Kalapi
Page No.
Date

Lecture 3 Algorithms :-

Algorithm — Sorting : sort(arr, arr+n) // sorts [startIndex, endIndex)
sort (v.begin(), v.end()) // sorts [startIterator, end it)
O(n logn) time complexity.

Algorithm — Reverse :- reverse(arr, arr+n);
reverse (v.begin(), v.end());
$\Theta(n)$ time complexity.

Algorithm — Maximum / Minimum
int el = *max-element(arr, arr+n);
int el = *max-element(v.begin(), v.end());

int el = *min-element(arr, arr+n);
int el = *min-element(v.begin(), v.end());

$\Theta(n)$ time complexity

Algorithm — Accumulate
initial sum
int sum = accumulate(arr, arr+n, 0);
int sum = accumulate(v.begin(), ~~veg~~ v.end(), 0);
$\Theta(n)$ time complexity

Algorithm — Count
find freq. of x
int cnt = count(arr, arr+n, x)
int cnt = count(v.begin(), v.end(), x)
$\Theta(n)$ time complexity

Algorithm — find : returns iterator to first occurance of an element.
Unsorted array.
auto it = find(arr, arr+n, x); (Points to .end() if x does not exist)
index = it-arr;

Kalapi
Page No.
Date

```
auto it = find(v.begin(), v.end(), x);
index = it-v.begin();
```

O(n) time complexity

Algorithm Binary-search O(logn); return true/false
works only on sorted array

```
bool res = binary_search(arr, arr+n, x);
bool res = binary_search(v.begin(), v.end(), x);
```

Algorithm:- lower bound O(logn); return iterator
works only on sorted array

```
auto it = lower_bound(arr, arr+n, x);
index = it - arr;
auto it = lower_bound(v.begin(), v.end(), x);
index = it - v.begin();
```

Returns: Iterator to the first element that is not lesser than x (or) first element greater than or equal to x.

If all elements are lesser than x → returns v.end().

Algorithm Upper Bound: O(logn); return iterator
works only on sorted array

```
auto it = upper_bound(arr, arr+n, x);
index = it-arr;
auto it = upper_bound(v.begin(), v.end(), x);
index = it - v.begin();
```

Returns: Iterator to the first element that is just greater than x.
If all elements are lesser or equal to x, returns v.end();

Algorithm Next Permutation: Gives permutations in lexicographical manner.

O(n) time complexity

```
string str = "abc"
bool res = next_permutation(s.begin(), s.end());
```

→ true ; if there is any next permutation possible
; arrange string in next permutation.

→ false ; if there is no next permutation possible
; Don't change the string.

Printing all permutations: for a string of length n, $\lfloor n$ permutation exist.

~~Ex~~ given string str, print all permutation.

```
=> sort(str.begin(), str.end()); O(nlogn)
   do
   {
      cout << str << "\n";
   }
   while (next_permutation(str.begin(), str.end()));
```

time complexity :- loop runs $\lfloor n$ times and each time O(n) time for next permutation.

∴ time complexity: ~~O(n * $\lfloor n$) huge~~

Θ $O(n\log n) + O(n * \lfloor n) = O(n * \lfloor n)$

Algorithm :- Previous Permutation :-

```
bool res = prev_permutation(s.begin(), s.end()).
```

Θ(n)

Kalapi
Page No:
Date

Just figure out - what all conditions are true ~~ ~~ According to our need.

Comparator sort(arr, arr+n, comp);
↳ comparator function:

```
bool comp(int ele1, int ele2)   // to sort in decreasing order.
{
 if (ele1>=ele2) return true; // do not swap
 else return false; // swap
}
```

~~vector<int> v; sort(v)~~

~~sort(arr, arr+n, greater<int>);~~ → ~~inbuilt comparator to sort in decreasing order.~~

~~vector<pair<int,int>> v;~~

// ~~sort(arr, arr+n, greater<pair<int,int>>~~

⇒ greater<int>()
⇒ greater<pair<int,int>>()
} inbuilt comparators for sorting in decreasing order

Imp:- Comparators can only be used for linear data structures eg. String, array, vector.