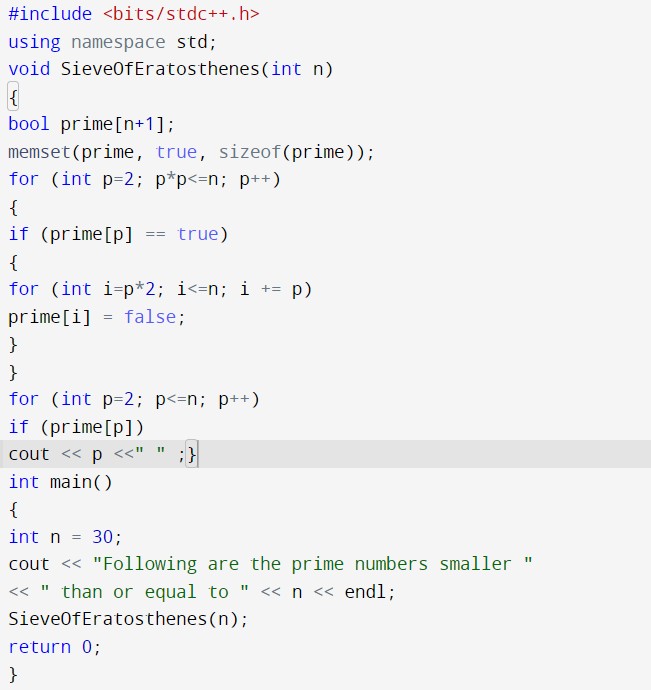
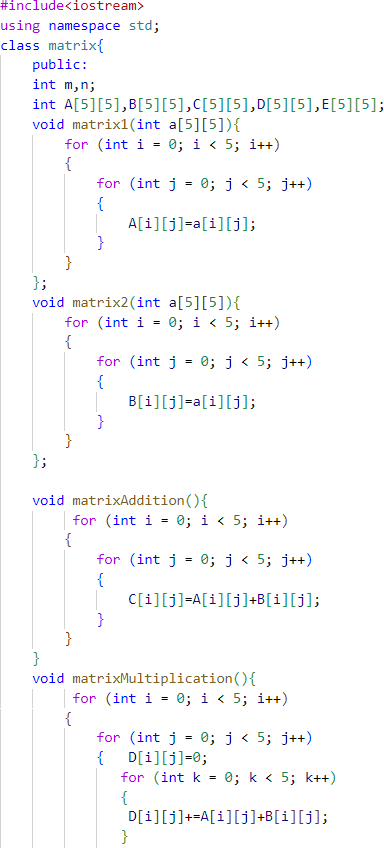
ASSIGNMENT 2

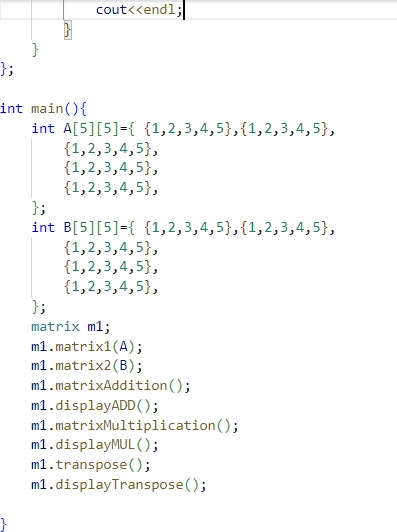
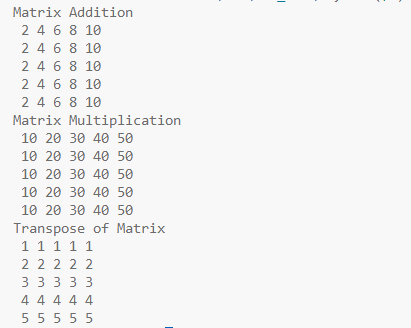
Implementation of Sieve



MATRIX OPERATIONS







ASSIGNMENT 3

|  |
| --- |
| **Prog.1:** Write a program to multiply two matrices of size n\*n and count the total no of program steps using the count method and plot the graph for how time complexity varies with the increasing size of matrices, and also do an analysis of actual time taken to perform matrix multiplication using the time() function. |
| #include<bits/stdc++.h> using namespace std;  int main(){ int n;  cin>>n; int a[n][n]; int  b[n][n];  for(int i=0;i<n;i++)for(int j=0;j<n;j++)cin>>a[i][j]; for(int i=0;i<n;i++)for(int j=0;j<n;j++)cin>>b[i][j];  int r[n][n]; time\_t start, end; time(&start);  int count=0; for(int i=0;i<n;i++){  for(int j=0;j<n;j++){  count++; r[i][j]=0;  for(int k=0;k<n;k++){  count++; r[i][j]+=a[i][k]\*b[k][j];  }  }  }  time(&end);  cout<<"Actual Time: "<<(double)(end - start)<<" ms\n"; cout<<"Count = "<<count<<"\n";  for(int i=0;i<n;i++){  for(int j=0;j<n;j++)cout<<r[i][j]<<" "; |

|  |
| --- |
| cout<<"\n";  }  return 0;  } |
|  |
| **Prog.2:** Implement Recursive Binary search and Linear search and determine the time taken to search an element |
| #include<bits/stdc++.h> using namespace std; int c=0; int binarySearch(int a[], int n, int key, int s, int e){ |

|  |
| --- |
| if(s>e){  c++; return -1;  }  int mid=(s+e)/2;c++; if(key>a[mid]){ c++; return binarySearch(a, n, key, mid+1, e);  }  if(key<a[mid]){ c++; return binarySearch(a, n, key, s, mid-1);  }  c++; return mid;  }  int linearSearch(int a[], int n, int key, int idx){ if(idx<0){ c++; return -1;  }  if(a[idx]==key){  c++; return idx;  }  c++; return linearSearch(a, n, key, idx-1);  }  int linearSearch(int a[], int n, int key){ c++; return linearSearch(a, n, key, n-1);  }  int binarySearch(int a[], int n, int key){ c++; return binarySearch(a, n, key, 0, n-1);  }  int main(){ int n;  cin>>n; int a[n];  for(int i=0;i<n;i++)cin>>a[i]; int key; cin>>key; |

|  |
| --- |
| cout<<"Binary Search: Index="<<binarySearch(a, n, key)<<"\t"; cout<<"Count="<<c<<"\n";  c=0;  cout<<"Linear Search: Index="<<linearSearch(a, n, key)<<"\t"; cout<<"Count="<<c<<"\n";  return 0;  } |
|  |
| **Prog.3:** Write a program to determine if a given matrix is a sparse matrix. Calculate its time and Space complexity. How it is more efficient than the conventional matrix? |
| #include<bits/stdc++.h> using namespace std; int main(){  int n, m; cin>>n>>m; int a[n][m]; for(int i=0;i<n;i++) for(int j=0;j<m;j++) cin>>a[i][j]; int c=0; |

|  |
| --- |
| for(int i=0;i<n;i++) for(int j=0;j<m;j++) if(a[i][j]==0)c++; if(c>(n\*m/2))cout<<"Sparse Matrix\n";  else cout<<"Not Sparse Matrix\n";  return 0;  } |
|  |
| Time Complexity: O(n X m) Space Complexity: O(n X m) |
|  |

[Document title]

ASSIGNMENT 4

|  |
| --- |
| **Prog.1:** Write a program to compare Binary search and Ternary search in terms of the number of comparisons done by each. Do this experimentally by plotting a graph and theoretically by analyzing the Algorithm. |
| #include <bits/stdc++.h> using namespace std;  int binarySearch(int nums[], int low, int high, int target, int &comp){ if (low > high) return -1;  int mid = (low + high)/2; if (target == nums[mid]){  comp++; return mid;}  else if (target < nums[mid]){ comp++;  return binarySearch(nums, low, mid - 1, target, comp);} else return binarySearch(nums, mid + 1, high, target, comp);  }  int ternarySearch(int nums[], int l, int r, int x, int &comp){ if(l>r) return -1; int mid1 = l + (r-l)/3; int mid2  = r - (r-l)/3; if(nums[mid1] == x){  comp++; return mid1;  }  if(nums[mid2] == x){ comp++;  return mid2;  }  if(x<nums[mid1]){ comp++;  return ternarySearch(nums,l,mid1-1,x,comp);  }  else if(x>nums[mid2]){ comp++;  return ternarySearch(nums,mid2+1,r,x,comp);  }  else return ternarySearch(nums,mid1+1,mid2-1,x,comp);  } |



|  |
| --- |
| int main(void)  { int A[] = {0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38};  int target = 6;  int n = sizeof(A) / sizeof(A[0]);  int compB = 0, compT = 0;  int indexB = binarySearch(A, 0, n-1, target, compB); int indexT = ternarySearch(A, 0, n-1, target, compT); cout<<"Binary Search:\nFound "<<target<<" at "<<indexB; cout<<"\tComparisons: "<<compB;  cout<<"\nTernary Search:\nFound "<<target<<" at "<<indexT; cout<<"\tComparisons: "<<compT;  return 0;  } |
|  |
| * Binary Search * Teranary Search |

ASSIGNMENT 5



|  |
| --- |
| **Prog.1:** Write a program to implement Merge sort using Link array. |
| #include <bits/stdc++.h> using namespace std;  int merge(vector<int> &a, vector<int> &link, int m1, int m2){ int i=m1, j=m2, k=0;  while(i!=0 and j!=0){ if(a[i]<=a[j]){ link[k]=i; k=i; i=link[i];  }else{ link[k]=j;  k=j; j=link[j];  }  } if(i==0) link[k]=j;  else link[k]=i; return link[0];  }  int mergeSort(vector<int> &a, vector<int> &link, int s, int e){ if(s==e){  return s;  }  int mid=(s+e)>>1;  int m1=mergeSort(a, link, s, mid); int m2=mergeSort(a, link, mid+1, e); return merge(a, link, m1, m2);  }  int mergeSort(vector<int> &a, vector<int> &link, int n){ return mergeSort(a, link, 1, n); } |

|  |
| --- |
| int main(){  int n; cin>>n;  vector<int> a(n+1), link(n+1); link[0]=0;  for(int i=1;i<=n;i++){  cin>>a[i]; link[i]=0;  }  int it=mergeSort(a, link, n); while(it!=0){ cout<<a[it]<<" "; it=link[it];  } cout<<"\n"; for(int i=0;i<=n;i++)cout<<link[i]<<" ";  return 0;  } |
|  |
| **Prog.2:** Modified merge sort using insertion sort after size of array becomes 16. |
| #include<bits/stdc++.h> using namespace std;  void insertionSort(vector<int> &a, int s, int e){ |



|  |
| --- |
| for(int i=s;i<=e;i++){  int picked=a[i]; int j=i-1;  while(a[j]>picked and j>=0){ a[j+1]=a[j];  j--;  }  a[j+1]=picked;  }  }  void merge(vector<int> &a, int s, int e){  int mid=(s+e)/2; vector<int> temp; int i=s, j=mid+1; while(i<=mid and j<=e){ if(a[i]<=a[j]) temp.push\_back(a[i++]); else temp.push\_back(a[j++]);  }  while(i<=mid)temp.push\_back(a[i++]); while(j<=e)temp.push\_back(a[j++]);  for(i=s;i<=e;i++) a[i]=temp[i-s];  }  void mergeSort(vector<int> &a, int s, int e){ if(s==e)return;  int mid=(s+e)>>1;  if(mid-s+1<=16) insertionSort(a, s, e); else mergeSort(a, s, mid); if(e- mid<=16) insertionSort(a, s, e); else mergeSort(a, mid+1, e);  merge(a, s, e);  }  void mergeSort(vector<int> &a, int n){ mergeSort(a, 0, n-1); } |

|  |
| --- |
| int main(){ int n;  cin>>n; vector<int> a(n);  for(int i=0;i<n;i++)cin>>a[i]; mergeSort(a, n); for(int i=0;i<n;i++)cout<<a[i]<<" ";  return 0;  } |
|  |

ASSIGNMENT 6

|  |
| --- |
| **Prog.1:** Write an O(nlogn) algorithm that receives as input two n-element unsorted arrays, a and b of real numbers and a value val such that the algorithm returns true if there exist indexes I and j such that a[i]+b[j] =val, otherwise false. |
| #include <bits/stdc++.h> using namespace std;  int main() { int n, target; cin >> n >> target; vector<int> a1(n), a2(n); for (int i = 0; i < n; i++) cin >> a1[i];  for (int i = 0; i < n; i++)  cin >> a2[i];  sort(a1.begin(), a1.end());  sort(a2.begin(), a2.end());  int i = 0, j = n - 1; bool ans = 0;  while (i < n && j >= 0 ) { if (a1[i]  + a2[j] == target) {  ans = 1; break;  }  else if (a1[i] + a2[j] < target) i++; else if (a1[i] + a2[j] > target) j--;  }  cout << (ans ? "True" : "False");  return 0;  } |



|  |
| --- |
|  |
| **Prog.2:** Implement a Quick sort algorithm where median-of-three partitioning partitions the array. |
| #include <bits/stdc++.h> using namespace std;  int medianThree(int arr[], int a, int b, int c) { if ((arr[a] > arr[b]) ^ (arr[a] > arr[c])) return a;  else if ((arr[b] < arr[a]) ^ (arr[b] < arr[c])) return b;  else return c;  }  int partition(int arr[], int low, int high) { int pivot = arr[high];  int i = (low - 1);  for (int j = low; j <= high - 1; j++) { if (arr[j] <= pivot) {  i++;  swap(arr[i], arr[j]);  }  }  swap(arr[i + 1], arr[high]); return (i + 1);  }  int partition\_r(int arr[], int low, int high) {  int median = medianThree(arr, low, (low+high)/2, high); swap(arr[median], arr[high]); |



|  |
| --- |
| return partition(arr, low, high);  }  void quickSort(int arr[], int low, int high) { if (low < high) {  int pi = partition\_r(arr, low, high); quickSort(arr, low, pi - 1); quickSort(arr, pi + 1, high); }  }  int main() { int n;  cin>>n; int arr[n];  for(int i=0;i<n;i++) cin>>arr[i];  quickSort(arr, 0, n - 1); cout<<"Sorted array: \n";  for(int i=0;i<n;i++) cout<<arr[i]<<" ";  return 0;  } |
|  |
| **Prog.3:** Implement a Quick sort algorithm with the help of a random pivot. |
| #include <bits/stdc++.h> using namespace std; |



|  |
| --- |
| int partition(int arr[], int low, int high) { int pivot = arr[high];  int i = (low - 1);  for (int j = low; j <= high - 1; j++) { if (arr[j] <= pivot) {  i++;  swap(arr[i], arr[j]);  }  }  swap(arr[i + 1], arr[high]); return (i + 1);  }  int partition\_r(int arr[], int low, int high) { srand(time(NULL));  int random = low + rand() % (high - low); swap(arr[random], arr[high]); return partition(arr, low, high);  }  void quickSort(int arr[], int low, int high) { if (low < high) {  int pi = partition\_r(arr, low, high); quickSort(arr, low, pi - 1); quickSort(arr, pi + 1, high); }  }  int main() { int n;  cin>>n; int arr[n];  for(int i=0;i<n;i++) cin>>arr[i];  quickSort(arr, 0, n - 1); cout<<"Sorted array: \n";  for(int i=0;i<n;i++) cout<<arr[i]<<" ";  return 0;  } |



|  |
| --- |
|  |

