of place. In our example, we want alternate positive and negative elements. So, you can imagine, positive elements occupy even indices of array and negative elements occupy odd indices of array. So, we call an element out of place, if it is negative and at even index or if it is positive and at odd index. Once we find out of place element, we then find the next out of place element which has opposite sign, i.e. if the current out of place element is +ve, we will find next -ve out of place element. Once we got such next element, we rotate this subarray to right by 1.

Let's dry run this on our example, A = { -5, -2, 5, 2, 4, 7, 1, 8, 0, -8 }.

Iteration No.	Out of place element	Next out of place element with opposite sign	Array after performing one right rotation
1.	-5	5	{ 5, -5, -2, 2, 4, 7, 1, 8, 0, -8 }
2.	-2	2	{ 5, -5, 2, -2, 4, 7, 1, 8, 0, -8 }
3.	7	-8	{ 5, -5, 2, -2, 4, -8, 7, 1, 8, 0 }

Since we are using right rotations to bring +ve and -ve elements in alternate order, the order of appearance of +ve as well as -ve elements remains unchanged. Although the follow up discussion was not related to quicksort's partition method, but you should know how to achieve rearrangements by maintaining the order of elements.

```
void rightrotate(int arr[], int n, int out_of_place_index, int current){
    int tmp = arr[current];
    for (int i = current; i > out_of_place_index; i--)
        arr[i] = arr[i-1];
    arr[out_of_place_index] = tmp;
}

void rearrange_in_order (int arr[], int n) {
    int out_of_place_index = -1;
    for (int i = 0; i < n; i++) {
        // if no entry has been found which is out of place,
        // check if the current entry is out of place or not.
        if (out_of_place_index == -1) {
```

```
            // condition for element to be out of place
            if (((arr[i] >= 0) && (i&1)) || ((arr[i] < 0) && !(i&1)))
                out_of_place_index = i;
        }
        if (out_of_place_index >= 0) {
            if (((arr[i] >= 0) && (arr[out_of_place_index] < 0)) ||
               ((arr[i] < 0) && (arr[out_of_place_index] >= 0))) {
                rightrotate(arr, n, out_of_place_index, i);
                // if the rotated elements are adjacent
                if (i - out_of_place_index < 2)
                    out_of_place_index = -1;
                // else the new out of place entry is two steps ahead
                else
                    out_of_place_index += 2;
            }
        }
    }
}
```

For A = { -5, -2, 5, 2, 4, 7, 1, 8, 0, -8 }, above logic will produce the output as:

⇨ 5 -5 2 -2 4 -8 7 1 8 0

Task for you: *Above implementations are for rearranging elements in alternate positive and negative manner. Edit above code (or write on your own) so that elements are rearranged in alternate negative and positive manner, i.e. this time, your first element should be negative.*

Let's now continue our discussion on quicksort's partition method. We saw rearranging alternate positive and negative elements.

How about rearranging alternate odd and even elements? If order doesn't matter, we will use quicksort's partition method.

```
void rearrange (int arr[], int n) {
    int i = -1;
    for (int j = 0; j < n; j++)
        if (arr[j]%2 == 0)          // Modification
            swap (arr[++i], arr[j]);
    // Now there are all even numbers till index i
    int even = 0, odd = i + 1;
    while (even < n && odd < n) {
        swap (arr[even], arr[odd]);
        odd++;
        even += 2;
    }
}
```

By now, you must have got the idea of how to rearrange alternately – with and without maintenance of the order of elements. Now, let's see one slightly different and important problem – **pushing all zeros to the end.**

The problem is – suppose there is an array, A = {1, 0, 2, 0, 0, 3, 4, 0, 5}. You need to push all zeros to the end, i.e. your output should be, A = {1, 2, 3, 4, 5, 0, 0, 0, 0}.

The idea is – maintain the 'count' of non-zero elements. So, at any moment, 'count' represents the index before which there are non-zero elements and element at index 'count' is zero. Traverse the array. When you see a non-zero element, you swap it with 'count' and increment 'count'.

```
void zerosToEnd (int arr[], int n) {
    int count = 0;
    for (int i = 0; i < n; i++)
        if (arr[i] != 0)
            swap (arr[count++], arr[i]);
}
```

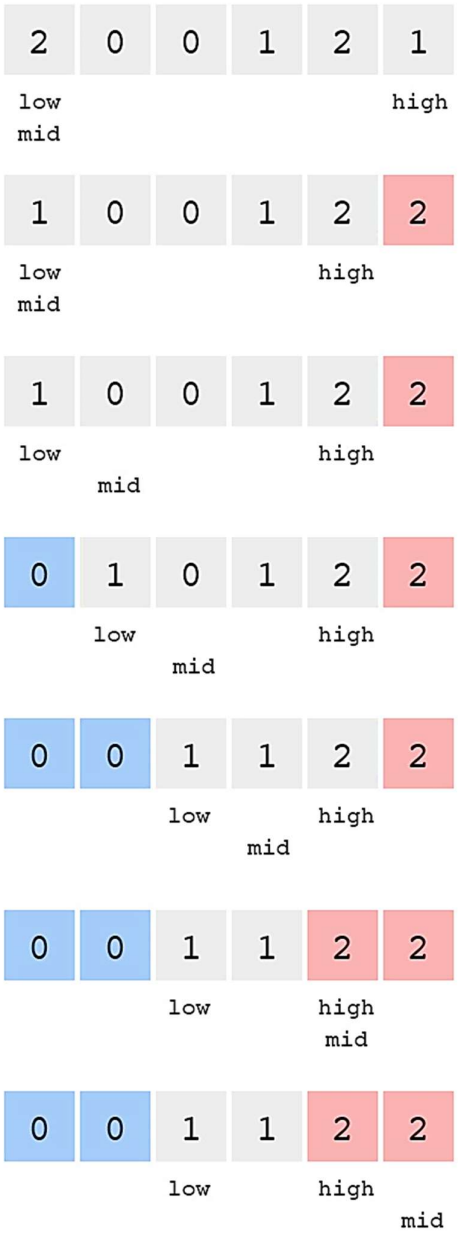
1.3 Dutch National Flag Problem

Given an array with only elements as '0', '1' and '2'. Sort this array.

The problem is very famous because the problem aims to sort this array in $O(n)$ time complexity. This is possible because there are only three types of elements and we can separate them using a special partitioning technique known as *'3-way partitioning'*. This is a partition scheme which segregates items into three different sets.

We do so by maintaining three variables – low, mid and high.

- If the element at arr[mid] is a 0, then swap arr[mid] and arr[low] and increase the low and mid pointers by 1.
- If the element at arr[mid] is a 1, increase the mid pointer by 1.
- If the element at arr[mid] is a 2, then swap arr[mid] and arr[high] and decrease the high pointer by 1.




```
void dnf(int arr[], int n) {
    int low = 0, mid = 0, high = n-1;
    while (mid <= high) {
        if (arr[mid] == 0)
            swap(arr[low++], arr[mid++]);
        else if (arr[mid] == 1)
            mid++;
        else
            swap(arr[mid], arr[high--]);
    }
}
```

1.4 Form the biggest number

Given an array of numbers, arrange them in a way that yields the largest value. For example, if the given numbers are {54, 546, 548, 60}, the arrangement 6054854654 gives the largest value. And if the given numbers are {1, 34, 3, 98, 9, 76, 45, 4}, then the arrangement 998764543431 gives the largest value.

Here if you notice, we need to sort the numbers in the reverse dictionary order. Here numbers are not sorted as per their value. They are sorted as if they are treated like strings. C++'s inbuilt string methods allow us an easy string to number and number to string conversion.

So the idea is – to write our own compare function for inbuilt sort() method, which will compare the numbers while sorting as if they are appended strings. For example, let X and Y be 542 and 60. To compare X and Y, we compare 54260 and 60542. Since 60542 is greater than 54260, we put Y first.

```
bool my_compare (int a, int b) {
    string temp1 = to_string(a).append(to_string(b));
    string temp2 = to_string(b).append(to_string(a));
    return (temp1 > temp2);
}

int main() {
    int arr[] = {1, 34, 98, 9, 548, 76, 4};
    int number = sizeof(arr)/sizeof(arr[0]);
    sort (arr, arr+number, my_compare);
    printArray (arr, number);
    return 0;
}
```

1.5 Finding Median in O(n) time

Before discussing about medians, let's discuss a more general problem first – finding k^{th} smallest element in an array. For e.g. if input array is A = [3, 5, 1, 2, 6, 9, 7], 4^{th} smallest element in array A is 5, because if you sort the array A, it becomes A = [1, 2, 3, 5, 6, 7, 9] and now you can easily see that 4th element is 5.

The **naïve way** is to sort the array and then find the element at index 'k'. Since quicksort doesn't take extra space, it is usually preferred over merge sort. But the worst-case complexity of quicksort is $O(n^2)$. But we can usually avoid it by selecting the pivot element randomly.

The **efficient way** is to optimize quicksort little more. One thing to notice is – if after partitioning, pivot is at position 'j', we can say that the pivot is j^{th} smallest element of the array. Our task is to make this 'j' equal to 'k'. If j < k, the k^{th} smallest element lies on the right side of the j, and if j > k, then on the left side. So, at any time, we need to partition only one side of the array. Theoretically, this algorithm still has the complexity of $O(n * \log(n))$ but practically, you do not need to sort the entire array before you find k^{th} smallest element. And if you choose the pivot element wisely, the expected time complexity reduces to $O(n)$. Let's take an example to see how this method works.

Consider A = [4, 2, 1, 7, 5, 3, 8, 10, 9, 6] and k = 5.

4	2	1	7	5	3	8	10	9	6

For demonstration purposes, we will consider pivot element as the first element of the array. So here pivot = 0.

After 1^{st} partition, the correct position of A[pivot] is at 3^{rd} index.

3	2	1	4	5	7	8	10	9	6

Since 3 is less than k – 1 (because zero-based indexing), k^{th} smallest element is on the right side.

Now the pivot is the 1^{st} element of the right subarray. So pivot = 4.

After 2^{nd} partition, the correct position of A[pivot] is at 4^{th} index.

3	2	1	4	5	7	8	10	9	6

Since 4 is equal to k – 1, k^{th} smallest element of A is 5.

This algorithm of using quicksort's partition method for selecting the k^{th} smallest/biggest element from an array is called '*quick-select algorithm*'.

There are a lot many efficient methods devised to choose pivot wisely. As of now, we won't be adopting the most efficient technique, but a sufficient good technique is to choose pivot element using random function.

```
int partition (int arr[], int l, int r) {
    int x = arr[r], i = 1;
    for (int j = 1; j <= r-1; j++) {
        if (arr[j] <= x)
            swap(arr[i++], arr[j]);
    }
    swap(arr[i], arr[r]);
    return i;
}
```

```
int randomPartition (int arr[], int l, int r) {
    int n = r - l + 1;
    int pivot = rand()%n;
    swap(arr[pivot+1], arr[r]);
    return partition(arr, l, r);
}

int KthSmallest (int arr[], int l, int r, int k) {
     // Run code only if k is smaller than
     // the number of elements in array
    if (k>0 && k <= r-l+1) {
        int pos = randomPartition(arr, l, r);
        if (pos == k)
            return arr[pos];
        if (pos > k)
            return KthSmallest(arr, l, pos-1, k);
        else
            return KthSmallest(arr, pos+1, r, k-pos+l-1);
    }
    // If k is more than number of elements in array
    return INT_MAX;
}
```

Now coming back to our median topic. We know, median of an array is the middle element of the sorted form of that array. So, if we know the size of the array, which we do most of the times, we can find median in $O(n)$ time by setting -

$k = \frac{N}{2}, if\ N\ is\ odd$ OR $k_1 = \frac{N-1}{2}\ and\ k_2 = \frac{N}{2}, if\ N\ is\ even.$

Then for odd case, median is the k^{th} smallest element and for even case, median is the average of $k_1^{th}\ and\ k_2^{th}$ smallest elements.

One more thing which you should know is – how to calculate k^{th} smallest element using heaps (or priority queues)? This is the method which you should tell to the interviewer first, if asked – how will you find k^{th} smallest element?

You can find k^{th} smallest element using – Min-heap, as well as Max-heap.

Min-heap logic:

When working with Min-heap, logic is simple. Create a heap of all elements of the array. Pop the root of the heap, k-1 times. Then the root of the heap will be the k^{th} smallest element. Time complexity of this method is $O(n + k * \log(n))$.

```cpp
int KthSmallest(vector<int> &vec, int k) {
    // use std::greater as the comparison function for min-heap
    priority_queue<int,vector<int>,greater<>> pq(vec.begin(), vec.end());
    // pop from min-heap exactly (k-1) times
    while (--k) {
        pq.pop();
    }
    // return the root of min-heap
    return pq.top();
}
```

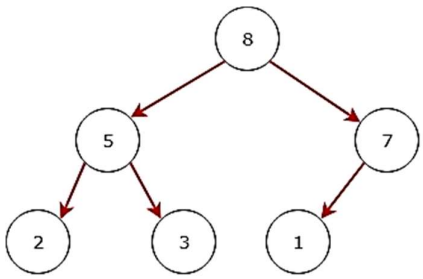
Max-heap logic:

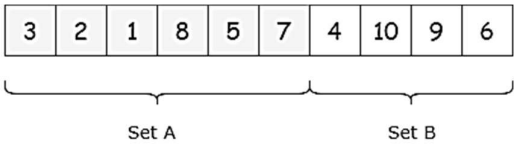
1. Create a max heap of size k from first k elements of array.
2. Do the following for each element left in the array:
    a. If current element is less than max on heap, add current element to heap and heapify.
    b. If not, then go to next element.
3. At the end, max heap will contain k smallest elements of array and root will be kth smallest element.

Let's take an example to see how above logic works. Let's say we want to find 6^{th} smallest element in the following array.

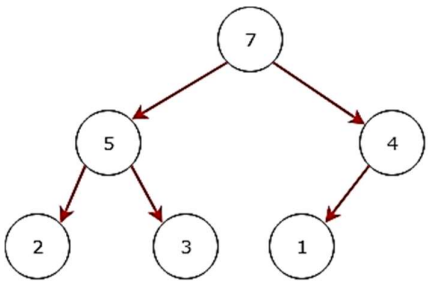
3	2	1	8	5	7	4	10	9	6

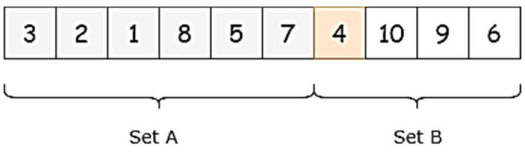
1. Create a Max-heap with first 6 elements of array.



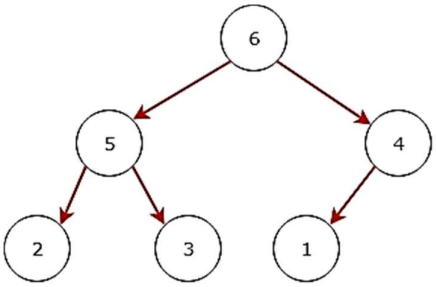


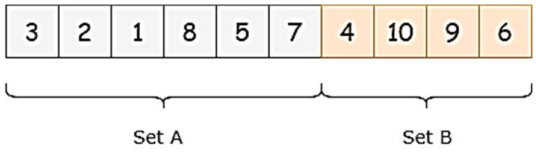
2. Take the next element from set B and check if it is less than the root of max heap. In this case, yes, it is. Remove the root and insert the new element into max heap.





3. It continues to 10, nothing happens as the new element is greater than the root of max heap. Same for 9. At 6, again the root of max heap is greater than 6. Remove the root and add 6 to max heap.





Again, new element from set B is less than root of max heap. Root is removed and new element is added. Array scan is finished, so just return the root of the max heap, 6 which is the sixth smallest element in given array.

Time complexity of this method is $O(k + (n - k) * \log(k))$.

```
int KthSmallest(vector<int> &v, int k) {
    // insert first k elements of the array into the heap
    priority_queue<int, vector<int>> pq(v.begin(), v.begin() + k);
    // do for remaining array elements
    for (int i = k; i < v.size(); i++) {
        // if current element is less than the root of the heap
        if (v[i] < pq.top()) {
            // replace root with the current element
            pq.pop();
            pq.push(v[i]);
        }
    }
    // return the root of max-heap
    return pq.top();
}
```

Chapter 2 – Searching Based Problems

2.1 Binary search

Search a sorted array by repeatedly dividing the search interval in half. Time complexity of BS is $O(\log_2 n)$. Binary search is not limited to arrays. It is an idea which can work on numbers, functions, etc. and not just on sequences. We will soon work on problems where this fact will get clearer.

Recursive version:

```
// Returns index of key if present or else -1.
int binarySearch (int arr[], int l, int r, int key) {
    if (r >= l) {
        int mid = l + (r-l)/2;
        if (arr[mid] == key) return mid;
        if (arr[mid] > key)
            return binarySearch(arr, l, mid-1, key);
        else
            return binarySearch(arr, mid+1, r, key);
    }
    return -1;
}
```

Iterative version:

```
int binarySearch (int arr[], int l, int r, int key) {
    while (l <= r) {
        int mid = l + (r-l)/2;
        if (arr[mid] == key)   return mid;
        if (arr[mid] < key)    l = mid + 1;
        else                   r = mid - 1;
    }
    return -1;
}
```

In C++'s algorithm library, we have a binary search function. Its declaration is:

```
bool binary_search(start pointer, end pointer, element to be found);
```

Apart from `binary_search()`, there is something called as – `lower_bound()` and `upper_bound()` as well. `lower_bound()` returns an iterator pointing to the first element in the range [first, last) which has a value **not** less than a particular value. `upper_bound()` returns an iterator pointing to the first element in the range [first, last) which has a value greater than 'value'.

```
iterator lower_bound(start pointer, end pointer, value);
```

```
iterator upper_bound(start pointer, end pointer, value);
```

Lower bound implementation:

```
int bs_lower_bound (int arr[], int n, int x) {
    int low = 0, high = n;   // Not n-1
    while (low < high) {
        int mid = low + (high-low)/2;
        if (x <= arr[mid])     high = mid;
        else                   low = mid + 1;
    }
    return low;
}
```

Upper bound implementation:

```
int bs_upper_bound (int arr[], int n, int x) {
    int low = 0, high = n;   // Not n-1
    while (low < high) {
        int mid = low + (high-low)/2;
        if (x >= arr[mid])     low = mid+1;
        else                   high = mid;
    }
    return low;
}
```

Using above functions:

```
int main() {
    int arr[] = {1, 7, 8, 10, 13, 20, 25, 30, 33};
    int size = sizeof(arr)/sizeof(arr[0]);
    // Case when key doesn't exist in array
    int* lb1 = lower_bound(arr, arr+size, 12);
    int lb2 = bs_lower_bound(arr, size, 12);
    cout << *lb1 << " " << arr[lb2] << endl;

    int* ub1 = upper_bound(arr, arr+size, 15);
    int ub2 = bs_upper_bound(arr, size, 15);
    cout << *ub1 << " " << arr[ub2] << endl;

    // Case when key exists in array
    int* lb3 = lower_bound(arr, arr+size, 13);
    int lb4 = bs_lower_bound(arr, size, 13);
    cout << *lb3 << " " << arr[lb4] << endl;

    int* ub3 = upper_bound(arr, arr+size, 20);
    int ub4 = bs_upper_bound(arr, size, 20);
    cout << *ub3 << " " << arr[ub4] << endl;
    return 0;
}
```

Output:

- ⇨ 13 13
- ⇨ 20 20
- ⇨ 13 13
- ⇨ 25 25

Problems

Q. 1 Given an array, find a **peak element** in it. A peak element is an element that is greater than its neighbours.

Solution: The idea is based on the technique of Binary Search to check if the middle element is the peak element or not. If the middle element is not the peak element, then check if the element on the right side is greater than the middle element. If yes, then there is always a peak element on the right side. If the element on the left side is greater than the middle element, then there is always a peak element on the left side. Form a recursion and the peak element can be found in $O(\log n)$ time.

```
int findPeak(int A[], int low, int high, int n) {
    int mid = (low + high) / 2;
    // check if mid element is greater than its neighbours
    if ((mid==0 || A[mid-1] <= A[mid]) && (mid==n-1 || A[mid+1] <= A[mid]))
        return mid;
    // If the left neighbor of mid is greater than the mid element
    if (mid-1 >= 0 && A[mid-1] > A[mid])
        return findPeak(A, low, mid - 1, n);
    // If the right neighbor of mid is greater than the mid element
    return findPeak(A, mid + 1, high, n);
}
```

Q. 2 Given a 2D array, find a **peak element** in it. Neighbours of a cell in 2D array are 4 cells situated at left, right, top and bottom. For corner elements, missing neighbours are considered of negative infinite value.

Solution: Consider mid column and find max value in it. Let index of the mid column be 'mid', value of the maximum element be 'max' and maximum element be at 'mat[max_index][mid]'. If max >= A[index][mid-1] and max >= A[index][pick+1], max is a peak, return max. If max < mat[max_index][mid-1], recur for left half of matrix. If max < mat[max_index][mid+1], recur for right half of matrix.

Time complexity of this solution is: $O(rows * \log(columns))$
Space complexity is: $O(columns)$ for recursion call stack.

```
const int MAX = 100;
int findMax(int arr[][MAX], int rows, int mid, int& max) {
    int max_index = 0;
    for (int i = 0; i < rows; i++) {
        if (max < arr[i][mid]) {
            max = arr[i][mid]; max_index = i;
        }
    }
    return max_index;
}

int findPeak(int arr[][MAX], int rows, int columns, int mid) {
    int max = 0, max_index = findMax(arr, rows, mid, max);
    // If we are on the first or last column, max is a peak.
    if (mid == 0 || mid == columns - 1)
        return max;
    // If max is a peak and we are on mid column
    if (max >= arr[max_index][mid - 1] && max >= arr[max_index][mid + 1])
        return max;
    // If max is less than its left
    if (max < arr[max_index][mid - 1])
        return findPeak(arr, rows, columns, mid - ceil((double)mid / 2));
    return findPeak(arr, rows, columns, mid + ceil((double)mid / 2));
}
```

Q. 3 There are 2 sorted arrays A and B of size n each. Write an algorithm to find the median of the array obtained after merging the above 2 arrays (i.e. array of length 2n).

Solution:

The naïve approach is to merge both arrays in $O(n)$ time and then get the middle element as median. However, we can do better in $O(\log n)$ time by comparing the medians of both the arrays. Here is the idea:

1) Calculate the medians m1 and m2 of the input arrays arr1[] and arr2[] respectively.
2) If m1 and m2 both are equal then we are done. Return m1 (or m2)
3) If m1 is greater than m2, then median is present in one of the below two subarrays.
   a) From first element of arr1 to m1 $\left(arr1\left[0 \ldots \left\lfloor\frac{n}{2}\right\rfloor\right]\right)$
   b) From m2 to last element of arr2 $\left(arr2\left[\left\lfloor\frac{n}{2}\right\rfloor \ldots n-1\right]\right)$
4) If m2 is greater than m1, then median is present in one of the below two subarrays.
   a) From m1 to last element of arr1 $\left(arr1\left[\left\lfloor\frac{n}{2}\right\rfloor \ldots n-1\right]\right)$
   b) From first element of arr2 to m2 $\left(arr2\left[0 \ldots \left\lfloor\frac{n}{2}\right\rfloor\right]\right)$
5) Repeat the above process until size of both the subarrays becomes 2.
6) If size of the two arrays is 2 then use below formula to get the median.

$$Median = \frac{max(arr1[0], arr2[0]) + min(arr1[1], arr2[1])}{2}$$

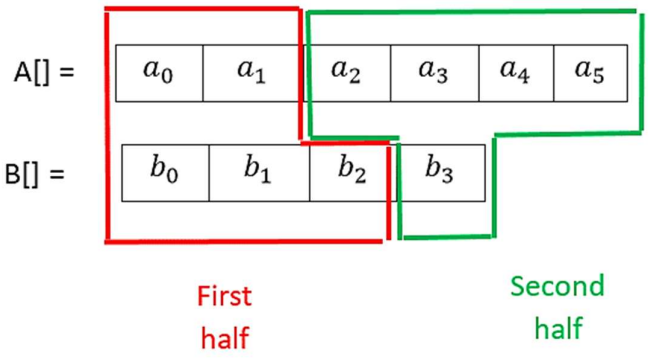
```
int median(int arr[], int n) {
    if (n % 2 == 0)
        return (arr[n/2] + arr[n/2 - 1]) / 2;
    else
        return arr[n/2];
}

/* This function returns median of arr1[] and arr2[].
Assumption: Both arr1[] and arr2[] are sorted arrays of same length. */
int getMedian(int arr1[], int arr2[], int n) {
    if (n <= 0)
        return -1;
    if (n == 1)
        return (arr1[0] + arr2[0]) / 2;
    if (n == 2)
        return (max(arr1[0], arr2[0]) + min(arr1[1], arr2[1])) / 2;
    int m1 = median(arr1, n);
    int m2 = median(arr2, n);
    if (m1 == m2)
        return m1;
    if (m1 < m2) {
        if (n % 2 == 0)
            return getMedian(arr1 + n/2 - 1, arr2, n - n/2 + 1);
        return getMedian(arr1 + n/2, arr2, n - n/2);
    }
    if (n % 2 == 0)
        return getMedian(arr2 + n/2 - 1, arr1, n - n/2 + 1);
    return getMedian(arr2 + n/2, arr1, n - n/2);
}
```

Above method doesn't work for arrays of unequal length. So, this could be a nice follow-up task: Write an algorithm which works for unequal sized arrays as well. Achieving this in logarithmic time is bit tricky. Here is our approach:

Start partitioning the two arrays into two groups of halves (not two parts, but both partitioned in such a way, that the partitioned sets should have same number of elements). The first half contains some first elements from first and second arrays, and the second half contains the rest (or the last) elements form first and second arrays. Because the arrays can be of different sizes, it does not mean to take half from each array. The below example clarifies the explanation.

Reach a condition such that, every element in the first half is less than or equal to every element in the second half. *How to reach this condition?*

Suppose, the total number of elements is even. Now let's say that we have found the partition (we will discuss partition procedure soon) such that a1 is less than or equal to a2, and b2 is less than or equal to b3.



Now check if a1 is less than or equal to b3, and if b2 is less than or equal to a2. If that's the case, it means that every element in the first half is less than or equal to every element in the second half. We can then directly calculate median using below formula:

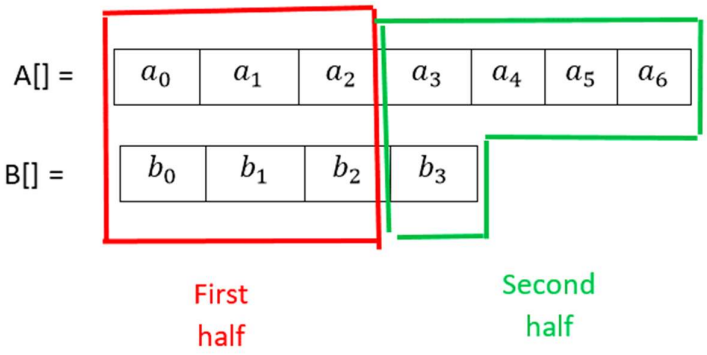
$$Median = \frac{\max(a1, b2) + \min(a2, b3)}{2} \quad in\ case\ of\ even\ numbers$$
$$= \max(a2, b2) \quad in\ case\ of\ odd\ numbers$$

But if that's not the case, then there are two possibilities:

a) If b2 > a2, then search on the right side of the array.
b) If a1 > b3, then search on the left side of the array.

Until above condition is found.

Here is the case of partition for odd elements case.



One should ask the question – *Why does the above condition lead to the median?*

The median is the $(n + 1)/2$ smallest element of the array, and here, the median is the $(n + m + 1)/2$ smallest element for the two arrays. If, all the elements in the first half are less than (or equal) to all elements in the second half, in case of odd numbers in total, just calculate the maximum between the last two elements in the first half (a2 and b2 in our example), and this will lead us to the $(n + m + 1)/2$ smallest element among the two arrays, which is the median $((7 + 4 + 1)/2 = 6)$. But in case of even numbers in total, calculate the average between the maximum of the last two elements in the first half (a1 and b2 in our example) with its successive number among the arrays which is the minimum of first two elements in the second half (a2 and b3 in our example). Now, here is how we make the partitions:

To make two halves, make the partition such that the index at partition of array A[] + the index at partition of array B[ ] is equal to the total number of elements plus one divided by 2, i.e. $(n + m + 1)/2$ (+1 is, if the total number of elements is odd).

First, define two variables: min_index and max_index, and initialize min_index to 0, and max_index to the length of the smaller array. In the examples given below, A[] is the smaller array.

To partition A[ ], use the formula: $i = \frac{\text{min}-index + max-inde}{2}$

To partition B[ ], use the formula: $j = \frac{(n+m+1)}{2} - i.$

The variable i means the number of elements to be inserted from A[ ] into the first half, and j means the number of elements to be inserted from B[ ] into the first half, the rest of the elements will be inserted into the second half.

A[] =	3	5	10	11	17

B[] =	9	13	20	21	23	27

Min index = 0
Max index = 5
i = (0+5)/2 = 2
j = (5+6+1)/2-2
= 4

First Iteration :

First half :
Elements from A[] : 3,5
Elements from B[] : 9,13,20,21

Second half :
Elements from A[] : 10,11,17
Elements from B[] : 23,27

When the number of elements is odd, we have an extra element in the first half :

5 <= 23 (True)
21 <= 10 (False)
It means that we have not reached the desired halves, so we need to search in the right, so min index = i + 1

Second Iteration :

First half :

Elements from A[] : 3,5,10,11

Elements from B[] : 9,13

Second half :

Elements from A[] : 17

Elements from B[] : 20,21,23,27

Min index = 3
Max index = 5
i = (3+5)/2 = 4
j = (5+6+1)/2-4 = 2

11 <= 20 (True)
13 <= 17 (True)
we have reached the desired halves, so we returning the max(11,13) which is the (n+m+1)/2 smallest element, i.e, the median if the two arrays

Below is another example which leads to the condition that returns a median that exists in the merged array.

A[] = | 2 | 3 | 5 | 8 |

B[] = | 10 | 12 | 14 | 16 | 18 | 20 |

First Iteration :

First half :

Elements from A[] : 2,3

Elements from B[] : 10,12,14

Second half :

Elements from A[] : 5,8

Elements from B[] : 16,18,20

Min index = 0
Max index = 4
i = (0+4)/2 = 2
j = (4+6+1)/2-2 = 3

3 <= 16 (True)
14 <= 5 (False)
It means that we have not reached the desired halves, so we need to search in the right, so min index = i + 1

Second Iteration :

First half :

Elements from A[] : 2,3,5

Elements from B[] : 10,12

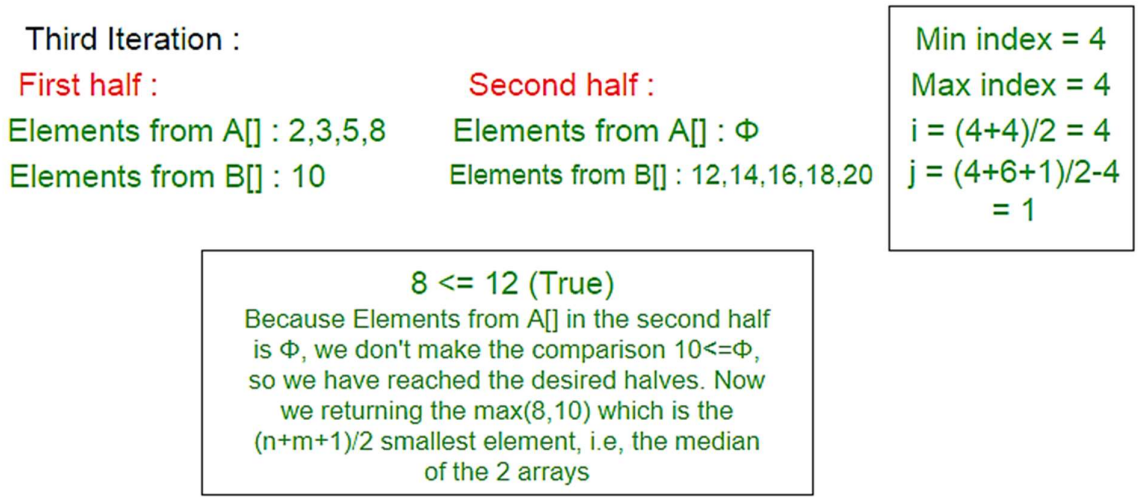
Second half :

Elements from A[] : 8

Elements from B[] : 14,16,18,20

Min index = 3
Max index = 4
i = (3+4)/2 = 3
j = (4+6+1)/2-3 = 2

5 <= 14 (True)
12 <= 8 (False)
It means that we have not reached the desired halves, so we need to search in the right, so min index = i + 1



Here is the code for above approach:

```
double findMedianSortedArrays(int *a, int n, int *b, int m) {
    int min_index = 0, max_index = n, i, j, median;
    while (min_index <= max_index) {
        i = (min_index + max_index) / 2;
        j = ((n + m + 1) / 2) - i;
        // If j is -ve, then the partition is not possible having i elements
        if (j < 0) {
            max_index = i-1;
            continue;
        }

        /* If i = n, it means that elements from a[] in the second half is
           an empty set. and if j = 0, it means that elements from b[] in
           the first half is an empty set. So, it is necessary to check that,
           because we compare elements from these two groups.

           Searching on right */
        if (i < n && j > 0 && b[j - 1] > a[i])
            min_index = i + 1;

        /* If i = 0, it means that Elements from a[] in the first half is
           an empty set and if j = m, it means that Elements from b[] in
           the second half is an empty set. So, it's necessary to check that,
           because we compare elements from these two groups.

           Searching on left */
        else if (i > 0 && j < m && b[j] < a[i - 1])
            max_index = i - 1;
```

```
            // we have found the desired halves.
            else {
                /* This condition happens when we don't have any elements in the
                   first half from a[] so we returning the last element in b[]
                   from the first half. */
                if (i == 0)
                    median = b[j - 1];

                /* and this condition happens when we don't have any elements
                   in the first half from b[] so we returning the last element
                   in a[] from the first half. */
                else if (j == 0)
                    median = a[i - 1];
                else
                    median = maximum(a[i - 1], b[j - 1]);
                break;
            }
        }
        if ((n + m) % 2 == 1)
            return (double)median;
        if (i == n)
            return (median+b[j]) / 2.0;
        if (j == m)
            return (median + a[i]) / 2.0;
        return (median + minimum(a[i], b[j])) / 2.0;
    }
```

Q. 4 Painter's Partition Problem.

We have to paint n boards of length {A1, A2, ... An}. There are k painters available and each takes 1-unit time to paint 1-unit of board. The problem is to find the minimum time to get this job done under the constraints that any painter will only paint continuous sections of boards, say board {2, 3, 4} or only board {1} or {2, 2} or nothing but not board {2, 4, 5}.

Examples:

Input: k = 2, A = {10, 10, 10, 10}

Output: 20.

Here we can divide the boards into 2 equal sized partitions, so each painter gets 20 units of board and the total time taken is 20.

Input: k = 2, A = {10, 20, 30, 40}

Output: 60.

Here we can divide first 3 boards for one painter and the last board for second painter.

Solution:

Let's forget about binary search for a moment. And let's try to solve this problem from scratch. We can observe that the problem can be broken down into: Given an array A of non-negative integers and a positive integer k, we have to divide A into k or fewer partitions such that the maximum sum of the elements in a partition, among overall partitions is minimized. So, for the second example above, possible divisions are:

- One partition: So, time is 100.

- Two partitions: (10) & (20, 30, 40), so time is 90. Similarly, we can put the first divider after 20 (=> time 70) or 30 (=> time 60); this means the minimum time from (100, 90, 70, 60) is 60.

A **brute force solution** is to consider all possible set of contiguous partitions and calculate the maximum sum partition in each case and return the minimum of all these cases.

1) Optimal Substructure:

We can implement the naive solution using recursion with the following optimal substructure property:

Assuming that we already have k-1 partitions in place (using k-2 dividers), we now have to put the k-1th divider to get k partitions. How can we do this? We can put the k-1th divider between the ith and i+1th element where i = 1 to n. Please note that putting it before the first element is the same as putting it after the last element.

The total cost of this arrangement can be calculated as the maximum of the following:
a) The cost of the last partition: sum(Ai ... An), where the k-1th divider is before element i.
b) The maximum cost of any partition already formed to the left of the k-1th divider.

Here a) can be found out using a simple helper function to calculate sum of elements between two indices in the array. How to find out b)?

We can observe that b) actually is to place the k-2 separators as fairly as possible, so it is a subproblem of the given problem. Thus, we can write the optimal substructure property as the following recurrence relation:

$$T(n,k) = min\left\{max_{i=1}^{n}\left\{T(i,k-1), \sum_{j=i+1}^{n} A_j\right\}\right\}$$

The base case are:

$$T(1,k) = A1$$

$$T(n,1) = \sum_{i=1}^{n} A_i$$

Following is the implementation of this recursive approach:

```
int sum(int arr[], int from, int to) {
    int total = 0;
    for (int i = from; i <= to; i++)
        total += arr[i];
    return total;
}

int partition(int arr[], int n, int k) {
    // base cases
    if (k == 1) // one partition
        return sum(arr, 0, n - 1);
    if (n == 1) // one board
        return arr[0];
    int best = INT_MAX;
    for (int i = 1; i <= n; i++)
        best = min(best, max(partition(arr, i, k - 1),  sum(arr, i, n - 1)));
    return best;
}
```

You can analyse that the time complexity of above solution is exponential. Following is the partial recursion tree for T(4, 3) in above equation.

```
          T(4, 3)

      /      /      \ ...

T(1, 2)  T(2, 2)  T(3, 2)

         /...        /...

     T(1, 1)     T(1, 1)
```

We can observe that many subproblems like T(1, 1) in the above problem are being solved again and again. Because of these two properties of this problem, we can solve it using dynamic programming, either by top down memoized method or bottom up tabular method.

```
// bottom up tabular dp
int partition(int arr[], int n, int k) {
    int dp[k + 1][n + 1] = { 0 }; // initialize table
    // base cases
    // k=1
    for (int i = 1; i <= n; i++)
        dp[1][i] = sum(arr, 0, i - 1);
    // n=1
    for (int i = 1; i <= k; i++)
        dp[i][1] = arr[0];
```

```
    // 2 to k partitions
    for (int i = 2; i <= k; i++) { // 2 to n boards
        for (int j = 2; j <= n; j++) {
            // track minimum
            int best = INT_MAX;
            // i-1 th separator before position arr[p = 1…j]
            for (int p = 1; p <= j; p++)
                best = min(best, max(dp[i - 1][p], sum(arr, p, j - 1)));
            dp[i][j] = best;
        }
    }
    return dp[k][n];
}
```

The time complexity of the above program is $O(k * N^3)$. It can be easily brought down to $O(k * N^2)$ by precomputing the cumulative sums in an array thus avoiding repeated calls to the sum function:

```
int partition(int arr[], int n, int k) {
    int dp[k + 1][n + 1] = { 0 };

    // calculate prefix sum
    int sum[n+1] = {0};
    for (int i = 1; i <= n; i++)
        sum[i] = sum[i-1] + arr[i-1];

    // base cases
    for (int i = 1; i <= n; i++)
        dp[1][i] = sum[i];
    for (int i = 1; i <= k; i++)
        dp[i][1] = arr[0];

    for (int i = 2; i <= k; i++) { // 2 to k partitions
        for (int j = 2; j <= n; j++) { // 2 to n boards
            // track minimum
            int best = INT_MAX;
            // i-1 th separator before position arr[p=1…j]
            for (int p = 1; p <= j; p++)
                best = min(best, max(dp[i-1][p], sum[j] - sum[p]));
            dp[i][j] = best;
        }
    }
    return dp[k][n];
}
```

Time Complexity: $O(k * N^2)$ and Space Complexity: $O(k * N)$

Now, let's look at this problem – with the view of binary search. We know that the invariant of binary search has two main parts:

a) the target value would always be in the searching range.
b) the searching range will decrease in each loop so that the termination can be reached.

We also know that the values in this range must be in sorted order. Here our target value is the maximum sum of a contiguous section in the optimal allocation of boards. Now how can we apply binary search for this? We can fix the possible low to high range for the target value and narrow down our search to get the optimal allocation.

We can see that the highest possible value in this range is the sum of all the elements in the array and this happens when we allot 1 painter all the sections of the board. The lowest possible value of this range is the maximum value of the array max, as in this allocation we can allot max to one painter and divide the other sections such that the cost of them is less than or equal to max and as close as possible to max. Now if we consider we use x painters in the above scenarios, it is obvious that as the value in the range increases, the value of x decreases and vice-versa. From this we can find the target value when x = k and use a helper function to find x, the minimum number of painters required when the maximum length of section a painter can paint is given.

```
int getMax(int arr[], int n) {
    int max = INT_MIN;
    for (int i = 0; i < n; i++)
        if (arr[i] > max) max = arr[i];
    return max;
}

int getSum(int arr[], int n) {
    int total = 0;
    for (int i = 0; i < n; i++) total += arr[i];
    return total;
}

// find minimum required painters for given maxlen
// which is the maximum length a painter can paint
int numberOfPainters(int arr[], int n, int maxLen) {
    int total = 0, numPainters = 1;
    for (int i = 0; i < n; i++) {
        total += arr[i];
        if (total > maxLen) {
            total = arr[i]; numPainters++;
        }
    }
    return numPainters;
}
```

```
int partition(int arr[], int n, int k) {
    int low = getMax(arr, n);
    int high = getSum(arr, n);
    while (low < high) {
        int mid = low + (high - low) / 2;
        int requiredPainters = numberOfPainters(arr, n, mid);
        // find better optimum in lower half. Here mid is included
        // because we may not get anything better.
        if (requiredPainters <= k)
            high = mid;
        // find better optimum in upper half. Here mid is excluded
        // because it gives required Painters > k, which is invalid
        else
            low = mid + 1;
    }
    return low;
}
```

Time Complexity: $O(N * \log(sum(arr[]))$

Q. 5 Aggressive Cows Problem. Farmer John has built a new long barn, with N (2 <= N <= 100,000) stalls. The stalls are located along a straight line at positions x_1, ..., x_N (0 <= x_i <= 1,000,000,000). His C (2 <= C <= N) cows don't like this barn's layout and become aggressive towards each other once put into a stall. To prevent the cows from hurting each other, FJ wants to assign the cows to the stalls, such that the minimum distance between any two of them is as large as possible. What is the largest minimum distance?

Input:
t – the number of test cases, then t test cases follows.
* Line 1: Two space-separated integers: N and C
* Lines 2 … N+1: Line i+1 contains an integer stall location, x_i

Output:
For each test case output one integer: the largest minimum distance.

Example:
1
5 3
1
2
8
4
9
Ans = 3

Explanation: FJ can put his 3 cows in the stalls at positions 1, 4 and 8, resulting in a minimum distance of 3.

Solution: It is clear, that we shall first sort the stalls' positions. Now, we need to put C cows in these N stalls. Let's work on above example to understand our approach.

1	2	4	8	9

Now, let's say we have just 2 cows. The minimum distance possible is 0 (you put both the cows in same stall) and the maximum distance possible is 9 – 1 = 8 (you put both of them in extreme stalls). But now we have 'C' cows. Here it's 3 in our example. So, in brute force, we check – Can we place 3 cows at distance 1 away from each other? If yes, then can we place 3 cows at distance 2 away from each other? If yes, then can we do so for distance 3? We keep on doing this till 8. We then print the maximum possible distance.

But instead of incrementing the distance by 1 and checking it for the next stall, we can use binary search here. Look that we are not applying binary search to any array or sequence, but we are using to get our optimal guess in fastest possible time. We check for possible distance = 4. If no, then search from 0 to 3 and if yes, then search from 5 to 8. This is the abstraction of idea of binary search.

```
typedef long long int ll;

bool check(int cows, ll* positions, int n, ll distance) {
    int count = 1;
    ll last_position = positions[0];
    for (int i = 1; i < n; i++) {
        if (positions[i] - last_position >= distance) {
            last_position = positions[i];
            count++;
        }
        if (count == cows)
            return true;
    }
    return false;
}

int main() {
    int t; cin >> t;
    while (t--) {
        int n, c; cin >> n >> c;
        ll positions[n];
        for (int i = 0; i < n; i++)
            cin >> positions[i];
        sort (positions, positions+n);
```

```
        ll start = 0, end = positions[n-1] - positions[0], ans = -1;
        while (start <= end) {
            ll mid = start + (end-start)/2;
            if (check(c, positions, n, mid)) {
                ans = mid; start = mid+1;
            } else
                end = mid-1;
        }
        cout << ans << endl;
    }
    return 0;
}
```

Q. 6 Candy Distribution Problem. Shaky has N (1<=N<=50000) candy boxes each of them contains a non-zero number of candies (between 1 and 1000000000). Shaky want to distribute these candies among his K (1<=K<=1000000000) friends. He wants to distribute them in a way such that:
1. All friends get equal number of candies.
2. All the candies which a friend gets must be from a single box only.
As he wants to make all of them happy so he wants to give as many candies as possible. Help Shaky in finding out what is the maximum number of candies which a student can get.

Input:
First line contains 1<=T<=20 the number of test cases. Then T test cases follow. First line of each test case contains N and K. Next line contains N integers, i^{th} of which is the number of candies in i^{th} box.

Output:
For each test case print the required answer in a separate line.

Example:
2
3 2
3 1 4
4 1
3 2 3 9

Ans:
3
9

Solution: The idea is similar to aggressive cows. Find the possible range of your answer and then apply binary search to find the optimal possible answer in fewest guesses.

```
typedef long long int ll;

bool check(int students, ll* candies, int n, ll distribution) {
    int count = 0;
    //ll last_position = positions[0];
    for (int i = 0; i < n; i++) {
        if (candies[i] - distribution >= 0) {
            count += candies[i]/distribution;
        }
        if (count >= students)
            return true;
    }
    return false;
}

int main() {
    int t; cin >> t;
    while (t--) {
        int n, k; cin >> n >> k;
        ll candies[n];
        for (int i = 0; i < n; i++)
            cin >> candies[i];

        sort (candies, candies+n);

        ll min = 0, max = candies[n-1], ans = 0;
        while (min <= max) {
            ll mid = min + (max-min)/2;
            if (check(k, candies, n, mid)) {
                ans = mid; min = mid+1;
            } else
                max = mid-1;
        }
        cout << ans << endl;
    }
    return 0;
}
```

Chapter 3 – Divide and Conquer

3.1 Introduction

This is a technique of solving a problem by dividing them into smaller and smaller subproblems until solving each subproblem becomes very easy and then merging/conquering the results of subproblem in such a way that it gives us back the result of the original problem. Most of the times, it is the merge step which is tricky to think about. We have been using many 'divide and conquer' algorithms by now (in Data Structures and in this book). Here is the quick review of that:

a) Binary Search: Here you keep on dividing your sequence into halves until your subproblem ends up to one cell of the array. This is a very basic DAC algorithm as it doesn't require a tricky merge step.
b) Mergesort: This is a classic example of DAC. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.
c) Quicksort: The algorithm first places the pivot element at its correct position and then divides the problem into left and right subarray. Here again, there is no explicit merge step.

In this chapter, we will be seeing few of the very famous problems which were solved efficiently using DAC approach.

3.2 Irrational Numbers

Many times, we encounter a floating-point inaccuracy in the value, because 64 bits were not sufficient enough to store a float value beyond a certain number of digits. So, if I wish to compute 1000 digits of π, or 2000 digits of $\sqrt{2}$, how will I do that? So, our problem here is – to calculate an irrational number up to 'd' digits of accuracy, and here 'd' can be any arbitrary number – say 1000.

We will come back to this problem in a while. For now, let's talk about something completely different – Catalan numbers. The definition of n^{th} Catalan number is –

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

or, we can use recursive definition:

$$C_0 = 1 \quad and \quad C_{n+1} = \sum_{i=0}^{n} C_i \times C_{n-i} \qquad for\ n \geq 0.$$

Here are some first Catalan numbers for n = 0, 1, 2, 3, … and so on.

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, …

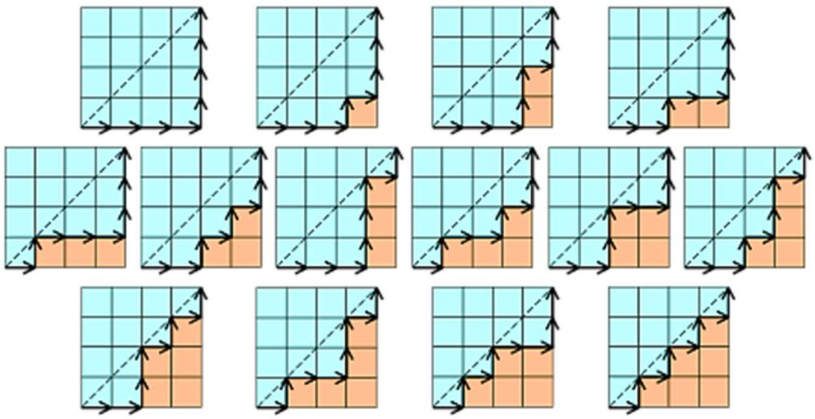
Catalan numbers find many applications in combinatorics. For example,

a) If you have 'n' pairs of parentheses, then there are C_n ways of writing expressions in which the parentheses are correctly matched. For e.g., n = 3 gives $C_n = 5$ and the 5 ways are: ((())) ()(()) ()()() (())() (()()).

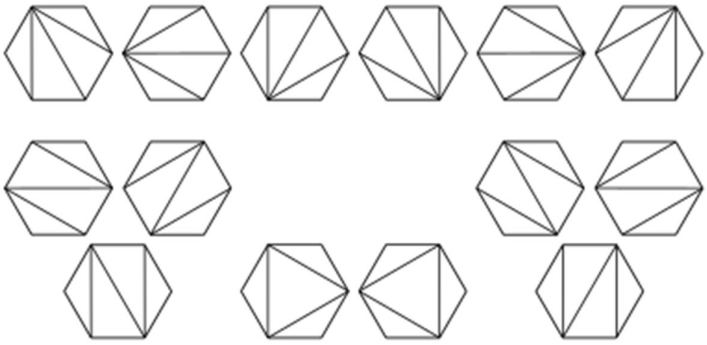
b) The number of full binary trees with n+1 leaf nodes is nth Catalan number.

c) The number of BSTs with n nodes is nth Catalan number.

d) C_n is the number of monotonic lattice paths along the edges of a grid with n × n square cells, which do not pass above the diagonal. A monotonic path is one which starts in the lower left corner, finishes in the upper right corner, and consists entirely of edges pointing rightwards or upwards.



e) A convex polygon with n + 2 sides can be cut into triangles by connecting vertices with non-crossing line segments (a form of polygon triangulation). The number of triangles formed is n and the number of different ways that this can be achieved is C_n.



f) C_n is the number of permutations of {1, ..., n} that avoid the permutation pattern 123 (or, alternatively, any of the other patterns of length 3); that is, the number of permutations with no three-term increasing subsequence. For n = 3, these permutations are 132, 213, 231, 312 and 321. For n = 4, they are 1432, 2143, 2413, 2431, 3142, 3214, 3241, 3412, 3421, 4132, 4213, 4231, 4312 and 4321.

So, we can see, Catalan numbers are everywhere. The reason we studied a little about Catalan numbers is because we will see them once again in context of irrational numbers.

Coming back to our problem of finding irrational numbers up to 'd' digits of precision, say $\sqrt{2}$ for example, we know we can't rely on computer system, because it will only give us the precision which could be stored in 64 bits. So, what we can do is – we can use numerical methods of finding roots of equation $x^2 - 2 = 0$ and then approximate the root to 'd' digits of accuracy. And the numerical method which we shall be using is Newton's method. According to this method, the successive approximation of root at i^{th} iteration is given by,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

And if we consider $f(x) = x^2 - a,$ then –

$$x_{i+1} = x_i - \frac{x_i^2 - a}{2x_i} = \frac{x_i + a/x_i}{2}$$

One thing to notice is – we need to compute x_i to 'd' digits of precision and we require a division operation here, a/x_i. Again, the computer can't perform such accurate division because the word length of ALU is just 64 bits. So, we need to somehow manage this division operation, persisting the accuracy. To do so, we first need to understand how ALU carries out division in modern chips. Inside a chip, a division operation is performed by using multiplication as a subroutine. So, in order to perform efficient division, we need to perform efficient multiplication first.

Efficient Multiplication:

The time complexity of multiplication of two numbers each 'n' bits long is $O(n^2)$. But it seems that we can do better. Let's solve this by considering a problem.

Given two binary strings that represent value of two integers, find the product of two strings. For example, if the first bit string is "1100" and second bit string is "1010", output should be 120.

Approach:

For simplicity, let the length of two strings be same and be n.

A Naive Approach is to follow the process we study in school. One by one take all bits of second number and multiply it with all bits of first number. Finally add all multiplications. This algorithm takes $O(n^2)$ time.

Using Divide and Conquer, we can multiply two integers in lesser time complexity. We divide the given numbers in two halves. Let the given numbers be X and Y. For simplicity let us assume that n is even.

$X = X_l \times 2^{\frac{n}{2}} + X_r$, where X_l and X_r contain leftmost and rightmost $n/2$ bits of X.

$Y = Y_l \times 2^{\frac{n}{2}} + Y_r$, where Y_l and Y_r contain leftmost and rightmost $n/2$ bits of Y.

The product XY can be written as following.

$$XY = (X_l \times 2^{\frac{n}{2}} + X_r)(Y_l \times 2^{\frac{n}{2}} + Y_r)$$

$$= 2^n X_l Y_l + 2^{\frac{n}{2}}(X_l Y_r + X_r Y_l) + X_r Y_r$$

If we take a look at the above formula, there are four multiplications of size n/2, so we basically divided the problem of size n into four sub-problems of size n/2. But that doesn't help because solution of recurrence $\mathsf{T(n)} = 4\mathsf{T(n/2)} + \mathsf{O(n)}$ is $O(n^2)$. The tricky part of this algorithm is to change the middle two terms to some other form so that only one extra multiplication would be sufficient. The following is tricky expression for middle two terms.

$$X_l Y_r + X_r Y_l = (X_l + X_r)(Y_l + Y_r) - X_l Y_l - X_r Y_r$$

So, the final value of XY becomes

$$XY = 2^n X_l Y_l + 2^{\frac{n}{2}}[(X_l + X_r)(Y_l + Y_r) - X_l Y_l - X_r Y_r] + X_r Y_r$$

With above trick, the recurrence becomes $\mathsf{T(n)} = 3\mathsf{T(n/2)} + \mathsf{O(n)}$ and solution of this recurrence is $O(n^{\log_2 3}) = O(n^{1.59})$.

What if the lengths of input strings are different and are not even? To handle this case of different length, we append 0's in the beginning of shorter one. To handle odd length, we put floor(n/2) bits in left half and ceil(n/2) bits in right half. So, the expression for XY changes to following.

$$XY = 2^{2ceil(n/2)} X_l Y_l + 2^{cei\ (n/2)}[(X_l + X_r)(Y_l + Y_r) - X_l Y_l - X_r Y_r] + X_r Y_r$$

The above algorithm is called **Karatsuba algorithm** and it can be used for any base.

```
// Given two unequal sized bit strings, converts them to same length
// by adding leading 0s in the smaller string. Returns the new length.
int makeEqualLength(string &str1, string &str2) {
    int len1 = str1.size();
    int len2 = str2.size();
    if (len1 < len2) {
        for (int i = 0; i < len2 - len1; i++)
            str1 = '0' + str1;
        return len2;
    }
    else if (len1 > len2) {
        for (int i = 0; i < len1 - len2; i++)
            str2 = '0' + str2;
    }
    return len1; // If len1 >= len2
}
```

```
// Function that adds two bit sequences and returns the addition
string addBitStrings(string first, string second) {
    string result;  // To store the sum bits
    // make the lengths same before adding
    int length = makeEqualLength(first, second);
    int carry = 0;  // Initialize carry
    // Add all bits one by one
    for (int i = length-1 ; i >= 0 ; i--) {
        int firstBit = first.at(i) - '0';
        int secondBit = second.at(i) - '0';
        // boolean expression for sum of 3 bits
        int sum = (firstBit ^ secondBit ^ carry)+'0';
        result = (char)sum + result;
        // boolean expression for 3-bit addition
        carry = (firstBit & secondBit) | (secondBit & carry) |
                (firstBit & carry);
    }
    // if overflow, then add a leading 1
    if (carry)  result = '1' + result;
    return result;
}

// A utility function to multiply single bits of strings a and b
int multiplyiSingleBit(string a, string b)
{  return (a[0] - '0')*(b[0] - '0');  }

// The main function that multiplies two bit-strings X and Y
// and returns result as long integer
long int multiply(string X, string Y) {
    // Find the maximum of lengths of x and Y and make length
    // of smaller string same as that of larger string
    int n = makeEqualLength(X, Y);
    // Base cases
    if (n == 0) return 0;
    if (n == 1) return multiplyiSingleBit(X, Y);
    int fh = n/2;    // First half of string, floor(n/2)
    int sh = (n-fh); // Second half of string, ceil(n/2)
    // Find the first half and second half of first string.
    string Xl = X.substr(0, fh); string Xr = X.substr(fh, sh);
    // Find the first half and second half of second string
    string Yl = Y.substr(0, fh); string Yr = Y.substr(fh, sh);
    // Recursively calculate the three products of inputs of size n/2
    long int P1 = multiply(Xl, Yl); long int P2 = multiply(Xr, Yr);
    long int P3 = multiply(addBitStrings(Xl, Xr),addBitStrings(Yl, Yr));
    // Combine the three products to get the final result.
    return P1*(1<<(2*sh)) + (P3 - P1 - P2)*(1<<sh) + P2;
}
```

```
int main() {
    printf ("%ld\n", multiply("1100", "1010"));
    printf ("%ld\n", multiply("110", "1010"));
    printf ("%ld\n", multiply("11", "1010"));
    printf ("%ld\n", multiply("1", "1010"));
    printf ("%ld\n", multiply("0", "1010"));
    printf ("%ld\n", multiply("111", "111"));
    printf ("%ld\n", multiply("11", "11"));
    return 0;
}
```

High-precision division:

We want to compute 1/b as any fraction a/b can be computed as a*(1/b). So, our focus will be on finding the reciprocals.

Suppose we have a reciprocal $\frac{1}{b}$, then to find high precision, what we want to do is – shift the less significant bits to right by multiplying it with R, where R is some big number which is a power of 2. So, R = 2^k where k is large. You can easily imagine this idea by seeing an example in base 10 arithmetic, ¼ = 0.25. But let's say, your computer system cannot store fractions, but only integers. Then how will you find the fractional part? Simply by multiplying the result by 100. So, you find 100/4 = 25. Since we were working in base 10, here R = 10^2 is a power of 10. And multiplying with R is easy as it requires just the shifting of bits by using shift operator. But the question still remains – how to divide by 'b' in R/b?

The answer to this question is – We will convert this division by 'b' to division by 'R' because division by R will be easy (after all its again shifting of bits in opposite direction). Question is – how will we make this conversion? To do so, we will use Newton's method again.

$$f(x) = \frac{1}{x} - \frac{b}{R}$$

Here, the root of this equation is x = R/b. Plugging this f(x) in Newton's method,

$$x_{i+1} = x_i - \frac{\left(\frac{1}{x_i} - \frac{b}{R}\right)}{-\frac{1}{x_i^2}}$$

After simplifying above expression, we get –

$$x_{i+1} = 2x_i - \frac{bx_i^2}{R}$$

Now if you look at above equation, $2x_i$ is easy, because we can multiply efficiently using Karatsuba (or we can even use naïve algorithm) – so bx_i^2 is easy, subtraction is easy and division by R is easy. So, using above successive formula, we can find R/b.

To get d digits of precision, we will require only $\log_2 d$ iterations. So, if $O(n^\alpha)$ is the time complexity of multiplication, then the time complexity of division is $O(n^\alpha \log_2 n)$. For Karatsuba multiplication, $\alpha = 1.59$.

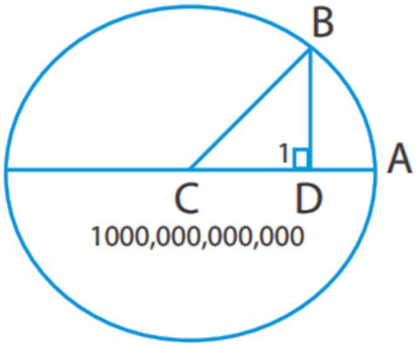
So, finally, here are the steps which we need to perform for high precision square root of 2:

For sqrt of 2 upto 'd' digits of precision, we need to solve, $x^2 - 2 * 10^{2d} = 0$. This will give us all d digits into integral part, so no decimal inaccuracy. However, storing such big 'd' digit integers isn't possible in 'long long' data-type as well. So, for that, we need to represent these integers as strings by making a class called 'BigInteger'. We now need to redefine all the operators, so that we can do arithmetic BigIntegers. One important operation will be, multiplying two BigIntegers. This is same as "multiplying two string representations of numbers". Hey! That's Karatsuba's Algorithm! We then solve the above equation using Newton's method.

Here is the demo of sqrt(2) implemented using above method:

http://people.csail.mit.edu/devadas/numerics_demo/sqrt2.html

Now that you have looked at the demo of sqrt of 2, let's try to compute one high precision calculation. Suppose, we have a circle of diameter equals to 1 trillion.



If BD = 1, then what is AD?

AD = AC – CD = $500{,}000{,}000{,}000 - \sqrt{(500{,}000{,}000{,}000^2 - 1)}$

Look at the result here:

http://people.csail.mit.edu/devadas/numerics_demo/chord.html

Well, the only numbers you see in the answer are – just Catalan numbers! For further explanation, refer:

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec11.pdf

3.3 Fast Exponentiation

Exponentiation is a mathematical operation expressed as x^m and computed as $x^m = x.x...x\ (m\ times)$. The time complexity of this basic approach is $O(n)$. But it seems we can do a lot better as this method is not suitable when m gets very large (and we know now that multiplication operation isn't easy to perform). The efficient way is to use – **Binary Exponentiation.**

If n is even, then x^n can be broken down to $(x^2)^{\frac{n}{2}}$. Programmatically finding x^2 is one step process. So, the computation steps have now been reduced to $n/2$ in just one step. If n is odd, we write $x^n = x.x^{n-1}$ and then follow above process on x^{n-1}. So eventually, we will end up doing $O(\log n)$ work.

First the recursive approach, which is a direct translation of the recursive formula:

```
long long binaryPower(long long x, long long n) {
    if (n == 0)
        return 1;
    long long res = binpow(x, n/2);
    if (n % 2)
        return res * res * x;
    else
        return res * res;
}
```

The second approach accomplishes the same task without recursion. It computes all the powers in a loop, and multiplies the ones with the corresponding set bit in n. Although the complexity of both approaches is identical, this approach will be faster in practice since we have the overhead of the recursive calls.

```
long long binaryPower(long long x, long long n) {
    long long res = 1;
    while (n > 0) {
        if (n & 1)
            res = res * x;
        x = x * x;
        n >>= 1;
    }
    return res;
}
```

However, storing such large exponentiation values isn't possible. So, many times we store modular results instead. Typically, we take mod of result using $M = 10^9 + 7$. Following is the iterative version of modular exponentiation logic.

```
int modularExponentiation(int x, int n, int M) {
    int result=1;
    while(n > 0) {
        if(n & 1)
            result = (result * x) % M;
        x = (x*x) % M;
        n = n/2;
    }
    return result;
}
```

The idea of binary exponentiation is not just valid upto numbers. We can use the above idea in matrix exponentiation. This reduces the time complexity from $O(n^4)$ to $O(n^3 \log n)$.

```
#define ll long long
const int MOD = 1e9 + 7;

// Helper functions
vector<vector<ll>> identity(int n) {
    vector<vector<ll>> values(n, vector<ll>(n, 0));
    for (int i = 0; i < n; i++)
        values[i][i] = 1;
    return values;
}

vector<vector<ll>> multiply(vector<vector<ll>> mat1, vector<vector<ll>>
mat2) {
    int rows1 = mat1.size(), cols1 = mat1[0].size(),
        cols2 = mat2[0].size();
        vector<vector<ll> > result(rows1, vector<ll>(cols2, 0));
        for (int i = 0; i < rows1; i++)
            for (int j = 0; j < cols2; j++)
                for (int k = 0; k < cols1; k++)
                    result[i][j] = (result[i][j]+mat1[i][k]*1ll*
                                    mat2[k][j]) % MOD;
        return result;
}

bool isSquare(vector<vector<ll>> mat) {
    int n = mat.size();
    for (int i = 0; i < n; i++)
        if (mat[i].size() != n)
            return false;
    return true;
}
```

```cpp
void printMatrix (vector<vector<ll>> mat) {
    for (int i = 0; i < mat.size(); i++) {
        for (int j = 0; j < mat[0].size(); j++) {
            cout << mat[i][j] << " ";
        }
        cout << endl;
    }
}

// Main logic of the code
vector<vector<ll>> fastExponentiation(vector<vector<ll>> mat, int power)
 {
    if (!isSquare(mat)) {
        cout << "Can't use exponentiation on rectangular matrix.\n";
        exit(0);
    }
    vector<vector<ll>> result = identity(mat.size());
    while (power) {
        if (power & 1)
            result = multiply(result, mat);
        mat = multiply(mat, mat);
        power >>= 1;
    }
    return result;
}

int main() {
    vector<vector<ll>> matrix = {
        {4, 0, -1},
        {2, -3, 0},
        {0, 1, -2}
    };
    int n = 10;
    vector<vector<ll>> result = fastExponentiation(matrix, n);
    printMatrix(result);
    return 0;
}
```

Applications of Binary/Fast Exponentiation:

1. Finding Nth Fibonacci number

Fibonacci numbers F_n is defined as,

$$F_0 = F_1 = 1$$
$$F_i = F_{i-1} + F_{i-2}$$

The first idea that comes to mind for calculating Fibonacci numbers is to run a for loop. The running time of this method is $O(n)$. This method is reasonably good when $n < 10^9$. If we want n to be upto 10^{18}, we need to switch to faster method.

Suppose we have a vector (matrix with one row and several columns) of (F_{i-2}, F_{i-1}) and we want to multiply it by some matrix M, so that we get (F_{i-1}, F_i) Let's call this matrix M:

$$(F_{i-2} \quad F_{i-1}) * M = (F_{i-1} \quad F_i)$$

Two questions arise immediately:

a) What are the dimensions of M?
b) What are exact values in M?

$(F_{i-2} \quad F_{i-1})$ is 1×2 matrix and the resultant matrix is also 1×2. So the dimensions of M should be 2×2. Suppose,

$$M = \begin{pmatrix} x & y \\ z & w \end{pmatrix}$$

$$(F_{i-2} \quad F_{i-1}) * \begin{pmatrix} x & y \\ z & w \end{pmatrix} = (F_{i-1} \quad F_i)$$

$$(x * F_{i-2} + z * F_{i-1} \quad y * F_{i-2} + w * F_{i-1}) = (F_{i-1} \quad F_i)$$

Clearly, using definition of Fibonacci numbers, we can say –

$$M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

Initially, we have F_0 and F_1. Arrange them as a vector:

$$(F_0 \quad F_1) = (1\ 1)$$

Multiplying this vector with the matrix M will get us to $(F_1, F_2) = (1, 2)$:

$$(1 \quad 1) * M = (1 \quad 2)$$

If we multiply (1, 2) with M, we get $(F_2, F_3) = (2, 3)$:

$$(1 \quad 2) * M = (2 \quad 3)$$

But we could get the same result by multiplying (1, 1) by M two times:

$$(1 \quad 1) * M * M = (2 \quad 3)$$

In general, multiplying k times by M gives us $(F_k, F_k + 1)$:

$$(1 \quad 1) * M^k = (F_k \quad F_k + 1)$$

Computing M^k takes $O((\text{size of } M)^3 * \log(k))$ time. In our problem, size of M is 2, so we can find N^{th} Fibonacci number in $O(2^3 * \log(N)) = O(\log N)$.

2. Linear Recurrent Sequences

Let's take a look at more general problem than before. Sequence A is a linear recurrent sequence if it satisfies two properties:

a) $A_i = c_1 A_{i-1} + c_2 A_{i-2} + c_3 A_{i-3} + \cdots + c_k A_{i-k}$ for $i > k$.

b) $A_0 = a_0, A_1 = a_1, \ldots A_{k-1} = a_{k-1}$ are given integers.

We need to find A_N modulo 1000000007, when **N** is up to 10^{18} and **k** up to 50.

Here we will look for a solution that involves matrix multiplication right from the start. If we obtain matrix **M**, such that:

$$\begin{pmatrix} A_{i-k} & A_{i-k+1} & \ldots & A_{i-2} & A_{i-1} \end{pmatrix} * M = \begin{pmatrix} A_{i-k+1} & A_{i-k+2} & \ldots & A_{i-1} & A_i \end{pmatrix}$$

then we can get A_N in the following manner:

$$\begin{pmatrix} a_0 & a_1 & \ldots & a_{k-2} & a_{k-1} \end{pmatrix} * M^{N-k+1} = \begin{pmatrix} A_{N-k+1} & A_{N-k+2} & \ldots & A_{N-1} & A_N \end{pmatrix}$$

The reasoning is the same as with Fibonacci numbers: we multiply matrix with 1 row and k columns by M and get matrix with 1 row and k columns. Therefore, M has k rows and k columns.

$$M = \begin{pmatrix} x_{11} & x_{12} & x_{13} & \ldots & x_{1k} \\ x_{21} & x_{22} & x_{23} & \ldots & x_{2k} \\ x_{31} & x_{32} & x_{33} & \ldots & x_{3k} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ x_{k1} & x_{k2} & x_{k3} & \ldots & x_{kk} \end{pmatrix}$$

Then we can write down equations, which are based on the definition of matrix multiplication:

$$\begin{aligned} A_{i-k} * x_{11} + A_{i-k+1} * x_{21} + \ldots + A_{i-1} * x_{k1} &= A_{i-k+1} \\ A_{i-k} * x_{12} + A_{i-k+1} * x_{22} + \ldots + A_{i-1} * x_{k2} &= A_{i-k+2} \\ &\ldots \\ A_{i-k} * x_{1k} + A_{i-k+1} * x_{2k} + \ldots + A_{i-1} * x_{kk} &= A_i \end{aligned}$$

From the first equation it is easy to see, that $x_{21} = 1$ and $x_{i1} = 0$; $i = 1, 3, 4, \ldots, k$. From the second equation we conclude that $x_{32} = 1$ and $x_{i2} = 0$; $i = 1, 2, 4, \ldots, k$. Following this logic up to k-1'th equation, we get $x_{i,i-1} = 1$; $i = 2, 3, \ldots, k$ and $x_{ij} = 0$; $i = 1, 2, 3, \ldots, k$ and $j \le k - 1$, $j \ne i - 1$. The last equation looks like the definition of A_i. Based on that, we get $x_{ik} = c_{k-i+1}$:

$$M = \begin{pmatrix} 0 & 0 & 0 & \ldots & 0 & c_k \\ 1 & 0 & 0 & \ldots & 0 & c_{k-1} \\ 0 & 1 & 0 & \ldots & 0 & c_{k-2} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & 0 & \ldots & 1 & c_1 \end{pmatrix}$$

This code will run in $O(k^3 \log N)$ if we use fast matrix exponentiation.

Let's go back to Fibonacci numbers. Introduce a new sequence:
$P_i = F_0 + F_1 + F_2 + \cdots + F_i$ (i.e. sum of first i Fibonacci numbers).

The problem is: find P_N modulo 1000000007 for **N** up to 10^{18}. Surprisingly, we can do it with matrices again. Let's imagine we have a matrix **M**, such that:

$$\begin{pmatrix} P_{i-1} & F_{i-2} & F_{i-1} \end{pmatrix} * M = \begin{pmatrix} P_i & F_{i-1} & F_i \end{pmatrix}$$

Then we can obtain P_N by doing:

$$\begin{pmatrix} P_1 & F_0 & F_1 \end{pmatrix} * M^{N-1} = \begin{pmatrix} P_N & F_{N-1} & F_N \end{pmatrix}$$ where

$$M = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

The first column is three 1's because $P_i = P_{i-1} + F_i = P_{i-1} + F_{i-2} + F_{i-1}$. You can find sum of first N numbers of any recurrent linear sequence A_i, not just Fibonacci**.** To do that, introduce the sum sequence $P_i = A_0 + A_1 + \ldots + A_i$. Put this sequence into the initial vector. Then find the matrix that gives you P_{i+1}, based on P_i and some A's.

3. Number of paths of length 'k' in directed graph

Given a directed, unweighted graph with N vertices and an integer k. The task is to find the number of paths of length k for each pair of vertices (u, v). Paths don't have to be simple i.e. vertices and edges can be visited any number of times in a single path.

The graph is represented as adjacency matrix where $G[i][j] = 1$ indicates that there is an edge from vertex i to vertex j and $G[i][j] = 0$ indicates no edge from i to j. It is obvious that given adjacency matrix is the answer to the problem for the case k = 1. It contains the number of paths of length 1 between each pair of vertices.

Let's assume that the answer for some k is Mat_k and the answer for $k + 1$ is Mat_{k+1}.

$$Mat_{k+1}[i][j] = \sum_{k=1}^{N} Mat_k[i][k] * G[k][j]$$

It is easy to see that the formula computes nothing other than the product of the matrices Mat_k and G. So,

$$Mat_{k+1} = Mat_k * G$$

Thus, the solution of the problem can be represented as,

$$Mat_k = G * G * \ldots * G \ (k\ times) = G^k$$

Problem

PK is an astronaut who went to a planet and was very fascinated by the language of its creatures. The language contains only lowercase English letters and is based on a simple logic that only certain characters can follow a particular character. Now he is interested in calculating the number of words of length **L** and ending at a particular character **C**. Help PK to calculate this value.

Input:
The input begins with **26** lines, each containing **26** space-separated integers. The integers can be either 0 or 1. The jth integer at **ith** line depicts whether **jth** English alphabet can follow **ith** English alphabet or not.
Next line contains an integer **T**. **T** is the number of queries.
Next **T** lines contains a character **C** and an integer **L**.

Output:
For each query output the count of words of length **L** ending with the character **C**. Answer to each query must be followed by newline character.
The answer may be very large so print it modulo 1000000007.

Constraints:
1 <= **T** <= 100
C is lowercase English alphabet.
2 <= **L** <= 10000000

Solution:
Complexity (per query): $O(z^3 \log l)$ where z is the number of alphabets and l is the length of word.
Explanation: Suppose we have a graph having each English alphabet as a vertex. There is an edge between the ith and jth English alphabet if the entry a[i][j] = 1, where a is the input matrix. Now each word in the language is simply a path from the starting alphabet to the ending alphabet. To calculate the numbers of words of length l ending at particular alphabet, we need to calculate total paths of length l-1 ending at that alphabet. This can be found by raising the adjacency matrix to the power l-1. The jth number in the ith row of this matrix gives the number of words of length l starting at character i and ending at character j. To find the total number of words ending at a particular alphabet take the sum of all the numbers in the jth column.

3.4 Fast Fourier Transform

This topic, in particular, is my favourite topic in the context of Divide and Conquer strategy. It's tricky and contains many *Aha!* moments. So, we will go through this topic slowly and steadily, building the base and then finally we shall deal with little bit of mathematics.

We all know what are polynomials. A polynomial of degree $n-1$ looks like,

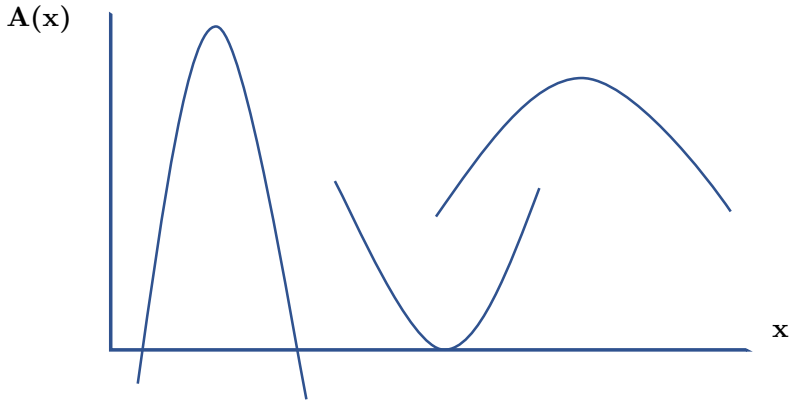
$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

Our task is – to efficiently perform three basic operations on such polynomials; evaluation of polynomial, addition of two polynomials and multiplication of two polynomials.

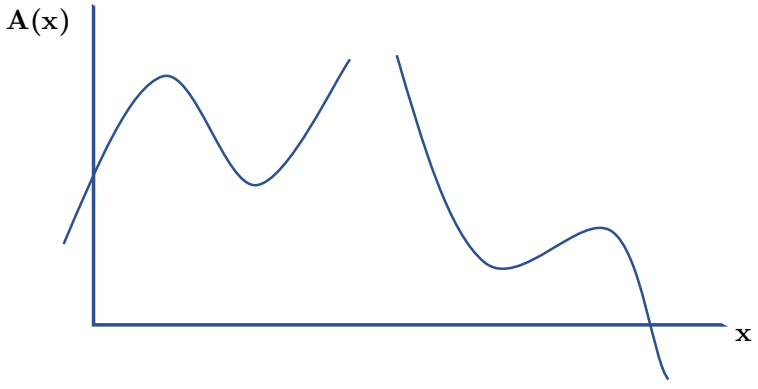
Q. Why are we so interested in polynomials?

Well, polynomials find applications almost everywhere. One application of polynomials is that – they can be used to represent signals. How?

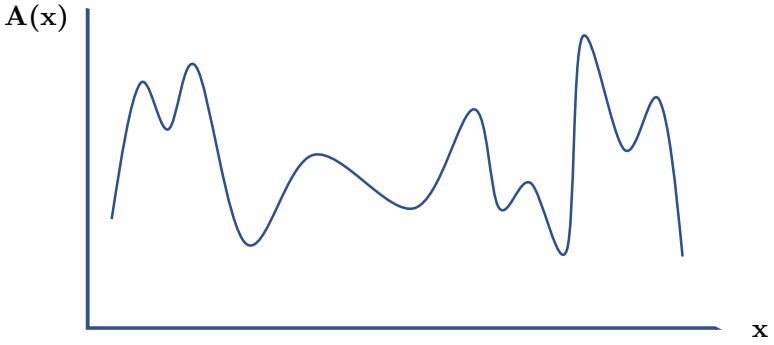
If you notice, a polynomial of degree 2, i.e. a quadratic expression in x, when plotted gives exactly one bump.



All curves drawn above can be represented by some quadratic expression of x. Similarly, all curves with two bumps can be represented as cubic expression of x.



So, if we plot a time on x-axis and the value of signal on y-axis, then if the signal has n-bumps, it could be represented in time domain as a polynomial of degree $n+1$.



So, the above random signal has 13 bumps. So, it's a polynomial of degree 14. Probably, representing a random signal as a polynomial, especially the one containing noise, would be a very bad representation, but at least we have some start to represent a signal. We will improve our idea of representation of signals gradually.

So, for a while, we will think of polynomial $A(x)$ as some signal varying with x. Now usually there are 3 ways of representing a polynomial.

1. Representation using Coefficients

   We can represent a polynomial as –

   $$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

   $$= < a_0,\ a_1,\ a_2,\ \ldots,\ a_{n-1} >$$

   Since, the polynomial of degree $n-1$ is uniquely defined by set of n coefficients, we maintain a coefficient vector of size n to represent a polynomial.

2. Representation using Roots

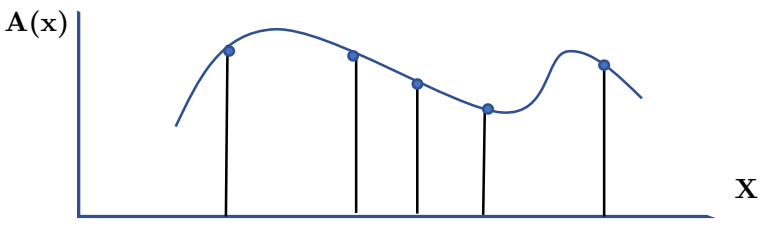
   A polynomial can also be represented by a constant c and $n-1$ roots, as follows:

   $$A(x) = c(x - r_0)(x - r_1) \ldots (x - r_{n-2})$$

   These roots may be real or may be complex.

3. Representation using Samples

   If we have a polynomial $A(x)$ of degree $n-1$, then for some interval of x, if we have n samples (approximately uniformly spaced in x) of that polynomial curve, we can uniquely represent that polynomial because there will be only 1 curve passing through all the points. For example, 2 points are enough to define a line, 3 points are enough to define a parabolic curve, etc.



In above curve, there are 3 bumps; so, the degree of the polynomial curve will be 4. Hence, we need 5 samples of this curve, to accurately represent the polynomial. So, what we need is set of points $< (x_0, y_0),\ (x_1, y_1),\ \ldots, (x_{n-1}, y_{n-1}) >$ where each point lies on the curve of polynomial. There is no harm if we have more than n samples. But we need at least n samples. This argument again reduces to the example, that we can't uniquely identify parabolic curve with less than 3 samples.

Now the question is, which representation of polynomials is the best? Well, we will explore the answer to this question in context of 3 operations which we wished to perform on polynomials; evaluation, addition and multiplication.

1. Evaluation of Polynomials

Evaluation of a polynomial means, given a value of $x = x_0$, compute the value of $A(x)$ at x_0. Let's see how much time it takes to evaluate a polynomial under different representations.

In coefficient representation, we can easily evaluate the polynomial in $O(n)$ time by writing the polynomial in *Horner's Form:*

$$A(x) = a_0 + x(a_1 + x(a_2 + \cdots x(a_{n-1}) \ldots))$$

So, start with a_{n-1}, multiply it with x_0. Add a_{n-2} to the result and multiply the result with x_0. Then take a_{n-3} and so on till we reach a_0. Eventually, we will be doing 1 multiplication and 1 addition n times, giving us the final answer in $O(n)$.

In roots representation, all we have to do is – replace x by x_0 and perform n multiplications. This again gives us answer in $O(n)$ time.

In samples representation, evaluations are difficult. If we want to find the value of $A(x)$ between two samples, we will need sophisticated interpolation techniques. The most efficient interpolation technique takes $O(n^2)$ time. There could be a whole separate domain of Numerical Analysis focusing just on interpolations; so, we won't go into details of it. But you can intuitively understand, evaluation using samples is difficult.

2. Addition of Polynomials

Given two polynomials, $A(x)$ and $B(x)$, we want to compute $C(x) = A(x) + B(x)$.

In coefficients representation, all we have to do is – add the coefficients corresponding to same powers of x. This will take at max n additions; so, we can add two polynomials in this representation in $O(n)$ time.

In roots representation, it is near impossible to add two polynomials. This is because – to add two polynomials, we will first have to convert the polynomials into coefficient representation, add them and convert them back into roots representation. In worst case, even if we could convert the polynomials from roots

form to coefficients form, we won't be able to convert it back to roots form after addition. Because it is not possible to find the roots of polynomial with degree more than 4 using some formula. We have to shift to numerical methods; and even then, we will get approximate results. So, we say – addition of two polynomials in roots form takes infinite time. Due to this reason, working with roots form in computations is near impossible.

To add two polynomials in samples representation, we assume that both polynomials are sampled at same values of x. If that's the case, we just have to add the corresponding samples. So, addition of two polynomials in samples form take $O(n)$ time.

3. Multiplication of Polynomials

Given two polynomials, $A(x)$ and $B(x)$, we want to compute $C(x) = A(x) \times B(x)$.

In coefficients form, we can express multiplication as –

$$(a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1})(b_0 + b_1 + b_2x^2 + \cdots + b_{n-1}x^{n-1}) =$$
$$a_0b_0 + (a_1b_0 + a_0b_1)x + (a_2b_0 + a_1b_1 + a_0b_2)x^2 + \cdots + a_{n-1}b_{n-1}x^{2n-2}$$

So, coefficient C_k in $C(x)$ is –

$$C_k = \sum_{j=0}^{k} A_j B_{k-j}$$

The value of k can go upto $2n - 2$. So, the overall time required to find all C_k's is $O(n^2)$, as each C_k takes $O(n)$ time.

In roots form, multiplication is very easy. If there are p roots of A(x) and q roots of B(x), then C(x) will have $p + q$ roots. We just need to copy roots of A(x) and B(x), and put them in a roots vector of $C(x)$. We need to multiply corresponding constant terms as well. This takes $O(n)$ time.

In samples form, multiplication is easy, assuming we have sampled both the polynomials at the same values of x. We just have to multiply corresponding samples of two polynomials. You can realize this fact by the following example:

Suppose we want to multiply two polynomials, $A(x) = x$ and $B(x) = x$. We have sampled both polynomials at $x_0 = 2$. Then, $A(x_0) = 2$ and $B(x_0) = 2$. Now we want to multiply both polynomials. $C(x) = A(x) \times B(x) = x^2$. If we sample the multiplied result at $x_0 = 2$, the value $C(x_0) = 4$ is same as the multiplication of corresponding samples of $A(x_0)$ and $B(x_0)$. So, in samples form,

$$C_k = A_k \times B_k \ \forall\, k$$

So, multiplying polynomials in samples form takes $O(n)$ time.

Here is the summary of time complexities of various operations in various forms. Bolded time complexities represent that the operations are inefficient.

	Coefficient vector	*Roots vector*	*Samples vector*
1. Evaluation	$O(n)$	$O(n)$	$\boldsymbol{O(n^2)}$
2. Addition	$O(n)$	$\boldsymbol{\infty}$	$O(n)$
3. Multiplication	$\boldsymbol{O(n^2)}$	$O(n)$	$O(n)$

So, which representation is the best one? None! All are good at something and bad at something. Which one is the worst? If we have to select one of them, probably it will be roots form, because we can't interconvert between root form and other forms very well. So, we will be focusing our discussion on coefficients form and the samples form.

Efficient Multiplication of Polynomials:

What we want is – somehow, if we could reduce the time complexity of multiplication in coefficient representation, probably it will be the best representation. However, we know that we can't do better than $O(n^2)$. So what we shall try is – find some conversion method from coefficient form to samples form and vice-versa which is faster than $O(n^2)$. So next time, when we have to multiply polynomials and we have coefficient vector, we will convert it to samples, multiply there efficiently in $O(n)$ and then covert resultant samples back to coefficient vector.

But why taking so much of effort to enhance computation of polynomial multiplication? Why do we care about it at all?

Because it's very similar to other operation called 'convolution' which is used in almost every signal processing technique. If we are given two vectors, we compute their convolution by reversing one vector and then computing the dot product of two vectors for all possible shifts of the reversed vector. So, if $C[x]$ represents the convoluted vector of $A[x]$ and $B[x]$, then –

$$C_k = \sum_{j=0}^{k} A_j B_{k+j}$$

for all possible values of k. Here j is the shift of the reversed vector.

If you notice, this expression is somewhat similar to polynomial multiplication in coefficient form. The only difference is – in coefficient multiplication, we don't reverse the coefficient vector. We compute the dot product for all possible shifts of one vector, without reversing it. This is the reason, two expressions differ in the sign of j. So, if we can solve polynomial multiplication in efficient way, we can solve convolution with same efficiency.

So, let's start over journey of how to find the conversion algorithm which takes from the coefficient land to samples land and vice-versa.

Inter-conversion of Coefficients and Samples forms:

Suppose we represent a polynomial as - $y = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$.

Let's just find the samples of this polynomial curve at different values of x. We know, we need n samples. So, the values at which we need to evaluate/sample y be $< x_0,\ x_1,\ x_2, \ldots,\ x_{n-1} >$. So, the sample values are:

$$y_0 = a_0 + a_1x_0 + a_2x_0^2 + \cdots + a_{n-1}x_0^{n-1}$$

$$y_1 = a_0 + a_1x_1 + a_2x_1^2 + \cdots + a_{n-1}x_1^{n-1}$$

$$\ldots$$

$$y_{n-1} = a_0 + a_1x_{n-1} + a_2x_{n-1}^2 + \cdots + a_{n-1}x_{n-1}^{n-1}$$

At the end, $< (x_0, y_0), (x_1, y_1) \ldots (x_{n-1}, y_{n-1}) >$ will represent the vector of sample points i.e. samples.

Writing above set of equations in matrix form, we get

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \ldots & x_1^{n-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \ldots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \ldots \\ y_{n-1} \end{bmatrix}$$

$$VA = Y_a$$

where matrix V is called **Vandermonde Matrix.**

Computing Vandermonde matrix will take $O(n^2)$ time, as each row will take $O(n)$ amount of computation. Once, we have Vandermonde matrix, we can calculate the samples for both the polynomials by multiplying it with coefficient vectors in $O(n^2)$ time.

Let's say we got the resultant sample vector C by multiplying corresponding samples of A and B. And now we want to convert samples to coefficients. We can do this as,

$$C = V^{-1}Y_c$$

We can find V^{-1} using Gaussian elimination. Gaussian elimination takes $O(n^3)$ operations to find the inverse of an $n \times n$ matrix. However, Vandermonde matrix is little bit special than ordinary matrices. Here, each column is one power more than the previous column. Mathematicians have tried to use this property of Vandermonde matrix and have found an efficient formula which can compute inverse of Vandermonde matrix in $O(n^2)$ work. You can have a look at the nasty formula of inverse here: https://proofwiki.org/wiki/Inverse_of_Vandermonde_Matrix

But even $O(n^2)$ is of no use to us; as we can't spend $O(n^2)$ time converting from one form to another, if we can multiply polynomials easily in $O(n^2)$ in coefficients form only. We need a better conversion algorithm, something of order $O(n \log_2 n)$.

Here is our goal in the problem:

Compute A(x) efficiently for all values of $x \in X$, where A(x) is a polynomial of x and X is a vector of positions of x for which we would like to evaluate the polynomial.

Divide and Conquer approach:

Let's try our first attempt to make this conversion algorithm efficient.

There are two ways to divide a vector. One way is to divide at the middle as we have seen in case of binary search, merge sort, etc. Another way is to divide the array into two arrays – one containing the entries at even positions and one containing entries at odd positions. We will choose the second method.

So, if $A(x) = < a_0,\ a_1,\ a_2,\ \dots,\ a_{n-1} >$, then

$$A_{even}(x) = < a_0,\ a_2,\ a_4, \dots,\ a_{n-2} >$$

$$A_{odd}(x) = < a_1,\ a_3,\ a_5, \dots,\ a_{n-1} >$$

One thing you should realize is - A_{even} and A_{odd} only contain coefficients at even and odd positions. In no way, we should assume that A_{even} or A_{odd} contain only even or odd powers of x. They are polynomials of half the degree.

$$A_{even}(x) = a_0 + a_2x^1 + a_4x^2 + \cdots = \sum_{k=0}^{\frac{n}{2}-1} a_{2k}x^k$$

$$A_{odd}(x) = a_1 + a_3x^1 + a_5x^2 + \cdots = \sum_{k=0}^{\frac{n}{2}-1} a_{2k+1}x^k$$

Using a little bit of Algebra, we can write –

$$A(x) = A_{even}(x^2) + x.A_{odd}(x^2)$$

You can substitute the values to check why above expression is true. So, if you notice, we have divided a problem of evaluating a polynomial of degree $n-1$ at x, to two smaller subproblems of evaluating polynomials of degree half the previous one at x^2. We can recursively, divide the evaluation of these two sub-problem polynomials into further smaller problems till we have only 1 term in the polynomial expression i.e. till degree of the polynomial to be evaluated reduces to 1. So, at next step, A_{even} and A_{odd} will be further divided into their even and odd parts and they will be evaluated at x^4. Now let's see how efficient this approach is.

The above algorithm works in the following way:

Suppose we need to evaluate $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3$ for $X = \{1, 2, 3, 4\}$, then,

a) We first divide $A(x)$ into A_{even} and A_{odd} as follows:

$$A_{even,1}(x) = a_0 + a_2x \text{ and } A_{odd,1}(x) = a_1 + a_3x$$

Here '1' represents that it's the first time we have broken the problem. We have to evaluate $A_{even,1}$ and $A_{odd,1}$ on X^2 i.e. $\{1, 4, 9, 16\}$.

b) We now have to further divide $A_{even,1}$ and $A_{odd,1}$ till we have only term left.

$$A_{even-even,2}(x) = a_0 \text{ and } A_{odd-ev\ \ ,2}(x) = a_2$$
$$A_{even-odd,2}(x) = a_1 \text{ and } A_{odd-odd,2}(x) = a_3$$

If there were any terms in x, we would have divided these expressions even further and all polynomials at this stage would have been evaluated at X^4 i.e. $\{1, 16, 81, 256\}$. However, since we have reached constant terms, we stop here.

c) It's now the time to combine our sub-problems. Let's say we want to find sample at x = 3. Combining above constant terms, would give us $A_{even,1}$ and $A_{odd,1}$ back.

$$\begin{aligned} A(3) &= A_{even,1}(9) + 3 * A_{odd,1}(9) \\ &= (a_0 + 9a_2) + 3 * (a_1 + 9a_3) \\ &= a_0 + 3a_1 + 9a_2 + 27a_3 \end{aligned}$$

If you put the value of x = 3 in $A(x)$, you will get the same result.

I know what are you thinking at this moment – All that for a drop of … Wait! All that for just finding a single sample. I know, it's hardly making any sense of why we are doing all this. But hopefully, we will end up with some marvellous results.

For now, let's analyse, how much work we are doing. We can write the recurrence relation as:

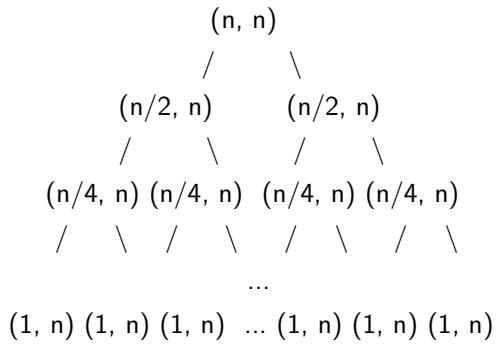
$$T(n, |X|) = 2.T\left(\frac{n}{2}, |X|\right) + O(n + |X|)$$

where n is the degree of polynomial, |X| is the size of the vector X i.e. the number of values of x at which we need to find samples. We saw that – the task of finding samples at each value of vector X on a polynomial of degree n, reduces to two subproblem polynomials. We now have to evaluate 2 polynomials of degree n/2, but on the vector X^2 that has same size as that of X. So |X| doesn't change. This gives us $2.T\left(\frac{n}{2}, |X|\right)$ term. After solving polynomials, we need to combine them. In combine step, we do n additions and |X| products, giving us $O(n + |X|)$ term.

But wait a second! Isn't size of vector X same as degree n?

Yes, *in this case, yes!* But what if, you want more samples than required. We already said, if we have more samples of a polynomial, there is no harm in it. But in this case, we can consider |X| = n.

Now, finding the total amount of work which we are doing is not clear from above recurrence relation. To get the idea of the recurrence relation, we draw the recurrence tree. Task at each lower level gets broken down into two subtasks.



On first layer, the single node represents $O(1 \times n^2)$ work i.e. $O(n^2)$ work, which we already know. We broke this work into smaller problems till we reached the last layer. In the last layer, there will be $2^{\log_2 n}$ leaf nodes and each leaf node represents $O(n)$ work. So, the total work is $O\left(2^{\log_2 n} \times n\right) = O(n \times n) = O(n^2)$.

So, despite all this effort of Divide and Conquer, we couldn't achieve better than $O(n^2)$. Frustrating! Well, we have to look where is the problem?

The problem is in the size of X. If somehow, with the degree, even the size of the vector X is reduced to half at each step, then the last level will contain $2^{\log_2 n}$ leaves and each leaf will be (1, 1) denoting $O(1)$ work. Therefore, then the work at the last level will be $O\left(2^{\log_2 n} \times 1\right) = O(n)$. And since there are total $\log_2 n$ levels, total work would be $O(n \log_2 n)$.

So somehow, we have to reduce the work by reducing the size of vector X. But how? Let's think backwards.

At last level, we want X to contain only 1 value. Let's say $X = \{1\}$ in last level. At each next step, the number of elements in X gets reduced to half and we actually evaluate each next step on square of the vector X of previous level. So, at second last step, we need two elements in X. If $X = \{-1, 1\}$, then last layer would work on $X^2 = \{1\}$. *Ah! So, what we need are square roots as every value has two square roots. And squaring each value would lead only half the distinct squares! So, what we need in vector X are actually square roots.*

Continuing with same argument, at third last level, we require X to contain $\{i, -i, -1, 1\}$. What we understand is – if we allowed only real values of x in vector X, then size of X will remain same on squaring it. But if we allowed complex numbers, especially the roots of unity, to be in X, then on squaring, the size of vector X reduces to half and suddenly the same divide and conquer algorithm turns to $O(n \log n)$ from $O(n^2)$.

So, we are now left with only one task – deciding, what should be the initial vector X?

If you notice, at each next step, the size of vector X reduces to half. So, the first level must have a vector X, whose size is of the form 2^h, where h is height of the recurrence tree, so that, after every level when size of X gets divided by 2, we eventually get only 1 element in X at the last level. In other words, X should always contain 2^k elements for some k, at every level.

This means, if the degree of your polynomial is 13, you can't have 14 values in X for samples. You will need 16 values. Again, remember what we said, having more samples of a signal, is infact better. If the degree of polynomial is 47, then the number of x values required in X for samples will be 64. So, if we require, 64 values in X initially, what we need is 64 roots of unity. So that, after squaring vector X at each level, we end up having only $\{1\}$ in the last level.

BTW, did you notice something strange? X contains the values of variable x plotted on x-axis, at which we need to sample the polynomial. And suddenly, for the sake of reducing the complexity of the algorithm, we are allowing X to contain complex numbers. This means we are actually sampling polynomials at complex values! Here is the hint: This might not make any sense in the domain of x, i.e. the time domain, if we imagine the polynomial to be signal and variable x to be time. You can try thinking it in frequency domain! XD.

Well, we did a lot of talking. Let's now finally, summarize what we studied and do some algebra related to that to derive some mathematical results.

From now onwards, I will consider that n is actually the number of samples I need. So n is the size of vector X. The degree of polynomial is assumed to be less than or equal to $n-1$. This is because, now our main focus will be on the vector X, instead of the polynomial. We also know that this n now has to be some power of 2 and X contains values which are nothing but n^{th} roots of unity. Let's start some mathematics by answering - How do we actually calculate roots of unity?

This is how the pattern works: The square root of 1 is -1 and 1. On complex plane, these two points divide the unit circle into two semicircles. The 4 roots of unity are $\{i, -i, -1, 1\}$. These points divide unit circle into 4 equal parts. 1 and -1 lie on real axis and i and $-i$ lie on imaginary axis. 8 roots of unity will be – 4 above roots plus 4 other roots which will be at angles $\pm 45°$ and $\pm 135°$. So, these 8 roots will divide unit circle into eight equal parts.

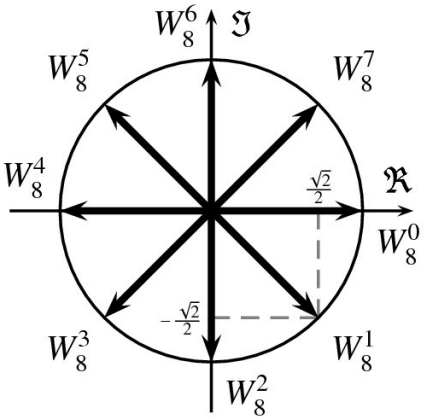
So, for n^{th} roots of unity, we can write, $W^n = 1$. Above equation should have n roots.

$$W^n = 1 = cos2\pi + i.sin2\pi = e^{i2\pi}$$

So, the n^{th} roots of unity are, $W = e^{i.\frac{2k\pi}{n}}$ where $k = 0, 1, 2, \ldots, n-1$.

The root corresponding to k = 1 is actually termed as principal $n^{th}\ root$, W_n. All other roots are powers of omega i.e. $W_n^0, W_n^1, W_n^2, \dots, W_n^{n-1}$ where $W_n = e^{\frac{i2\pi}{n}}$.

Here is the picture of 8 roots of unity from Wikipedia:



So, our vector $X = < x_0,\ x_1,\ x_2, \dots,\ x_{n-1} > = < W_n^0,\ W_n^1,\ W_n^2, \dots,\ W_n^{n-1} >$

Now, if we look back at our Vandermonde matrix equation,

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \dots \\ y_{n-1} \end{bmatrix}$$

Substituting above values, we get:

$$\begin{bmatrix} y_0 \\ y_1 \\ \dots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & W_n^0 & (W_n^0)^2 & \dots & (W_n^0)^{n-1} \\ 1 & W_n^1 & (W_n^1)^2 & \dots & (W_n^1)^{n-1} \\ \dots & \dots & \dots & \dots & \dots \\ 1 & W_n^{n-1} & (W_n^{n-1})^2 & \dots & (W_n^{n-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \dots \\ a_{n-1} \end{bmatrix}$$

If you notice, any y_k can be written as $A(x_k)$.

$$y_k = a_0 + a_1\left(W_n^k\right) + a_2\left(W_n^k\right)^2 + \cdots + a_{n-1}\left(W_n^k\right)^{n-1}$$

$$\therefore y_k = \sum_{j=0}^{n-1} a_j \left(W_n^k\right)^j$$

$$\boxed{\therefore y_k = \sum_{j=0}^{n-1} a_j e^{\frac{i2\pi}{n} . jk}}$$

The above formula is nothing but **Discrete Fourier Transform!**

So, **the Discrete Fourier Transform (DFT) of a polynomial $A(x)$ or equivalently the vector of coefficients $< a_0, a_1, a_2, \ldots, a_{n-1} >$ is defined as the values of the polynomial at the points which are n^{th} roots of unity.**

$$\begin{aligned}\mathrm{DFT}(a_0, a_1, \ldots, a_{n-1}) &= (y_0, y_1, \ldots, y_{n-1}) \\ &= (A(w_{n_0}), A(w_{n,1}), \ldots, A(w_{n,n-1})) \\ &= (A(w_n^0), A(w_n^1), \ldots, A(w_n^{n-1}))\end{aligned}$$

And the divide and conquer algorithm used to efficiently compute the DFT sequence, without using Vandermonde matrix, is called FFT (Fast Fourier Transform).

Again! Did you notice something strange? We know that, discrete time signal is a signal which is sampled at regular time intervals and DFT of a discrete time signal takes us to its frequency domain representation. Now here we say, that discrete time signal is actually the coefficients of polynomial and DFT gives us the samples of that polynomials, sampled at roots of unity. Well, it's very hard to imagine what's happening here and form a link between these concepts. Imagine. If you can!

Try to keep your calm while reading the next statement: *Till now, what we have studied is – how to efficiently convert from coefficient representation to samples representation. But we still don't know how to come back from samples world to coefficients world. A small part is still pending.* 😉

Probably, you guessed it right. To come back to coefficients world, what we need is – Inverse Discrete Fourier Transform (IDFT). But I would like to take some time, and reach the result of IDFT by starting from basics – our Vandermonde matrix! Now, it is assumed that we have samples and what we want is the coefficient vector.

We have, $Y = VA$. So, we can write, $A = V^{-1}Y$. Calculating inverse of Vandermonde matrix is now a totally different scenario because the values inside it are now the n^{th} roots of unity. So, we are trying to find the inverse of following Vandermonde matrix:

$$V = \begin{bmatrix} 1 & W_n^0 & (W_n^0)^2 & (W_n^0)^3 & \ldots & (W_n^0)^{n-1} \\ 1 & W_n^1 & (W_n^1)^2 & (W_n^1)^3 & \ldots & (W_n^1)^{n-1} \\ 1 & W_n^2 & (W_n^2)^2 & (W_n^2)^3 & \ldots & (W_n^2)^{n-1} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & W_n^{n-1} & (W_n^{n-1})^2 & (W_n^{n-1})^3 & \ldots & (W_n^{n-1})^{n-1} \end{bmatrix}$$

$$\therefore V = \begin{bmatrix} W_n^0 & W_n^0 & W_n^0 & W_n^0 & \ldots & W_n^0 \\ W_n^0 & W_n^1 & W_n^2 & W_n^3 & \ldots & W_n^{n-1} \\ W_n^0 & W_n^2 & W_n^4 & W_n^6 & \ldots & W_n^{2(n-1)} \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ W_n^0 & W_n^{n-1} & W_n^{2(n-1)} & W_n^{3(n-1)} & \ldots & W_n^{(n-1)(n-1)} \end{bmatrix}$$

Note: Notice that V is a symmetric matrix.

The inverse of above Vandermonde matrix is given by –

$$V^{-1} = \frac{\bar{V}}{n}$$

where $\bar{V}$ is the complex conjugate of V. The conjugate of a complex number $e^{i\theta}$ is $e^{-i\theta}$. So, according to above result,

$$V^{-1} = \frac{1}{n}\begin{bmatrix} W_n^0 & W_n^0 & W_n^0 & W_n^0 & \dots & W_n^0 \\ W_n^0 & W_n^{-1} & W_n^{-2} & W_n^{-3} & \dots & W_n^{-(n-1)} \\ W_n^0 & W_n^{-2} & W_n^{-4} & W_n^{-6} & \dots & W_n^{-2(n-1)} \\ \dots & \dots & \dots & \dots & \dots & \dots \\ W_n^0 & W_n^{-(n-1)} & W_n^{-2(n-1)} & W_n^{-3(n-1)} & \dots & W_n^{-(n-1)(n-1)} \end{bmatrix}$$

Proof: To prove above result, all we need to prove is: $V.\bar{V} = n.I$

Let $P = V.\bar{V}$. Then,

$$P_{jk} = (row\ j\ of\ V).(column\ k\ of\ \bar{V})$$

$$= \sum_{m=0}^{n-1} e^{i.\frac{2\pi}{n}.jm}.e^{i.\frac{2\pi}{n}.km}$$

$$= \sum_{m=0}^{n-1} e^{i.\frac{2\pi}{n}.(j-k)m}$$

If $j = k$, i.e. diagonal elements,

$$P_{jk} = \sum_{m=0}^{n-1} 1 = n$$

If $j \neq k$,

$$P_{jk} = \sum_{m=0}^{n-1} \left(e^{i.\frac{2\pi}{n}(j-k)}\right)^m$$

This is sum of n terms of geometric series.

$$P_{jk} = \frac{\left(e^{i.\frac{2\pi}{n}(j-k)}\right)^n - 1}{e^{i.\frac{2\pi}{n}(j-k)} - 1}$$

Now, $n's$ get cancelled. $(j-k)$ is an integer. $e^{i.2\pi.integer} = 1$.

$$\therefore P_{jk} = 0$$

Hence proved,

$$V^{-1} = \frac{\bar{V}}{n}$$

Finally, we can write:

$$A = \frac{\bar{V}}{n} Y$$

Writing the matrix equations, we can show that,

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j W_n^{-k}$$

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j e^{-i.\frac{2\pi}{n}.kj}$$

The above formula is nothing but – **Inverse Discrete Fourier Transform.**

The problem of computing IDFT is solved by same FFT algorithm only instead of W_k^n we have to use W_k^{-n}, and at the end we need to divide the resulting coefficients by n. Thus, the computation of IDFT also takes $O(n \log n)$ time.

C++ Implementation:

Here we present a simple recursive implementation of the FFT and the inverse FFT, both in one function, since the difference between the forward and the inverse FFT are so minimal. To store the complex numbers, we use the complex type in the C++ STL.

The function gets passed a vector of coefficients, and the function will compute the DFT or inverse DFT and store the result again in this vector. The argument invert shows whether the direct or the inverse DFT should be computed. Inside the function we first check if the length of the vector is equal to one, if this is the case then we don't have to do anything. Otherwise we divide the vector a into two vectors $a0$ and $a1$ and compute the DFT for both recursively. Then we initialize the value wn and a variable w, which will contain the current power of wn. Then the values of the resulting DFT are computed using the above formulas.

If the flag invert is set, then we replace wn with wn^{-1}, and each of the values of the result is divided by 2 (since this will be done in each level of the recursion, this will end up dividing the final values by n).

Using this function, we can create a function for multiplying two polynomials. This function works with polynomials with integer coefficients; however, you can also adjust it to work with other types.

```cpp
using cd = complex<double>;
const double PI = acos(-1);

void fft(vector<cd> &a, bool invert) {
    int n = a.size();
    if (n == 1)
        return;
    vector<cd> a0(n / 2), a1(n / 2);
    for (int i = 0; 2 * i < n; i++) {
        a0[i] = a[2*i];
        a1[i] = a[2*i+1];
    }
    fft(a0, invert);
    fft(a1, invert);

    double ang = 2 * PI / n * (invert ? -1 : 1);
    cd w(1), wn(cos(ang), sin(ang));
    for (int i = 0; 2 * i < n; i++) {
        a[i] = a0[i] + w * a1[i];
        a[i + n/2] = a0[i] - w * a1[i];
        if (invert) {
            a[i] /= 2;
            a[i + n/2] /= 2;
        }
        w *= wn;
    }
}

vector<int> multiply(vector<int> const& a, vector<int> const& b) {
    vector<cd> fa(a.begin(), a.end()), fb(b.begin(), b.end());
    int n = 1;
    while (n < a.size() + b.size())
        n <<= 1;
    fa.resize(n);
    fb.resize(n);
    fft(fa, false);
    fft(fb, false);
    for (int i = 0; i < n; i++)
        fa[i] *= fb[i];
    fft(fa, true);

    vector<int> result(n);
    for (int i = 0; i < n; i++)
        result[i] = round(fa[i].real());
    return result;
}
```

```
int main() {
    vector<int> a = {1, 1, 1, 1};
    vector<int> b = {2, 1, 2};
    vector<int> c = multiply(a, b);

    for (int i = 0; i < c.size(); i++)
        cout << c[i] << " ";
    cout << endl;

    return 0;
}
```

There are many programming problems which could be solved using FFT, especially some problems like string matching (will be discussed in strings section). Here is one problem, which could simply be solved using FFT.

All possible sums

We are given two arrays a[ ] and b[ ]. We have to find all possible sums a[i] + b[j], and for each sum count how often it appears.

For example, for a = [1, 2, 3] and b = [2, 4] we get: then sum 3 can be obtained in 1 way, the sum 4 also in 1 way, 5 in 2, 6 in 1, 7 in 1.

We construct for the arrays a and b, two polynomials A and B. The numbers of the array will act as the exponents in the polynomial $a[i] \to x^{a[i]}$ and the coefficients of this term will by how often the number appears in the array.

Then, by multiplying these two polynomials in $O(n \log n)$ time, we get a polynomial C, where the exponents will tell us which sums can be obtained, and the coefficients tell us how often. To demonstrate this on the example:

$$(1x^1 + 1x^2 + 1x^3)(1x^2 + 1x^4) = 1x^3 + 1x^4 + 2x^5 + 1x^6 + 1x^7$$

Chapter 4 – Greedy Technique

4.1 Introduction

A greedy algorithm constructs a solution to the problem by always making a choice that looks the best at the moment. A greedy algorithm never takes back its choices, but directly constructs the final solution. For this reason, greedy algorithms are usually very efficient.

The difficulty in designing greedy algorithms is to find a greedy strategy that always produces an optimal solution to the problem. The locally optimal choices in a greedy algorithm should also be globally optimal. It is often difficult to argue that a greedy algorithm works.

As an example, we consider a problem where we are given a set of coins and our task is to form a sum of money n using the coins. The values of the coins are coins = {c1, c2, ..., ck}, and each coin can be used as many times we want. What is the minimum number of coins needed?

For example, if the coins are the euro coins (in cents) {1, 2, 5, 10, 20, 50, 100, 200} and n = 520, we need at least four coins. The optimal solution is to select coins 200 + 200 + 100 + 20 whose sum is 520.

A simple greedy algorithm to the problem always selects the largest possible coin, until the required sum of money has been constructed. This algorithm works in the example case, because we first select two 200 cent coins, then one 100 cent coin and finally one 20 cent coin. But does this algorithm always work? It turns out that if the coins are the euro coins, the greedy algorithm always works, i.e., it always produces a solution with the fewest possible number of coins.

The correctness of the algorithm can be shown as follows:

First, each coin 1, 5, 10, 50 and 100 appears at most once in an optimal solution, because if the solution would contain two such coins, we could replace them by one coin and obtain a better solution. For example, if the solution would contain coins 5 + 5, we could replace them by coin 10. In the same way, coins 2 and 20 appear at most twice in an optimal solution, because we could replace coins 2 + 2 + 2 by coins 5 + 1 and coins 20 + 20 + 20 by coins 50 + 10. Moreover, an optimal solution cannot contain coins 2 + 2 + 1 or 20 + 20 + 10, because we could replace them by coins 5 and 50. Using these observations, we can show for each coin x that it is not possible to optimally construct a sum x or any larger sum by only using coins that are smaller than x. For example, if x = 100, the largest optimal sum using the smaller coins is 50 + 20 + 20 + 5 + 2 + 2 = 99. Thus, the greedy algorithm that always selects the largest coin produces the optimal solution.

This example shows that it can be difficult to argue that a greedy algorithm works, even if the algorithm itself is simple.

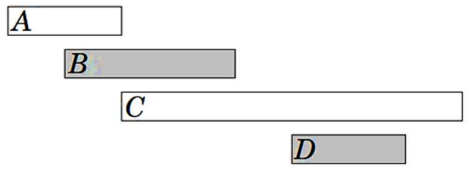
In the general case, the coin set can contain any coins and the greedy algorithm does not necessarily produce an optimal solution. We can prove that a greedy algorithm does not work by showing a counterexample where the algorithm gives a wrong answer. In this problem we can easily find a counterexample: if the coins are $\{1, 3, 4\}$ and the target sum is 6, the greedy algorithm produces the solution $4 + 1 + 1$ while the optimal solution is $3 + 3$. It is not known if the general coin problem can be solved using any greedy algorithm. However, as we will see in one of the upcoming chapters, in some cases, the general problem can be efficiently solved using a dynamic programming algorithm that always gives the correct answer.

4.2 Scheduling Problem

Many scheduling problems can be solved using greedy algorithms. A classic problem is as follows: Given n events with their starting and ending times, find a schedule that includes as many events as possible. It is not possible to select an event partially. For example, consider the following events:

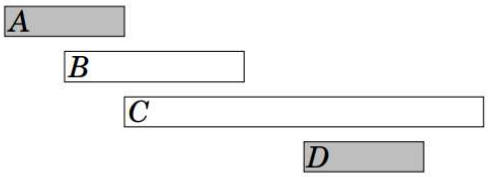
Event	*Starting Time*	*Ending Time*
A	1	3
B	2	5
C	3	9
D	6	8

In this case the maximum number of events is two. For example, we can select events B and D as follows:

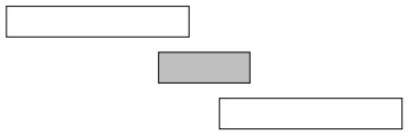


It is possible to invent several greedy algorithms for the problem, but which of them works in every case?

The first idea is to select as short events as possible. In the example case this algorithm selects the following events:

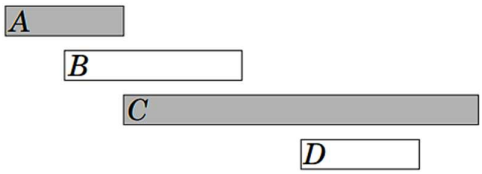


However, selecting short events is not always a correct strategy. For example, the algorithm fails in the following case:

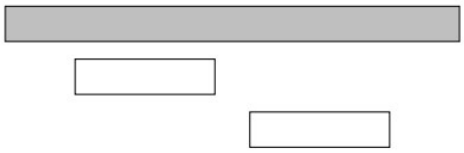


If we select the short event, we can only select one event. However, it would be possible to select both long events.

Another idea is to always select the next possible event that begins as early as possible. This algorithm selects the following events:

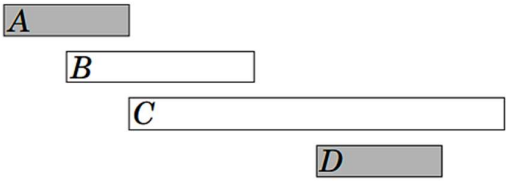


However, we can find a counterexample also for this algorithm. For example, in the following case, the algorithm only selects one event:



If we select the first event, it is not possible to select any other events. However, it would be possible to select the other two events.

The third idea is to always select the next possible event that ends as early as possible. This algorithm selects the following events:



It turns out that this algorithm always produces an optimal solution. The reason for this is that it is always an optimal choice to first select an event that ends as early as possible. After this, it is an optimal choice to select the next event using the same strategy, etc., until we cannot select any more events. One way to argue that the algorithm works is to consider what happens if we first select an event that ends later than the event that ends as early as possible. Now, we will have at most an equal number of choices how we can select the next event. Hence, selecting an event that ends later can never yield a better solution, and the greedy algorithm is correct.

```
struct Activitiy {
    int start, finish;
};

bool compare(Activitiy s1, Activitiy s2) {
    return (s1.finish < s2.finish);
}

void printMaxActivities(Activitiy arr[], int n) {
    // Sort jobs according to finish time
    sort(arr, arr+n, compare);
    cout << "Following activities are selected:\n";

    // The first activity always gets selected
    int i = 0;
    cout << "(" << arr[i].start << ", " << arr[i].finish << "), ";

    // Consider rest of the activities
    for (int j = 1; j < n; j++) {
        // If this activity has start time greater than or equal to the
        // finish time of previously selected activity, then select it.
        if (arr[j].start >= arr[i].finish) {
          cout << "(" << arr[j].start << ", " << arr[j].finish << "), ";
          i = j;
        }
    }
}

int main() {
    Activitiy arr[] = {{5, 9}, {1, 2}, {3, 4}, {0, 6}, {5, 7}, {8, 9}};
    int n = sizeof(arr)/sizeof(arr[0]);
    printMaxActivities(arr, n);
    return 0;
}
```

4.3 Sequencing Problem

Problems

Q. 1 Given arrival and departure times of all trains that reach a railway station, the task is to find the minimum number of platforms required for the railway station so that no train waits. We are given two arrays which represent arrival and departure times of trains that stop.

Solution: The idea is to consider all events in sorted order. Once the events are in sorted order, trace the number of trains at any time keeping track of trains that have arrived, but not departed.

1. Sort the arrival and departure time of trains.
2. Create two pointers i=0, and j=0 and a variable to store ans and current count plat
3. Run a loop while i<n and j<n and compare the ith element of arrival array and jth element of departure array.
4. If the arrival time is less than or equal to departure then one more platform is needed so increase the count, i.e. plat++ and increment i
5. Else if the arrival time greater than departure then one less platform is needed so decrease the count, i.e. plat++ and increment j
6. Update the ans, i.e ans = max (ans, plat).

```
int minPlatforms(int arr[], int dep[], int n) {
    sort(arr, arr + n);
    sort(dep, dep + n);
    int plat_needed = 1, result = 1, i = 1, j = 0;
    while (i < n && j < n) {
        if (arr[i] <= dep[j]) {
            plat_needed++; i++;
        }
        else if (arr[i] > dep[j]) {
            plat_needed--; j++;
        }
        if (plat_needed > result)
            result = plat_needed;
    }
    return result;
}

int main() {
    int arr[] = { 900, 940, 950, 1100, 1500, 1800 };
    int dep[] = { 910, 1200, 1120, 1130, 1900, 2000 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Platforms Required = " << minPlatforms(arr, dep, n);
    return 0;
}
```

Chapter 5 – Backtracking

Chapter 6 – Dynamic Programming

6.1 Introduction

6.2 Common DP Problems