Analysis and Design of Algorithms

Lab File

Name – Tushya Gupta
Roll number – UE233106
Group – 6

Index

Quicksort

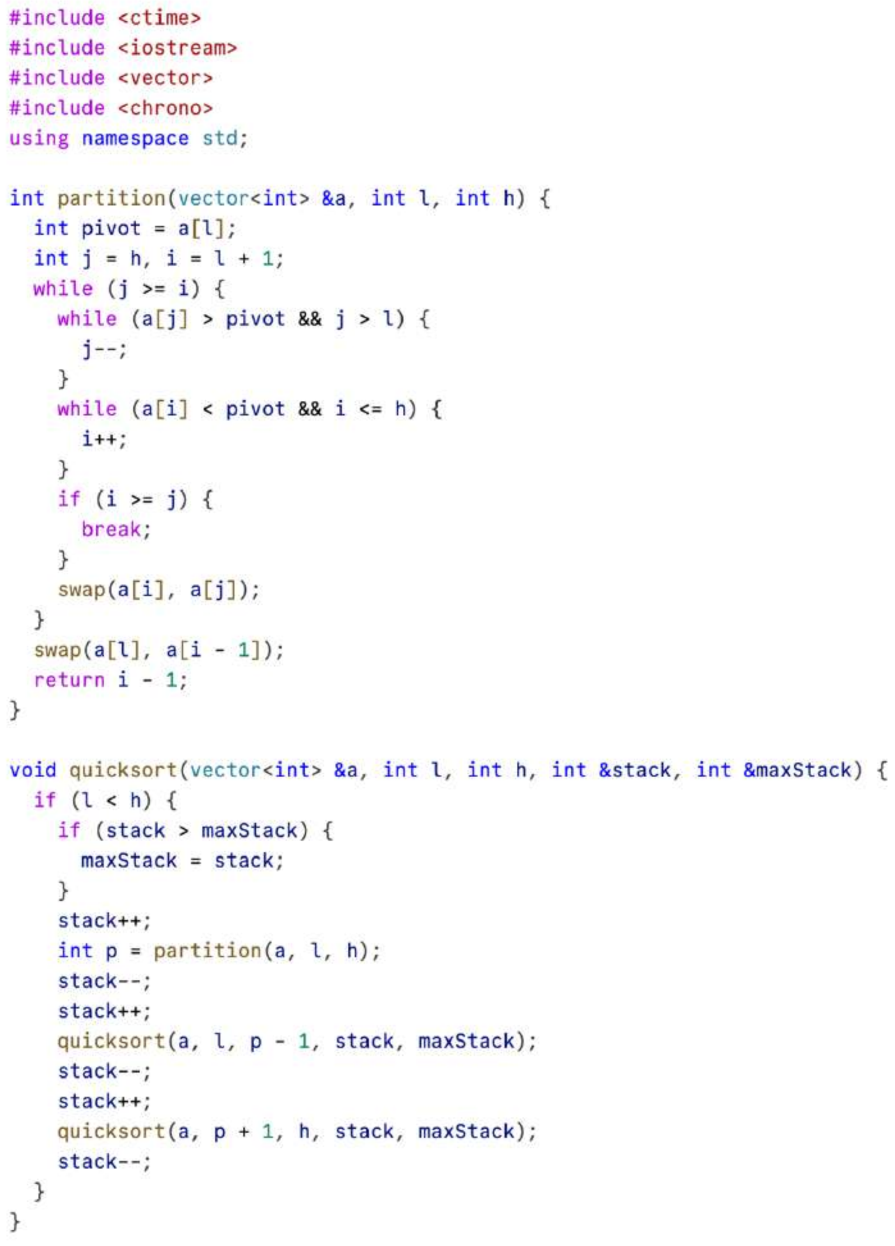
Idea –

- sorting of an array using a pivot ensuring the elements to the left of the pivot are smaller than the pivot and elements right of pivot are larger than it
- The pivot is put in the position where it should be when the whole array is sorted
- Uses a recursive structure to sort each side split along the index of the pivot

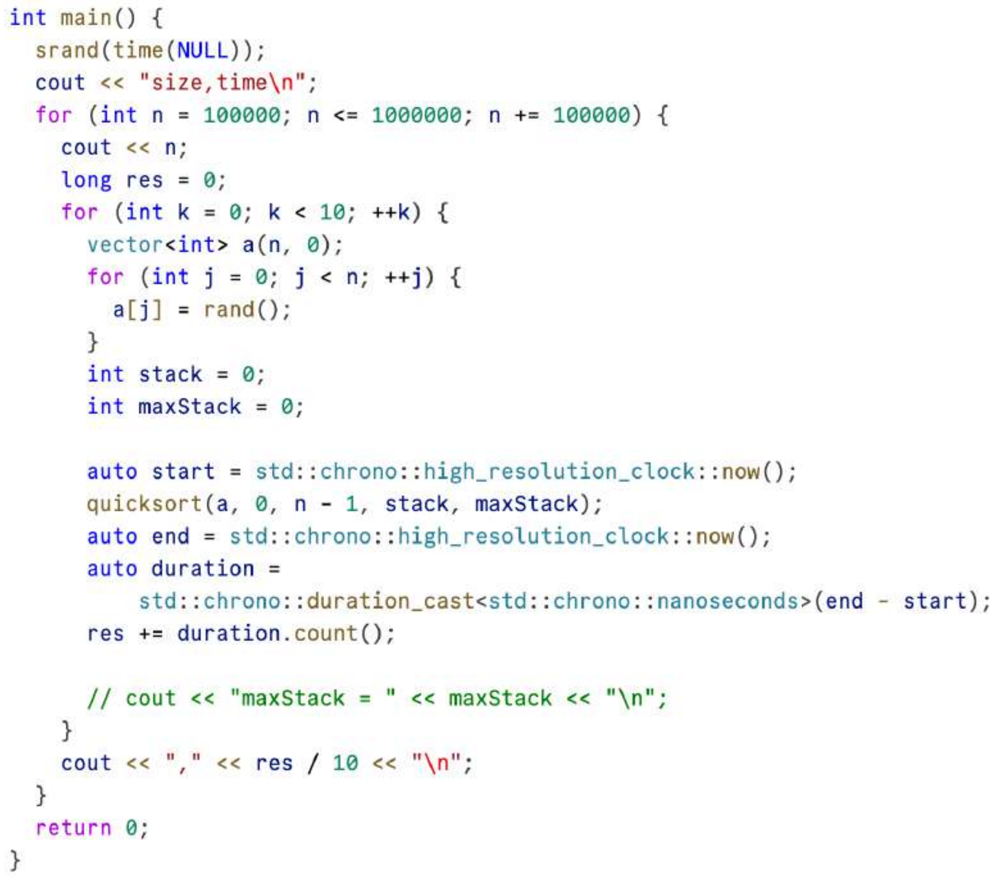
Complexity –

- Time – O(nlogn)
- Space – O(n)

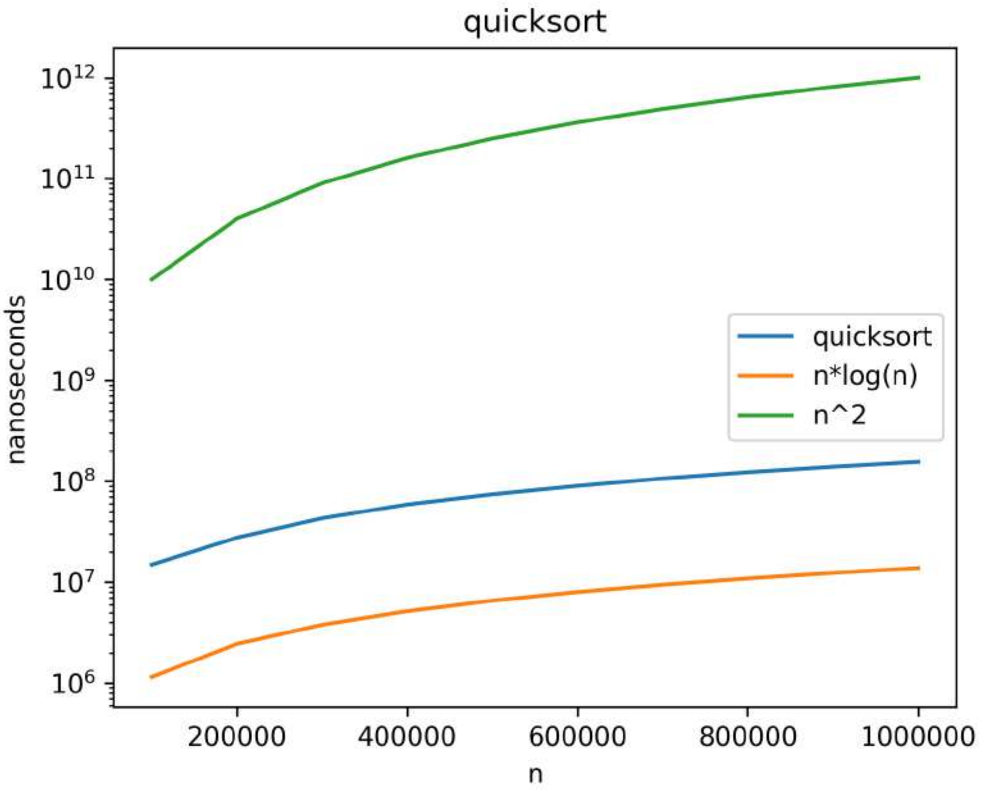
Code –



```
#include <ctime>
#include <iostream>
#include <vector>
#include <chrono>
using namespace std;

int partition(vector<int> &a, int l, int h) {
  int pivot = a[l];
  int j = h, i = l + 1;
  while (j >= i) {
    while (a[j] > pivot && j > l) {
      j--;
    }
    while (a[i] < pivot && i <= h) {
      i++;
    }
    if (i >= j) {
      break;
    }
    swap(a[i], a[j]);
  }
  swap(a[l], a[i - 1]);
  return i - 1;
}

void quicksort(vector<int> &a, int l, int h, int &stack, int &maxStack) {
  if (l < h) {
    if (stack > maxStack) {
      maxStack = stack;
    }
    stack++;
    int p = partition(a, l, h);
    stack--;
    stack++;
    quicksort(a, l, p - 1, stack, maxStack);
    stack--;
    stack++;
    quicksort(a, p + 1, h, stack, maxStack);
    stack--;
  }
}
```



```
int main() {
  srand(time(NULL));
  cout << "size,time\n";
  for (int n = 100000; n <= 1000000; n += 100000) {
    cout << n;
    long res = 0;
    for (int k = 0; k < 10; ++k) {
      vector<int> a(n, 0);
      for (int j = 0; j < n; ++j) {
        a[j] = rand();
      }
      int stack = 0;
      int maxStack = 0;

      auto start = std::chrono::high_resolution_clock::now();
      quicksort(a, 0, n - 1, stack, maxStack);
      auto end = std::chrono::high_resolution_clock::now();
      auto duration =
          std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
      res += duration.count();

      // cout << "maxStack = " << maxStack << "\n";
    }
    cout << "," << res / 10 << "\n";
  }
  return 0;
}
```



Iterative Quicksort

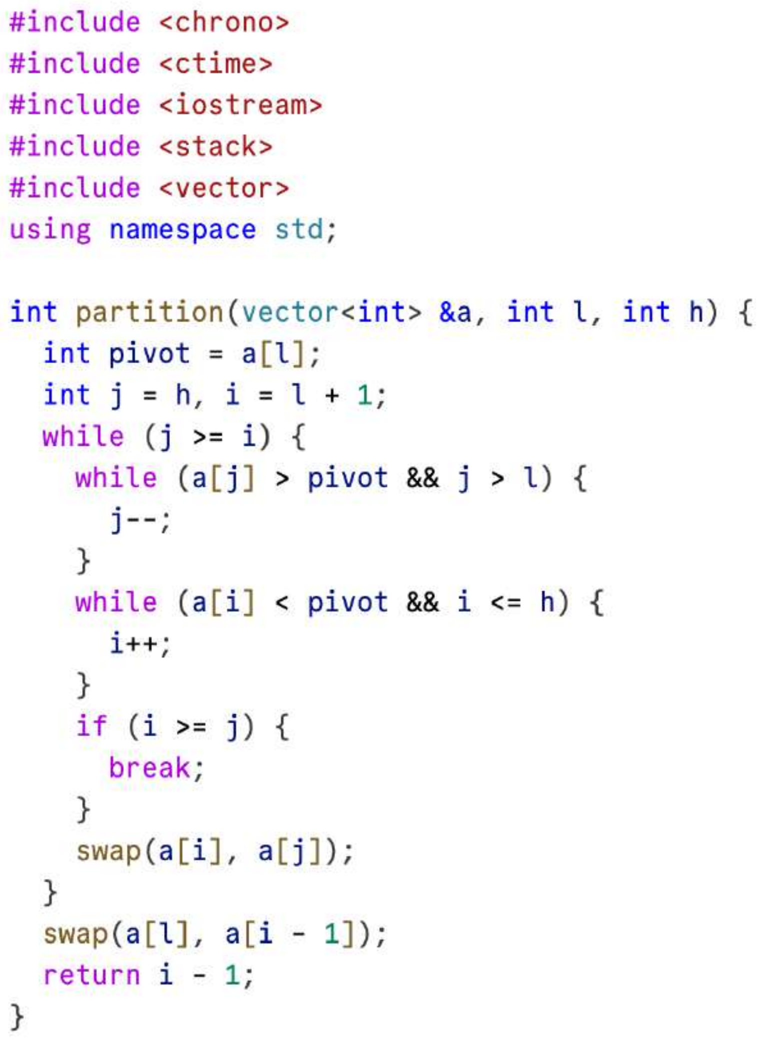
Idea –

- Similar to recursive quicksort but uses a stack to maintain the index of the places where the pivot cut the array

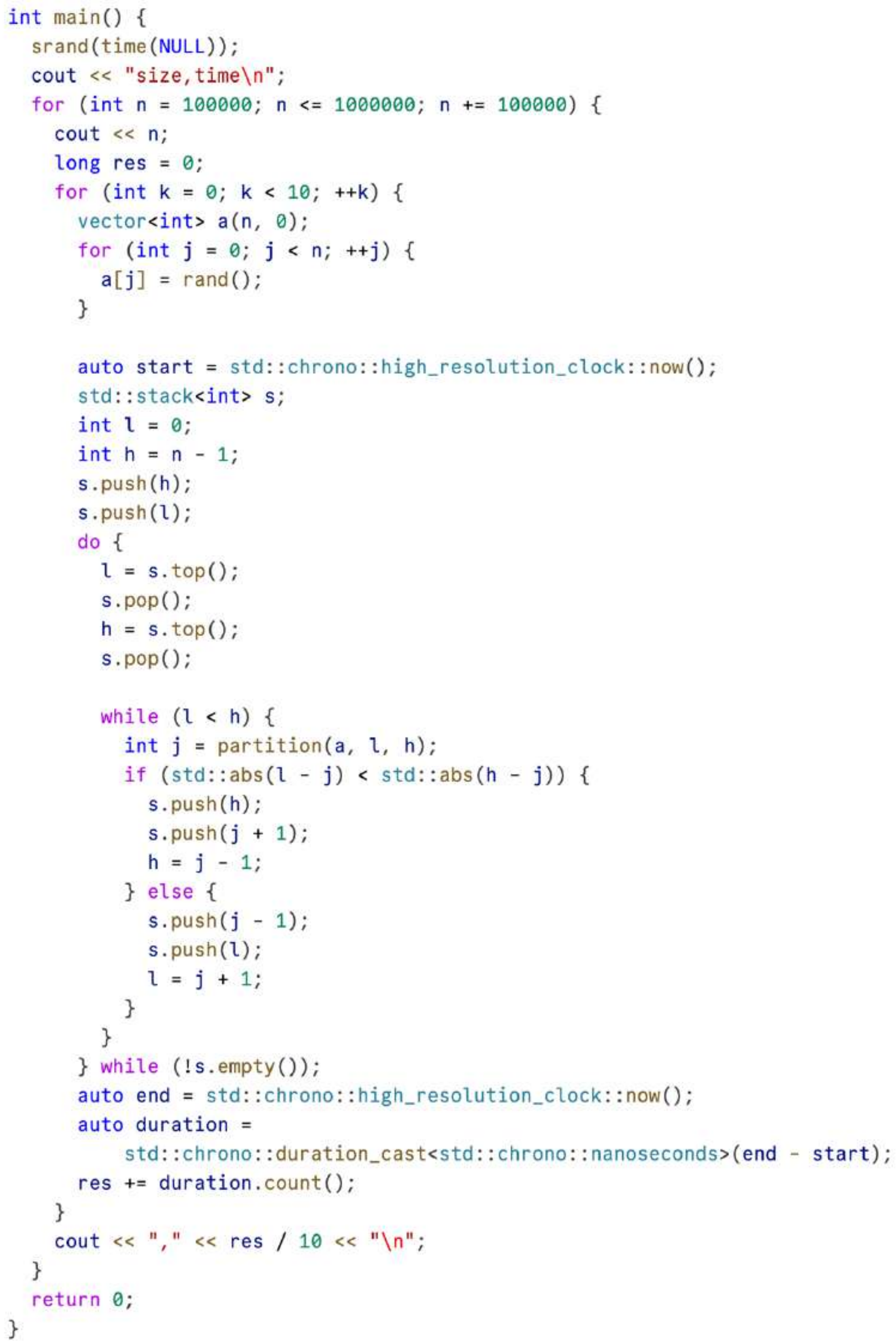
Complexity –

- Time – O(nlogn)
- Space – O(logn)

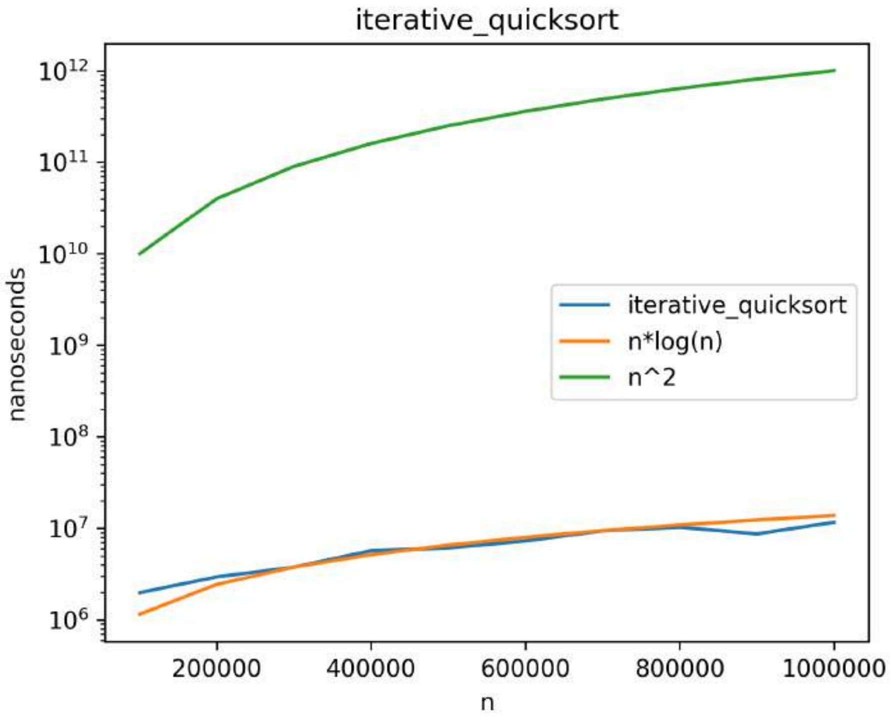
Code –



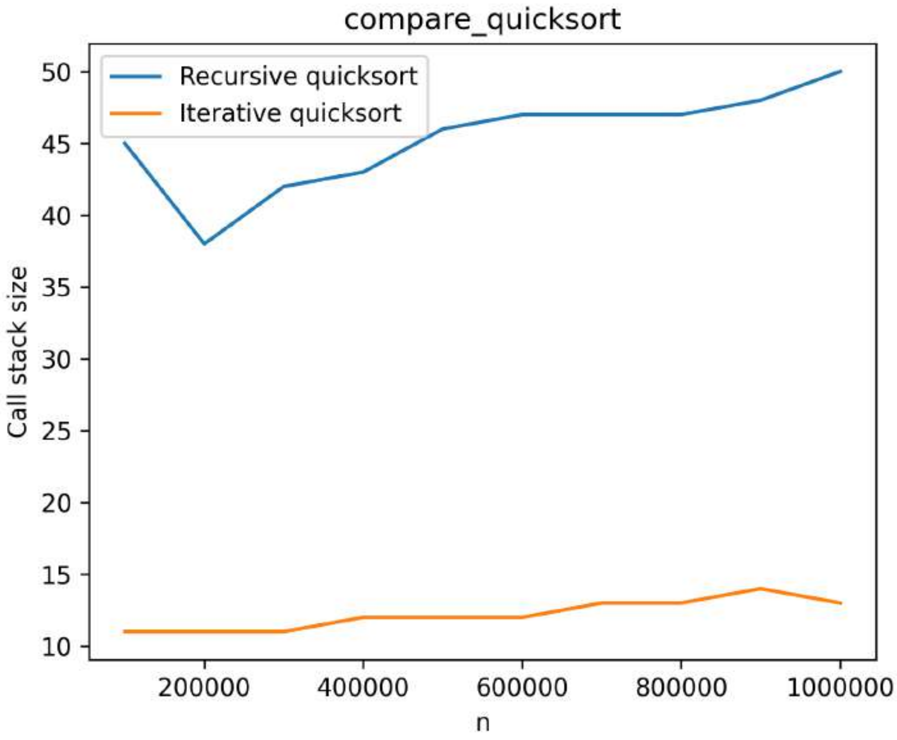
```
#include <chrono>
#include <ctime>
#include <iostream>
#include <stack>
#include <vector>
using namespace std;

int partition(vector<int> &a, int l, int h) {
  int pivot = a[l];
  int j = h, i = l + 1;
  while (j >= i) {
    while (a[j] > pivot && j > l) {
      j--;
    }
    while (a[i] < pivot && i <= h) {
      i++;
    }
    if (i >= j) {
      break;
    }
    swap(a[i], a[j]);
  }
  swap(a[l], a[i - 1]);
  return i - 1;
}
```



```cpp
int main() {
  srand(time(NULL));
  cout << "size,time\n";
  for (int n = 100000; n <= 1000000; n += 100000) {
    cout << n;
    long res = 0;
    for (int k = 0; k < 10; ++k) {
      vector<int> a(n, 0);
      for (int j = 0; j < n; ++j) {
        a[j] = rand();
      }

      auto start = std::chrono::high_resolution_clock::now();
      std::stack<int> s;
      int l = 0;
      int h = n - 1;
      s.push(h);
      s.push(l);
      do {
        l = s.top();
        s.pop();
        h = s.top();
        s.pop();

        while (l < h) {
          int j = partition(a, l, h);
          if (std::abs(l - j) < std::abs(h - j)) {
            s.push(h);
            s.push(j + 1);
            h = j - 1;
          } else {
            s.push(j - 1);
            s.push(l);
            l = j + 1;
          }
        }
      } while (!s.empty());
      auto end = std::chrono::high_resolution_clock::now();
      auto duration =
          std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
      res += duration.count();
    }
    cout << "," << res / 10 << "\n";
  }
  return 0;
}
```



Comparing Iterative vs Stack Quicksort



1D Peak

Idea –

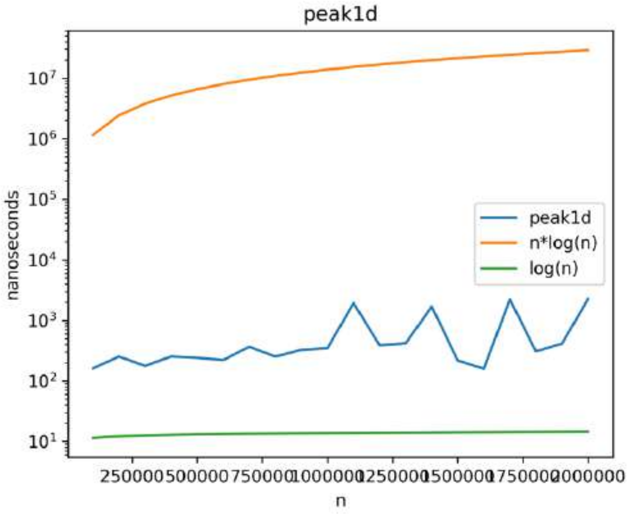
- Finding a number that is larger than the one on its left and right
- Using binary search algorithm finds a peak

Complexity –

- Time – O(logn)
- Space – O(1)

Code –

```
#include <chrono>
#include <ctime>
#include <iostream>
using namespace std;

int peak(int a[], int l, int h, int n) {
  // if (l > h) {
  //   return -1;
  // }
  int mid = (l + h) / 2;
  if (mid < n - 1 && a[mid] < a[mid + 1]) {
    return peak(a, mid + 1, h, n);
  } else if (mid > 0 && a[mid] < a[mid - 1]) {
    return peak(a, l, mid - 1, n);
  } else {
    return mid;
  }
}

int main() {
  srand(time(NULL));
  cout << "size,time\n";
  for (int n = 100000; n <= 2000000; n += 100000) {
    cout << n;
    long res = 0;
    for (int k = 0; k < 10; ++k) {
      int a[n];
      for (int j = 0; j < n; ++j) {
        a[j] = rand();
      }
      int stack = 0;
      int maxStack = 0;

      auto start = std::chrono::high_resolution_clock::now();
      peak(a, 0, n - 1, n);
      auto end = std::chrono::high_resolution_clock::now();
      auto duration =
          std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
      res += duration.count();
    }
    cout << "," << res / 10 << "\n";
  }
  return 0;
}
```



2D Peak

Idea –

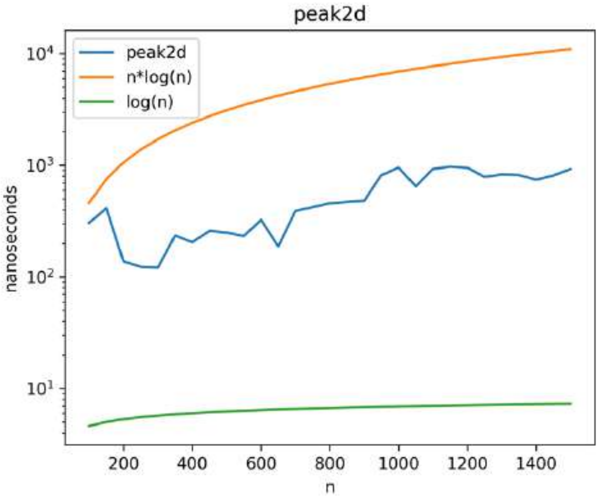
- Finding a number that is larger than the one on all its side(left, right, up, down)
- Using binary search algorithm finds a peak but in a 2D fashion

Complexity –

- Time – O(logn)
- Space – O(1)

Code –

```
#include <chrono>
#include <ctime>
#include <iostream>
using namespace std;

#define N 5

int peak(int **a, int y1, int y2, int x1, int x2, int n) {
  int m1 = (y1 + y2) / 2;
  int m2 = (x1 + x2) / 2;
  if (m2 < n - 1 && x2> m2+1 && a[m1][m2] < a[m1][m2 + 1]) {
    return peak(a, y1, y2, m2 + 1, x2, n);
  } else if (m2 > 0 && x1 < m2-1 && a[m1][m2] < a[m1][m2 - 1]) {
    return peak(a, y1, y2, x1, m2 - 1, n);
  } else if (m1 < n - 1 && y2 > m1+1 && a[m1][m2] < a[m1 + 1][m2]) {
    return peak(a, m1 + 1, y2, x1, x2, n);
  } else if (m1 > 0 && y1 < m1-1 && a[m1][m2] < a[m1 - 1][m2]) {
    return peak(a, y1, m1 - 1, x1, x2, n);
  } else {
    return a[m1][m2];
  }
}

int main() {
  srand(time(NULL));
  cout << "size,time\n";
  for (int n = 100; n <= 1500; n += 50) {
    cout << n;
    long res = 0;
    for (int k = 0; k < 10; ++k) {
      int **a = new int *[n];
      for (int i = 0; i < n; i++) {
        a[i] = new int[n];
        for (int j = 0; j < n; ++j) {
          a[i][j] = rand();
        }
      }
      int stack = 0;
      int maxStack = 0;

      auto start = std::chrono::high_resolution_clock::now();
      peak(a, 0, n - 1, 0, n - 1, n);
      auto end = std::chrono::high_resolution_clock::now();
      auto duration =
          std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
      res += duration.count();

      // cout << "maxStack = " << maxStack << "\n";
    }
    cout << "," << res / 10 << "\n";
  }
  return 0;
}
```



Magic Square

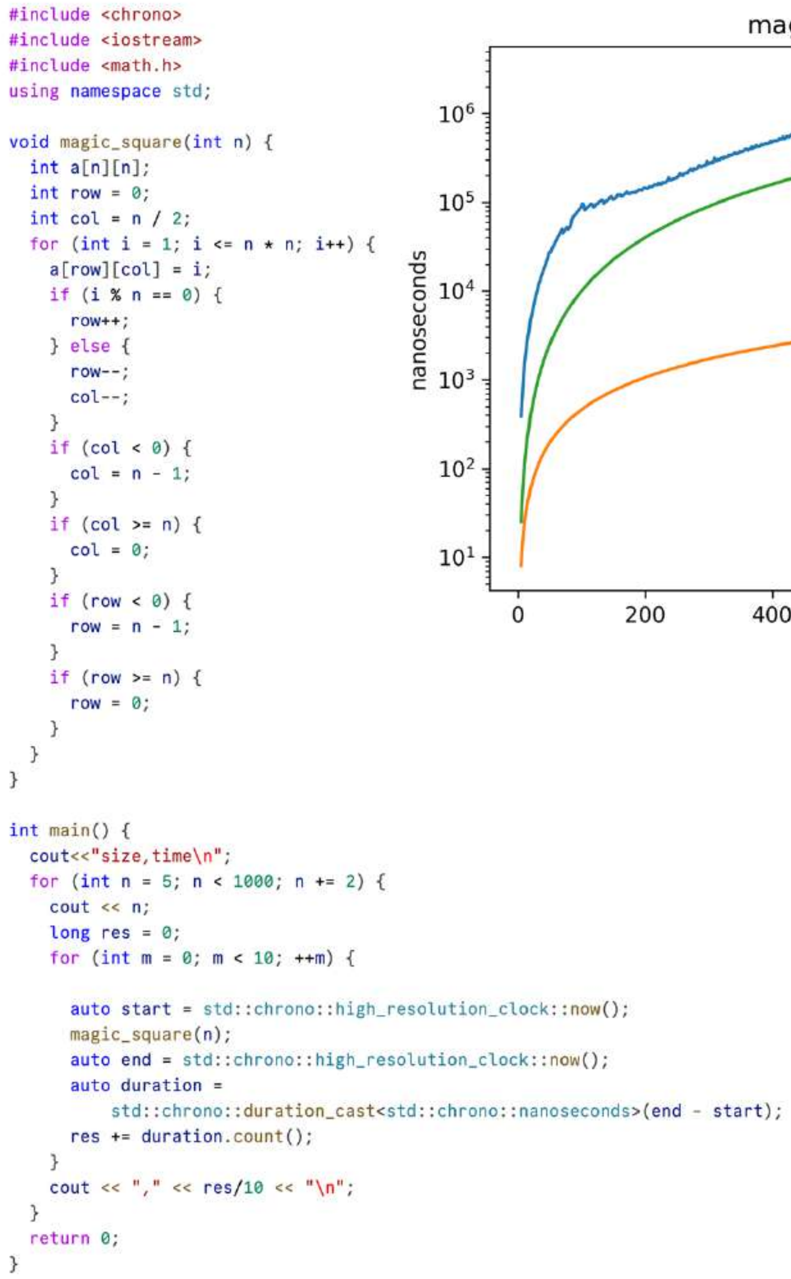
Idea –

- Create a matrix in with the sum of all its rows, columns and diagonals is equal
- Follows a set pattern in where the numbers increase in a diagonal upwards to the left and goes down one when number is a multiple of the size. It starts from the center of the first row

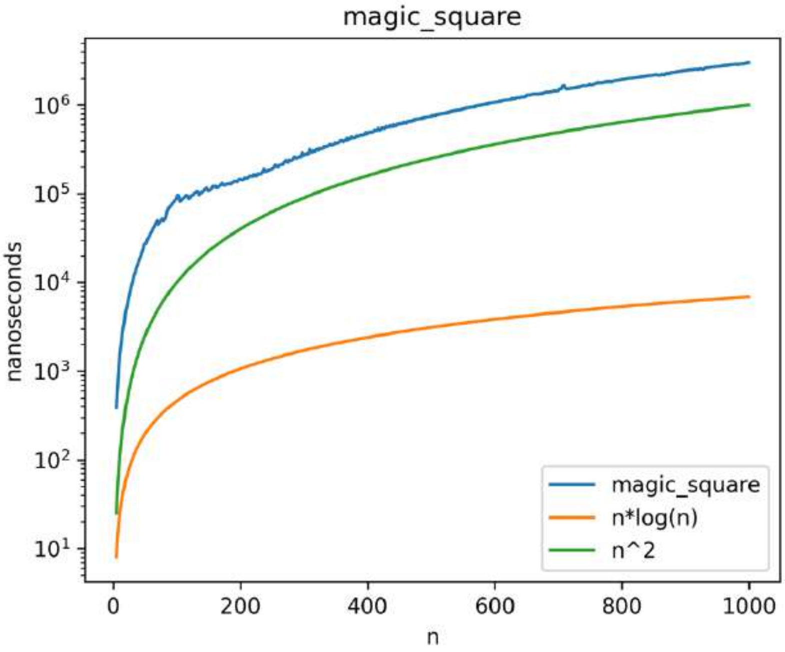
Complexity –

- Time – O(n^2)
- Space – O(n^2)

Code –



```
#include <chrono>
#include <iostream>
#include <math.h>
using namespace std;

void magic_square(int n) {
  int a[n][n];
  int row = 0;
  int col = n / 2;
  for (int i = 1; i <= n * n; i++) {
    a[row][col] = i;
    if (i % n == 0) {
      row++;
    } else {
      row--;
      col--;
    }
    if (col < 0) {
      col = n - 1;
    }
    if (col >= n) {
      col = 0;
    }
    if (row < 0) {
      row = n - 1;
    }
    if (row >= n) {
      row = 0;
    }
  }
}

int main() {
  cout<<"size,time\n";
  for (int n = 5; n < 1000; n += 2) {
    cout << n;
    long res = 0;
    for (int m = 0; m < 10; ++m) {

      auto start = std::chrono::high_resolution_clock::now();
      magic_square(n);
      auto end = std::chrono::high_resolution_clock::now();
      auto duration =
          std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
      res += duration.count();
    }
    cout << "," << res/10 << "\n";
  }
  return 0;
}
```



Cosine Similarity

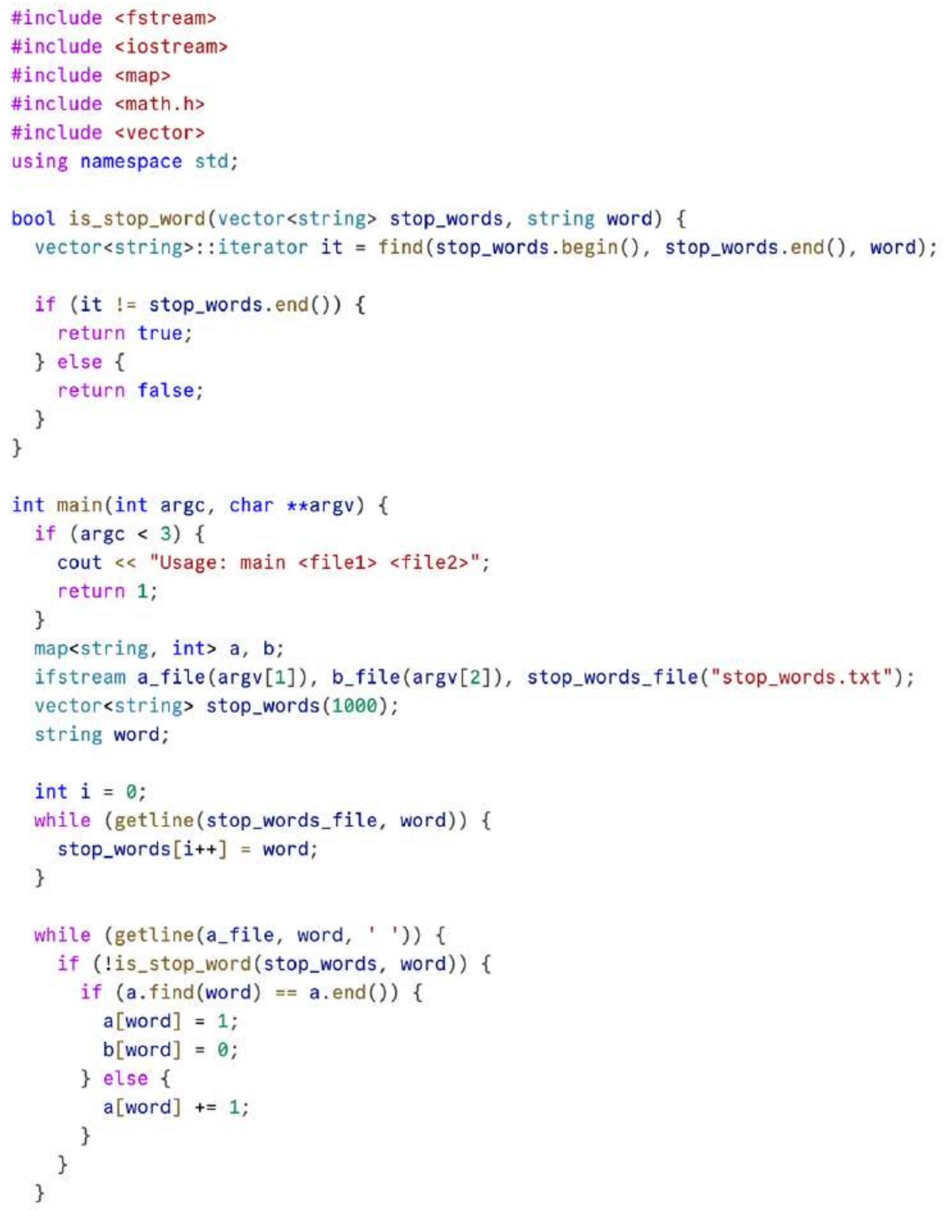
Idea –

- Find out the similarity between two documents by checking the number of time each word occurs in each file
- Each files words are treated like a vector and the angle between the vectors created from the two file give the value for cosQ which tells us the similarity between the files

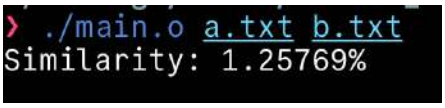
Complexity –

- Time – O(n+m)
- Space – O(n+m)

Code –



```
#include <fstream>
#include <iostream>
#include <map>
#include <math.h>
#include <vector>
using namespace std;

bool is_stop_word(vector<string> stop_words, string word) {
  vector<string>::iterator it = find(stop_words.begin(), stop_words.end(), word);

  if (it != stop_words.end()) {
    return true;
  } else {
    return false;
  }
}

int main(int argc, char **argv) {
  if (argc < 3) {
    cout << "Usage: main <file1> <file2>";
    return 1;
  }
  map<string, int> a, b;
  ifstream a_file(argv[1]), b_file(argv[2]), stop_words_file("stop_words.txt");
  vector<string> stop_words(1000);
  string word;

  int i = 0;
  while (getline(stop_words_file, word)) {
    stop_words[i++] = word;
  }

  while (getline(a_file, word, ' ')) {
    if (!is_stop_word(stop_words, word)) {
      if (a.find(word) == a.end()) {
        a[word] = 1;
        b[word] = 0;
      } else {
        a[word] += 1;
      }
    }
  }
  while (getline(b_file, word, ' ')) {
    if (!is_stop_word(stop_words, word)) {
      if (b.find(word) == b.end()) {
        b[word] = 1;
        a[word] = 0;
      } else {
        b[word] += 1;
      }
    }
  }

  float a_mag = 0, b_mag = 0, a_dot_b = 0;
  map<string, int>::iterator a_iter = a.begin(), b_iter = b.begin();
  while (a_iter != a.end() && b_iter != b.end()) {
    a_dot_b += a_iter->second * b_iter->second;
    a_mag += a_iter->second * a_iter->second;
    b_mag += b_iter->second * b_iter->second;
    a_iter++;
    b_iter++;
  }
  a_mag = sqrt(a_mag);
  b_mag = sqrt(b_mag);

  float cosine = a_dot_b / (a_mag * b_mag);
  cout << "Similarity: " << cosine * 100 << "%\n";

  a_file.close();
  b_file.close();
  return 0;
}
```



```
❯ ./main.o a.txt b.txt
Similarity: 1.25769%
```

Fractional Knapsack

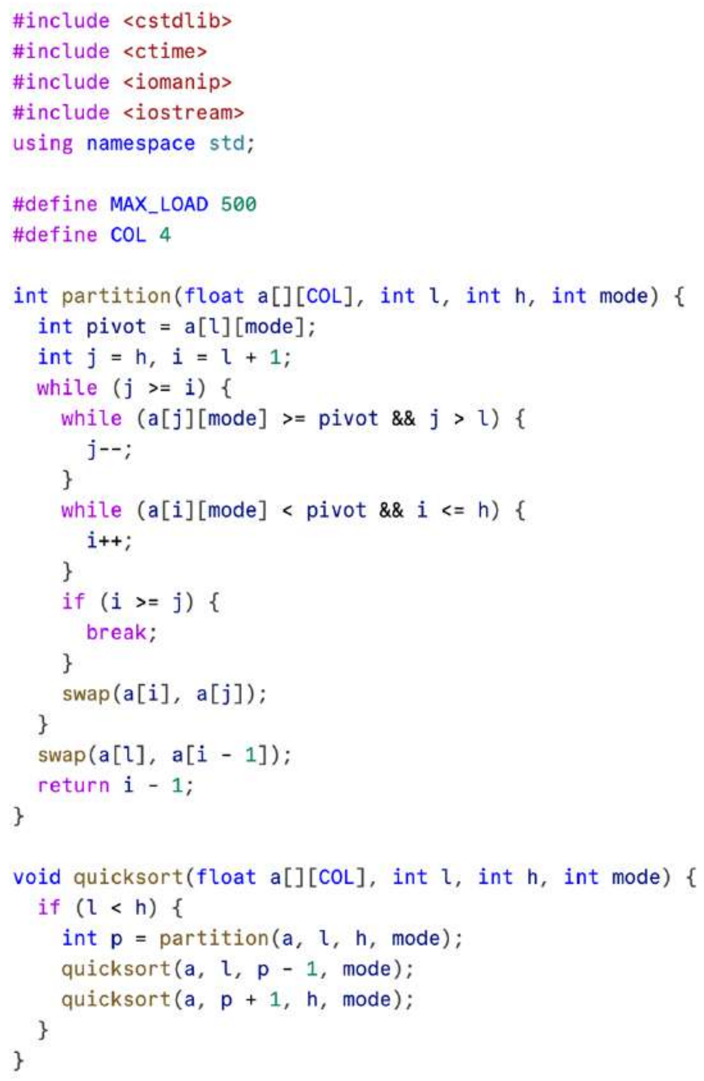
Idea –

- Problem involving the calculation of max(here price) that can be gotten from a set of values
- We sort the array according to the metric we need(here we sort 3 time, according to profit, weight and ratio of profit/weight) and then pick the items in non-increasing manner
- The fractional part allows us to pick out items in fraction
    - For example if we have a capacity of 10 and the item weights 20 we can take out 10 out the weight instead of leaving that item and going to the next

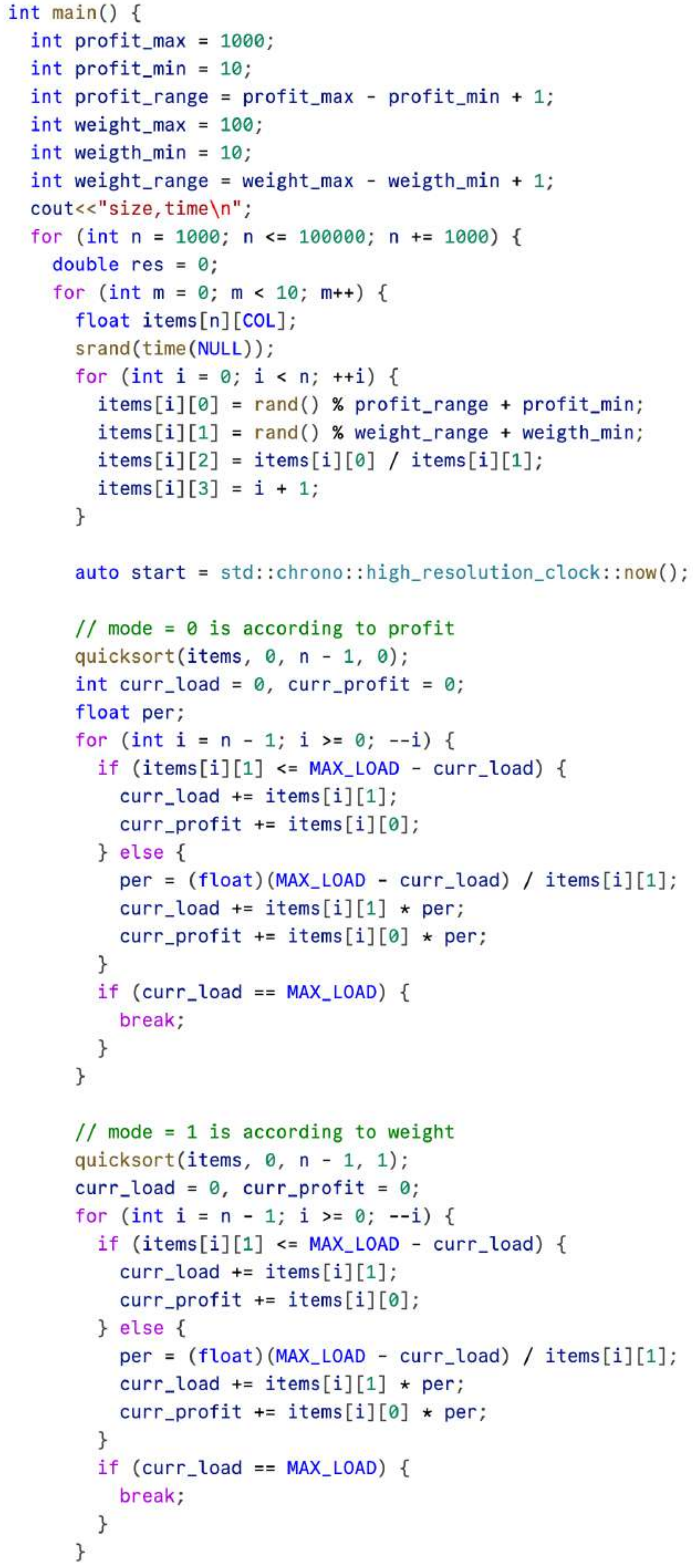
Complexity –

- Time – O(nlogn)
- Space – O(1)

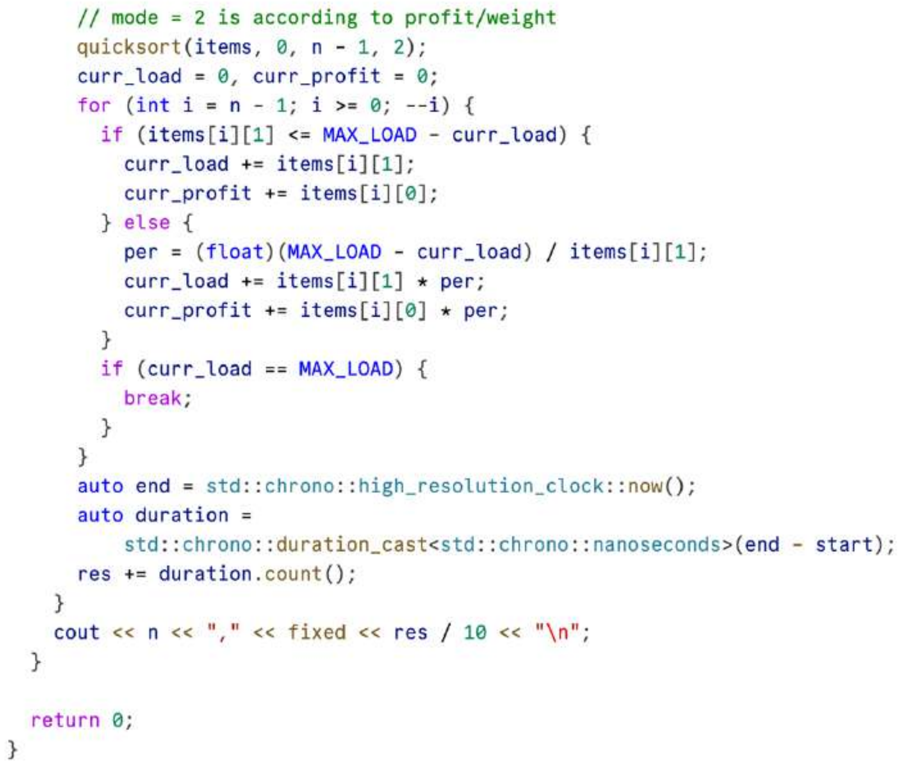
Code –



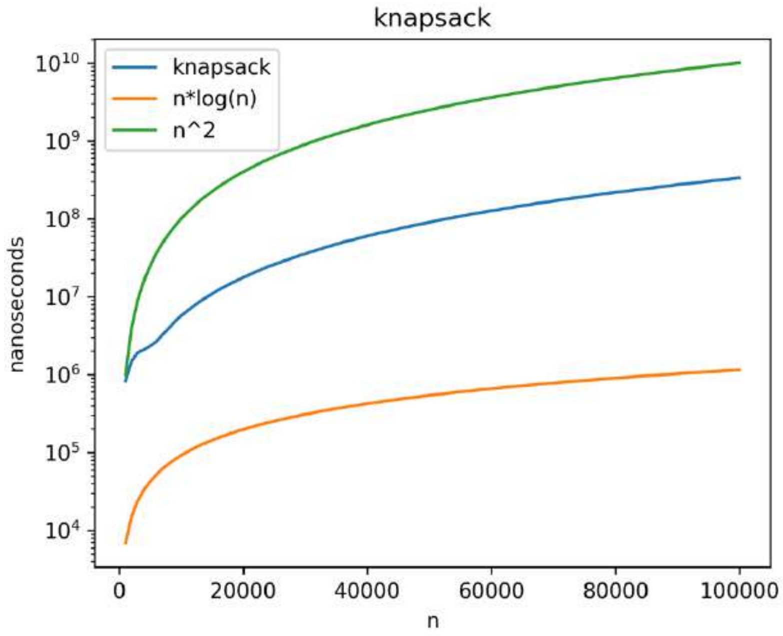
```
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <iostream>
using namespace std;

#define MAX_LOAD 500
#define COL 4

int partition(float a[][COL], int l, int h, int mode) {
  int pivot = a[l][mode];
  int j = h, i = l + 1;
  while (j >= i) {
    while (a[j][mode] >= pivot && j > l) {
      j--;
    }
    while (a[i][mode] < pivot && i <= h) {
      i++;
    }
    if (i >= j) {
      break;
    }
    swap(a[i], a[j]);
  }
  swap(a[l], a[i - 1]);
  return i - 1;
}

void quicksort(float a[][COL], int l, int h, int mode) {
  if (l < h) {
    int p = partition(a, l, h, mode);
    quicksort(a, l, p - 1, mode);
    quicksort(a, p + 1, h, mode);
  }
}
```



```
int main() {
  int profit_max = 1000;
  int profit_min = 10;
  int profit_range = profit_max - profit_min + 1;
  int weight_max = 100;
  int weigth_min = 10;
  int weight_range = weight_max - weigth_min + 1;
  cout<<"size,time\n";
  for (int n = 1000; n <= 100000; n += 1000) {
    double res = 0;
    for (int m = 0; m < 10; m++) {
      float items[n][COL];
      srand(time(NULL));
      for (int i = 0; i < n; ++i) {
        items[i][0] = rand() % profit_range + profit_min;
        items[i][1] = rand() % weight_range + weigth_min;
        items[i][2] = items[i][0] / items[i][1];
        items[i][3] = i + 1;
      }

      auto start = std::chrono::high_resolution_clock::now();

      // mode = 0 is according to profit
      quicksort(items, 0, n - 1, 0);
      int curr_load = 0, curr_profit = 0;
      float per;
      for (int i = n - 1; i >= 0; --i) {
        if (items[i][1] <= MAX_LOAD - curr_load) {
          curr_load += items[i][1];
          curr_profit += items[i][0];
        } else {
          per = (float)(MAX_LOAD - curr_load) / items[i][1];
          curr_load += items[i][1] * per;
          curr_profit += items[i][0] * per;
        }
        if (curr_load == MAX_LOAD) {
          break;
        }
      }

      // mode = 1 is according to weight
      quicksort(items, 0, n - 1, 1);
      curr_load = 0, curr_profit = 0;
      for (int i = n - 1; i >= 0; --i) {
        if (items[i][1] <= MAX_LOAD - curr_load) {
          curr_load += items[i][1];
          curr_profit += items[i][0];
        } else {
          per = (float)(MAX_LOAD - curr_load) / items[i][1];
          curr_load += items[i][1] * per;
          curr_profit += items[i][0] * per;
        }
        if (curr_load == MAX_LOAD) {
          break;
        }
      }
```



```
      // mode = 2 is according to profit/weight
      quicksort(items, 0, n - 1, 2);
      curr_load = 0, curr_profit = 0;
      for (int i = n - 1; i >= 0; --i) {
        if (items[i][1] <= MAX_LOAD - curr_load) {
          curr_load += items[i][1];
          curr_profit += items[i][0];
        } else {
          per = (float)(MAX_LOAD - curr_load) / items[i][1];
          curr_load += items[i][1] * per;
          curr_profit += items[i][0] * per;
        }
        if (curr_load == MAX_LOAD) {
          break;
        }
      }
      auto end = std::chrono::high_resolution_clock::now();
      auto duration =
          std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
      res += duration.count();
    }
    cout << n << "," << fixed << res / 10 << "\n";
  }

  return 0;
}
```



Job Scheduling

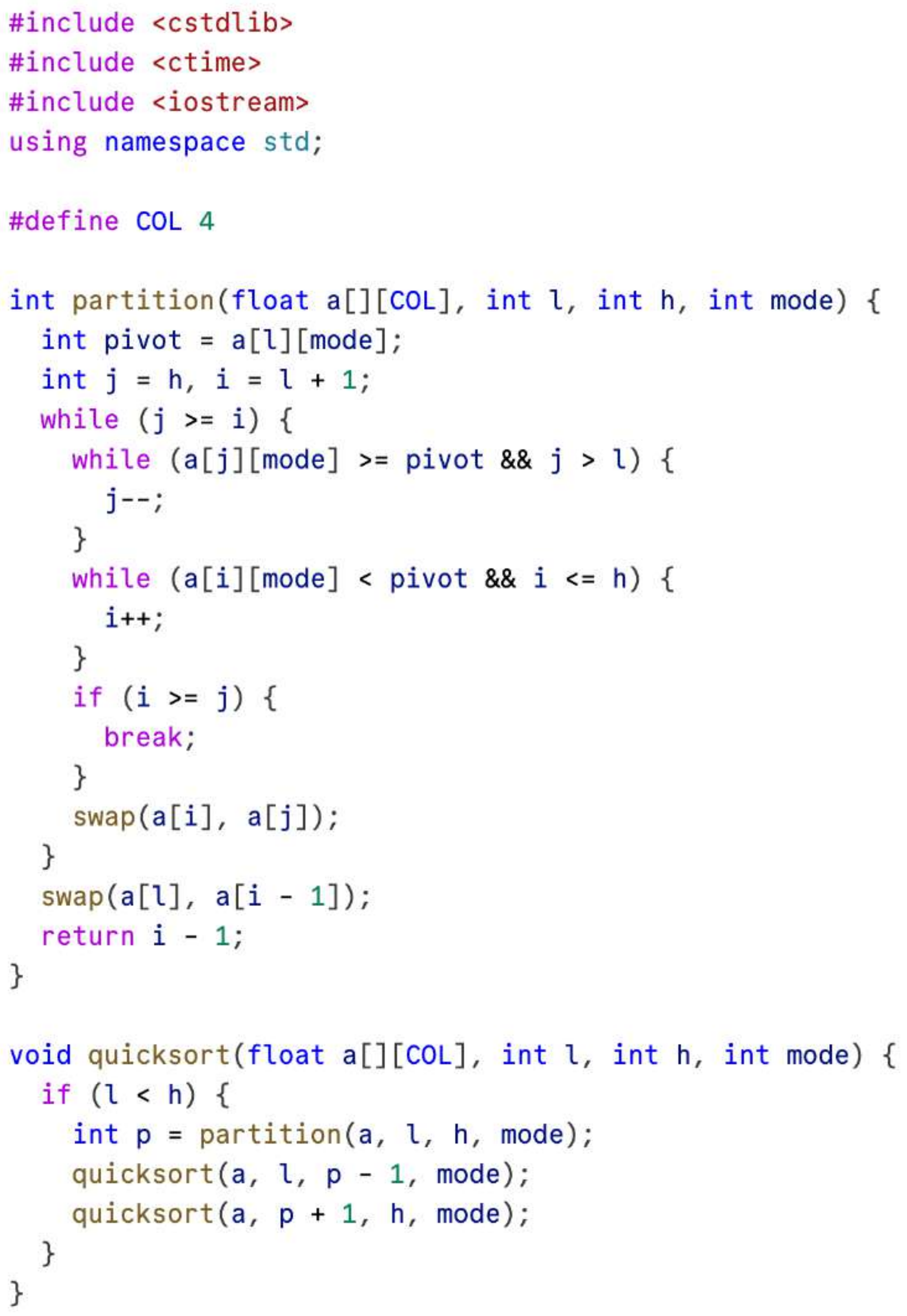
Idea –

- Schedule all jobs for the day ensuring maximum number of jobs are done
- FCFS, SJF and sorting by final_time used

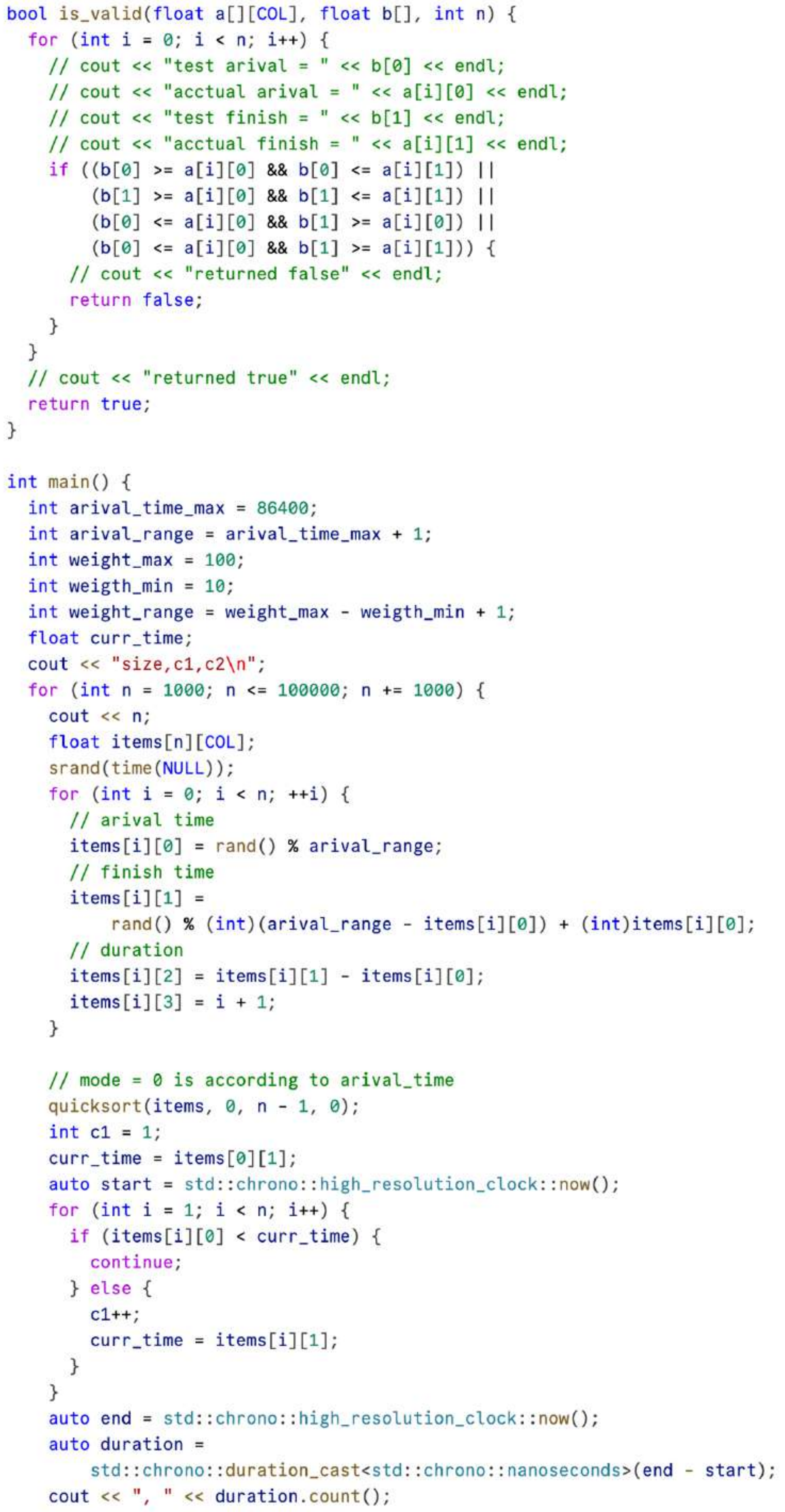
Complexity –

- Time – O(nlogn)
- Space – O(1)

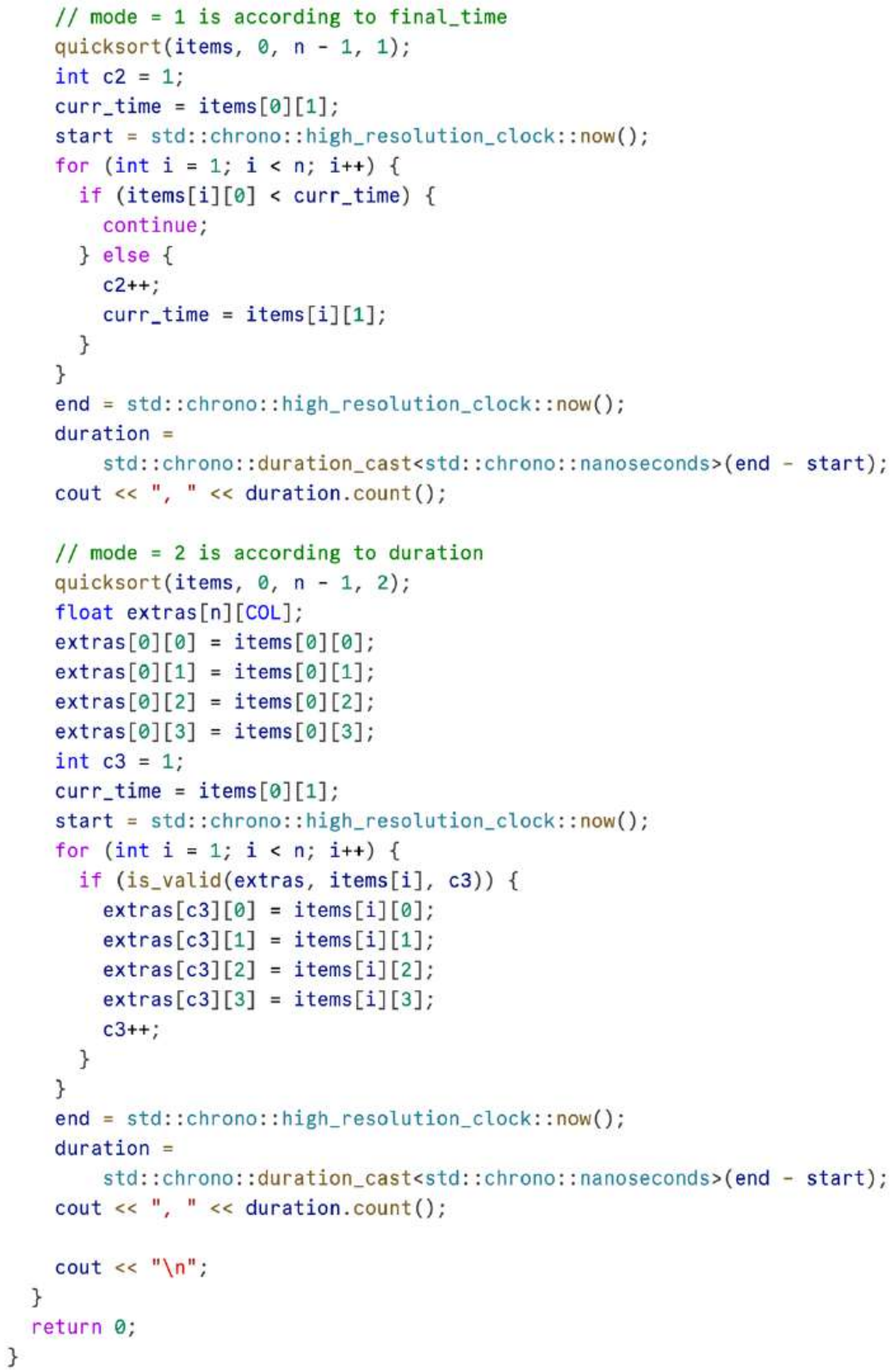
Code –



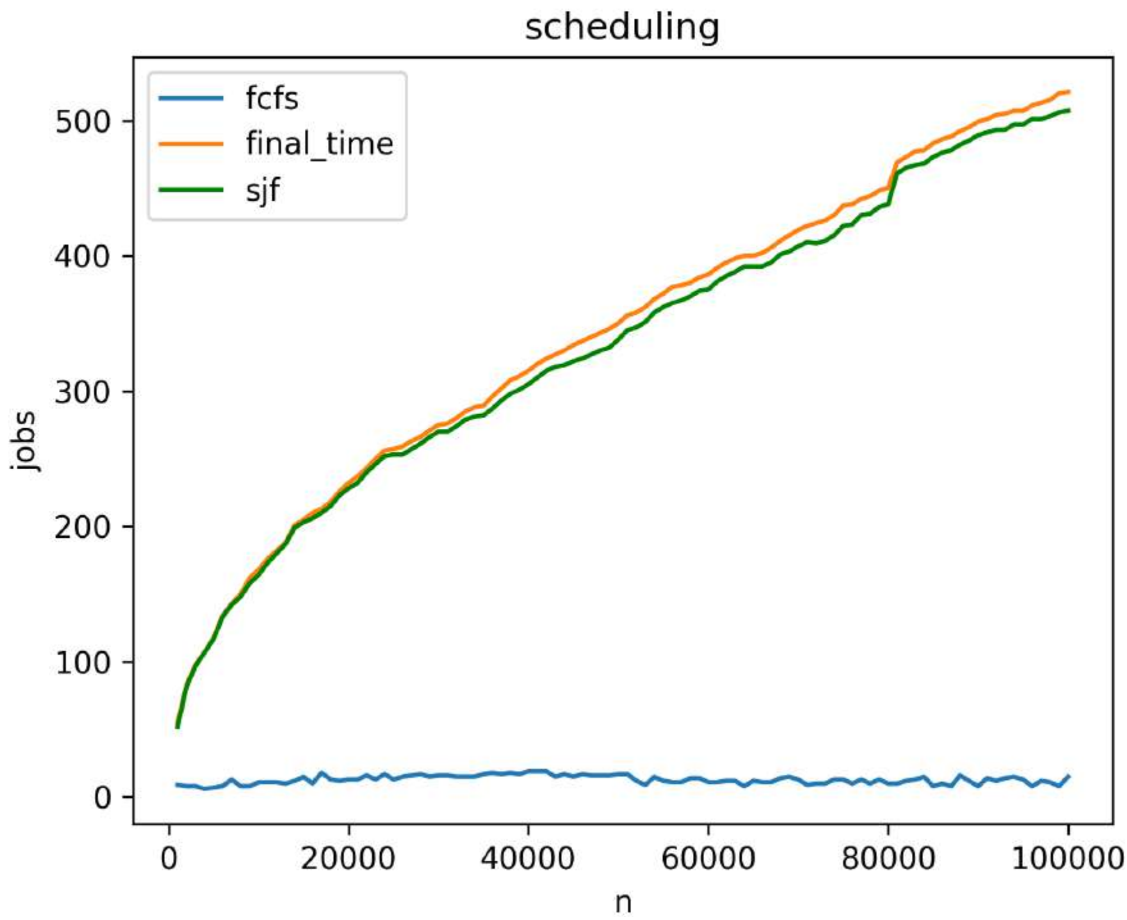
```
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

#define COL 4

int partition(float a[][COL], int l, int h, int mode) {
  int pivot = a[l][mode];
  int j = h, i = l + 1;
  while (j >= i) {
    while (a[j][mode] >= pivot && j > l) {
      j--;
    }
    while (a[i][mode] < pivot && i <= h) {
      i++;
    }
    if (i >= j) {
      break;
    }
    swap(a[i], a[j]);
  }
  swap(a[l], a[i - 1]);
  return i - 1;
}

void quicksort(float a[][COL], int l, int h, int mode) {
  if (l < h) {
    int p = partition(a, l, h, mode);
    quicksort(a, l, p - 1, mode);
    quicksort(a, p + 1, h, mode);
  }
}
```

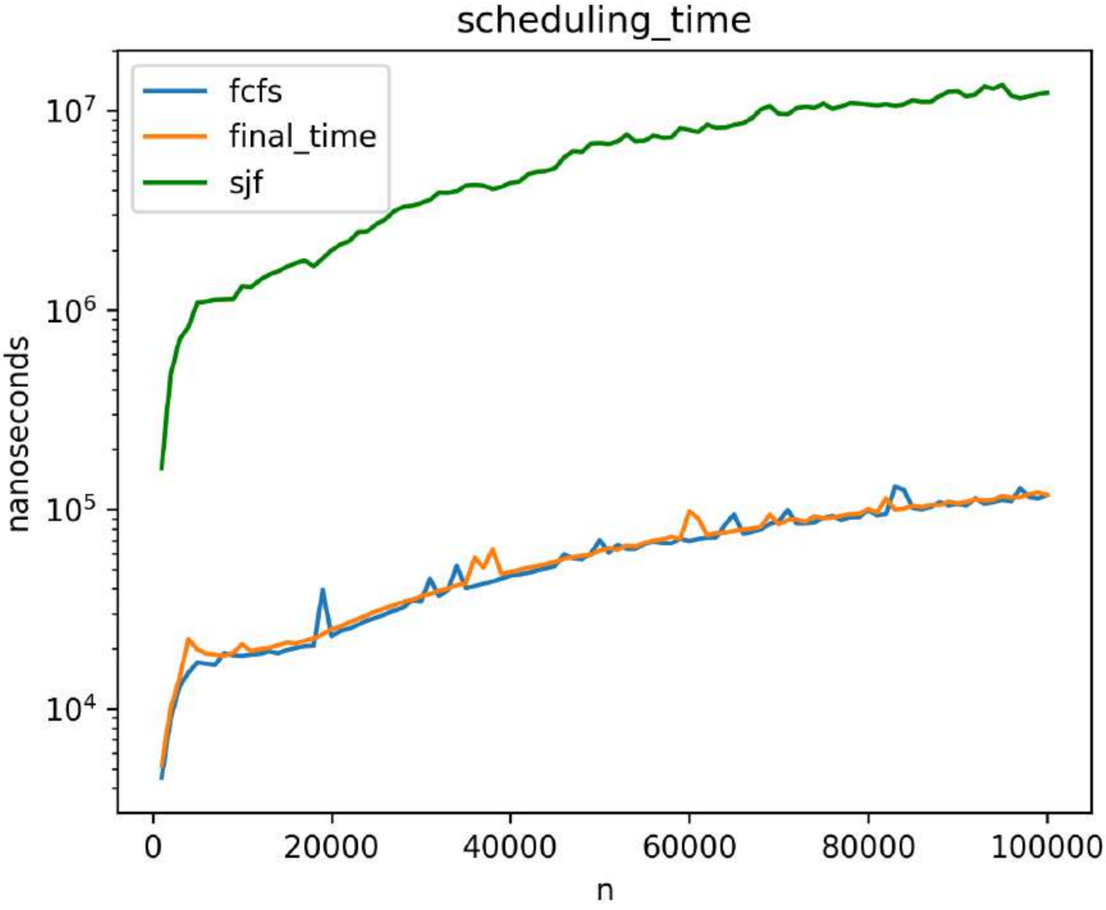


```
bool is_valid(float a[][COL], float b[], int n) {
  for (int i = 0; i < n; i++) {
    // cout << "test arival = " << b[0] << endl;
    // cout << "acctual arival = " << a[i][0] << endl;
    // cout << "test finish = " << b[1] << endl;
    // cout << "acctual finish = " << a[i][1] << endl;
    if ((b[0] >= a[i][0] && b[0] <= a[i][1]) ||
        (b[1] >= a[i][0] && b[1] <= a[i][1]) ||
        (b[0] <= a[i][0] && b[1] >= a[i][0]) ||
        (b[0] <= a[i][0] && b[1] >= a[i][1])) {
      // cout << "returned false" << endl;
      return false;
    }
  }
  // cout << "returned true" << endl;
  return true;
}

int main() {
  int arival_time_max = 86400;
  int arival_range = arival_time_max + 1;
  int weight_max = 100;
  int weigth_min = 10;
  int weight_range = weight_max - weigth_min + 1;
  float curr_time;
  cout << "size,c1,c2\n";
  for (int n = 1000; n <= 100000; n += 1000) {
    cout << n;
    float items[n][COL];
    srand(time(NULL));
    for (int i = 0; i < n; ++i) {
      // arival time
      items[i][0] = rand() % arival_range;
      // finish time
      items[i][1] =
          rand() % (int)(arival_range - items[i][0]) + (int)items[i][0];
      // duration
      items[i][2] = items[i][1] - items[i][0];
      items[i][3] = i + 1;
    }

    // mode = 0 is according to arival_time
    quicksort(items, 0, n - 1, 0);
    int c1 = 1;
    curr_time = items[0][1];
    auto start = std::chrono::high_resolution_clock::now();
    for (int i = 1; i < n; i++) {
      if (items[i][0] < curr_time) {
        continue;
      } else {
        c1++;
        curr_time = items[i][1];
      }
    }
    auto end = std::chrono::high_resolution_clock::now();
    auto duration =
        std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
    cout << ", " << duration.count();
```



```
    // mode = 1 is according to final_time
    quicksort(items, 0, n - 1, 1);
    int c2 = 1;
    curr_time = items[0][1];
    start = std::chrono::high_resolution_clock::now();
    for (int i = 1; i < n; i++) {
      if (items[i][0] < curr_time) {
        continue;
      } else {
        c2++;
        curr_time = items[i][1];
      }
    }
    end = std::chrono::high_resolution_clock::now();
    duration =
        std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
    cout << ", " << duration.count();

    // mode = 2 is according to duration
    quicksort(items, 0, n - 1, 2);
    float extras[n][COL];
    extras[0][0] = items[0][0];
    extras[0][1] = items[0][1];
    extras[0][2] = items[0][2];
    extras[0][3] = items[0][3];
    int c3 = 1;
    curr_time = items[0][1];
    start = std::chrono::high_resolution_clock::now();
    for (int i = 1; i < n; i++) {
      if (is_valid(extras, items[i], c3)) {
        extras[c3][0] = items[i][0];
        extras[c3][1] = items[i][1];
        extras[c3][2] = items[i][2];
        extras[c3][3] = items[i][3];
        c3++;
      }
    }
    end = std::chrono::high_resolution_clock::now();
    duration =
        std::chrono::duration_cast<std::chrono::nanoseconds>(end - start);
    cout << ", " << duration.count();

    cout << "\n";
  }
  return 0;
}
```


scheduling
fcfs
final_time
sjf
jobs
0
100
200
300
400
500
n
0
20000
40000
60000
80000
100000


scheduling_time
fcfs
final_time
sjf
nanoseconds
10^4
10^5
10^6
10^7
n
0
20000
40000
60000
80000
100000

Matrix Multiplication - Divide and Conquer

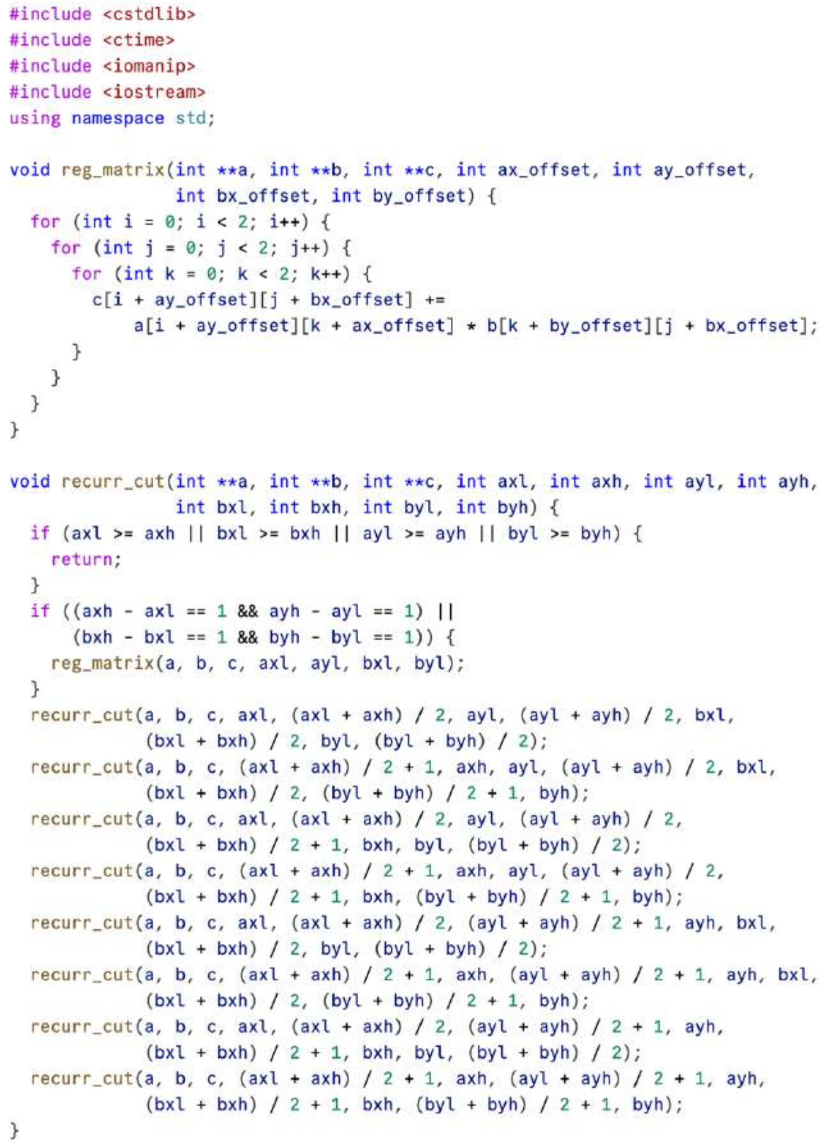
Idea –

- Using divide and conquer to break the matrix into smaller parts and multiplying them to get the resulting matrix
- Each matrix is divided into 4 equal parts by cutting it at its central axis
- After we get a down to a 2x2 matrix we solve it using conventional method
- Works best for a square matrix of size in 2^n. If not it 2^n we add padding to it

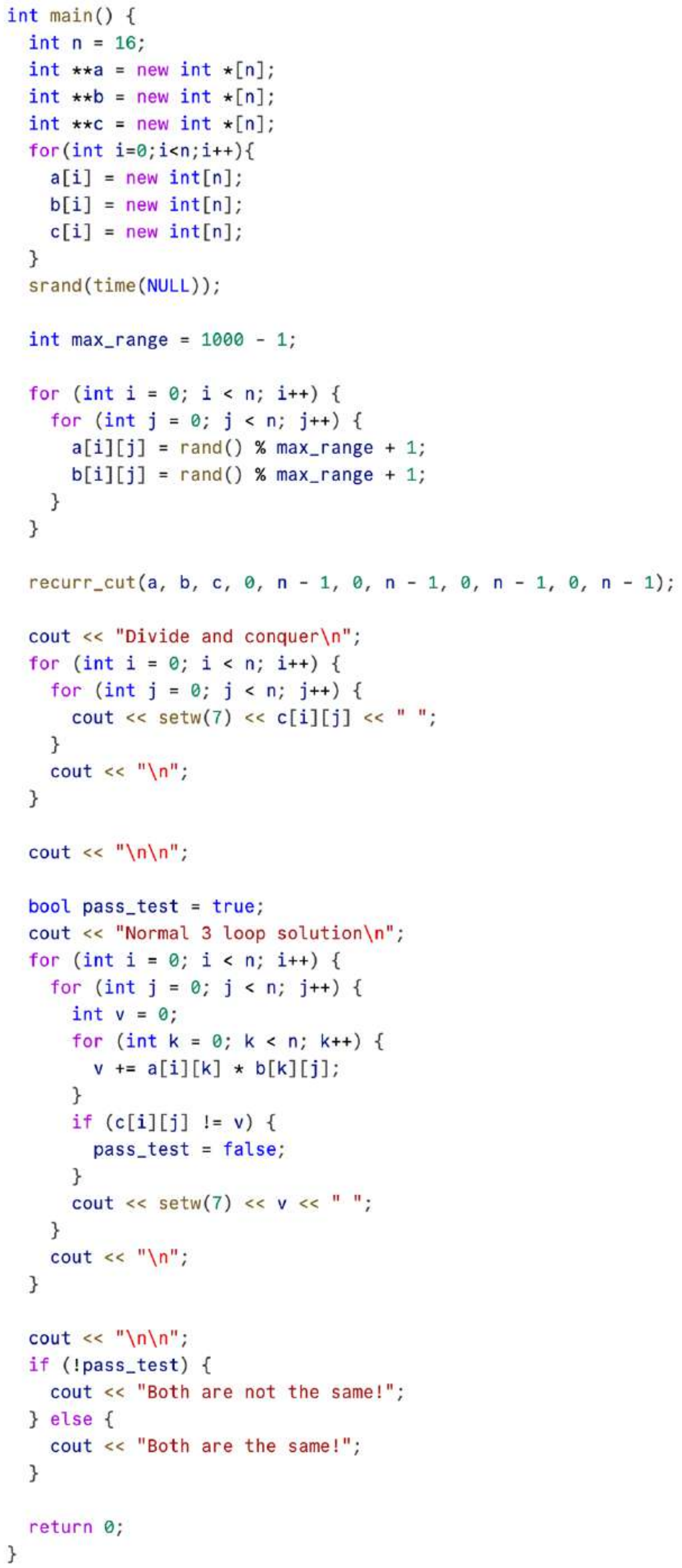
Complexity –

- Time – O(n^3)
- Space – O(n^2)

Code –



```
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <iostream>
using namespace std;

void reg_matrix(int **a, int **b, int **c, int ax_offset, int ay_offset,
                int bx_offset, int by_offset) {
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
      for (int k = 0; k < 2; k++) {
        c[i + ay_offset][j + bx_offset] +=
            a[i + ay_offset][k + ax_offset] * b[k + by_offset][j + bx_offset];
      }
    }
  }
}

void recurr_cut(int **a, int **b, int **c, int axl, int axh, int ayl, int ayh,
                int bxl, int bxh, int byl, int byh) {
  if (axl >= axh || bxl >= bxh || ayl >= ayh || byl >= byh) {
    return;
  }
  if ((axh - axl == 1 && ayh - ayl == 1) ||
      (bxh - bxl == 1 && byh - byl == 1)) {
    reg_matrix(a, b, c, axl, ayl, bxl, byl);
  }
  recurr_cut(a, b, c, axl, (axl + axh) / 2, ayl, (ayl + ayh) / 2, bxl,
             (bxl + bxh) / 2, byl, (byl + byh) / 2);
  recurr_cut(a, b, c, (axl + axh) / 2 + 1, axh, ayl, (ayl + ayh) / 2, bxl,
             (bxl + bxh) / 2, (byl + byh) / 2 + 1, byh);
  recurr_cut(a, b, c, axl, (axl + axh) / 2, ayl, (ayl + ayh) / 2,
             (bxl + bxh) / 2 + 1, bxh, byl, (byl + byh) / 2);
  recurr_cut(a, b, c, (axl + axh) / 2 + 1, axh, ayl, (ayl + ayh) / 2,
             (bxl + bxh) / 2 + 1, bxh, (byl + byh) / 2 + 1, byh);
  recurr_cut(a, b, c, axl, (axl + axh) / 2, (ayl + ayh) / 2 + 1, ayh, bxl,
             (bxl + bxh) / 2, byl, (byl + byh) / 2);
  recurr_cut(a, b, c, (axl + axh) / 2 + 1, axh, (ayl + ayh) / 2 + 1, ayh, bxl,
             (bxl + bxh) / 2, (byl + byh) / 2 + 1, byh);
  recurr_cut(a, b, c, axl, (axl + axh) / 2, (ayl + ayh) / 2 + 1, ayh,
             (bxl + bxh) / 2 + 1, bxh, byl, (byl + byh) / 2);
  recurr_cut(a, b, c, (axl + axh) / 2 + 1, axh, (ayl + ayh) / 2 + 1, ayh,
             (bxl + bxh) / 2 + 1, bxh, (byl + byh) / 2 + 1, byh);
}
```



```
int main() {
  int n = 16;
  int **a = new int *[n];
  int **b = new int *[n];
  int **c = new int *[n];
  for(int i=0;i<n;i++){
    a[i] = new int[n];
    b[i] = new int[n];
    c[i] = new int[n];
  }
  srand(time(NULL));

  int max_range = 1000 - 1;

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      a[i][j] = rand() % max_range + 1;
      b[i][j] = rand() % max_range + 1;
    }
  }

  recurr_cut(a, b, c, 0, n - 1, 0, n - 1, 0, n - 1, 0, n - 1);

  cout << "Divide and conquer\n";
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      cout << setw(7) << c[i][j] << " ";
    }
    cout << "\n";
  }

  cout << "\n\n";

  bool pass_test = true;
  cout << "Normal 3 loop solution\n";
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      int v = 0;
      for (int k = 0; k < n; k++) {
        v += a[i][k] * b[k][j];
      }
      if (c[i][j] != v) {
        pass_test = false;
      }
      cout << setw(7) << v << " ";
    }
    cout << "\n";
  }

  cout << "\n\n";
  if (!pass_test) {
    cout << "Both are not the same!";
  } else {
    cout << "Both are the same!";
  }

  return 0;
}
```

Matrix Multiplication – Strassens Algorithm

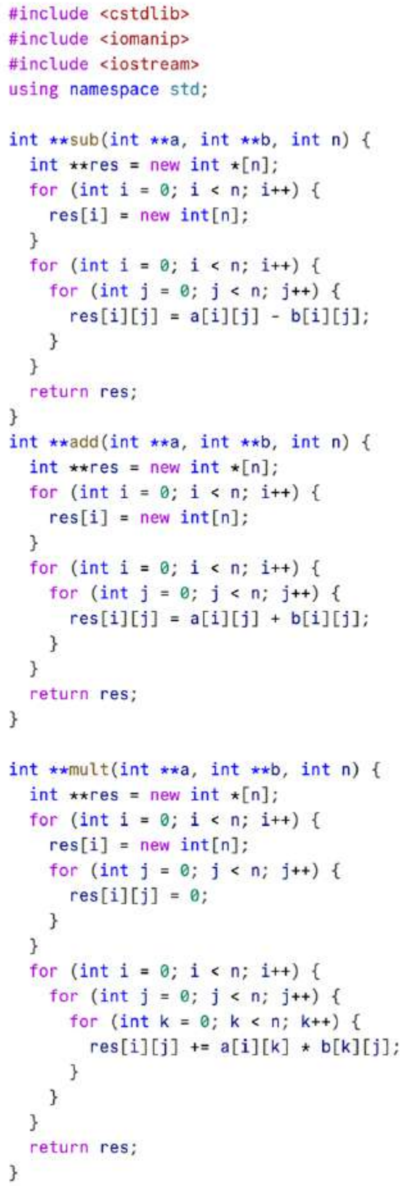
Idea –

- Gives a specific formula to turn the O(n^3) complexity into a O(n^2.81) by dividing the matrix into only 7 parts instead of the 8 that happen in the normal divide and conquer method

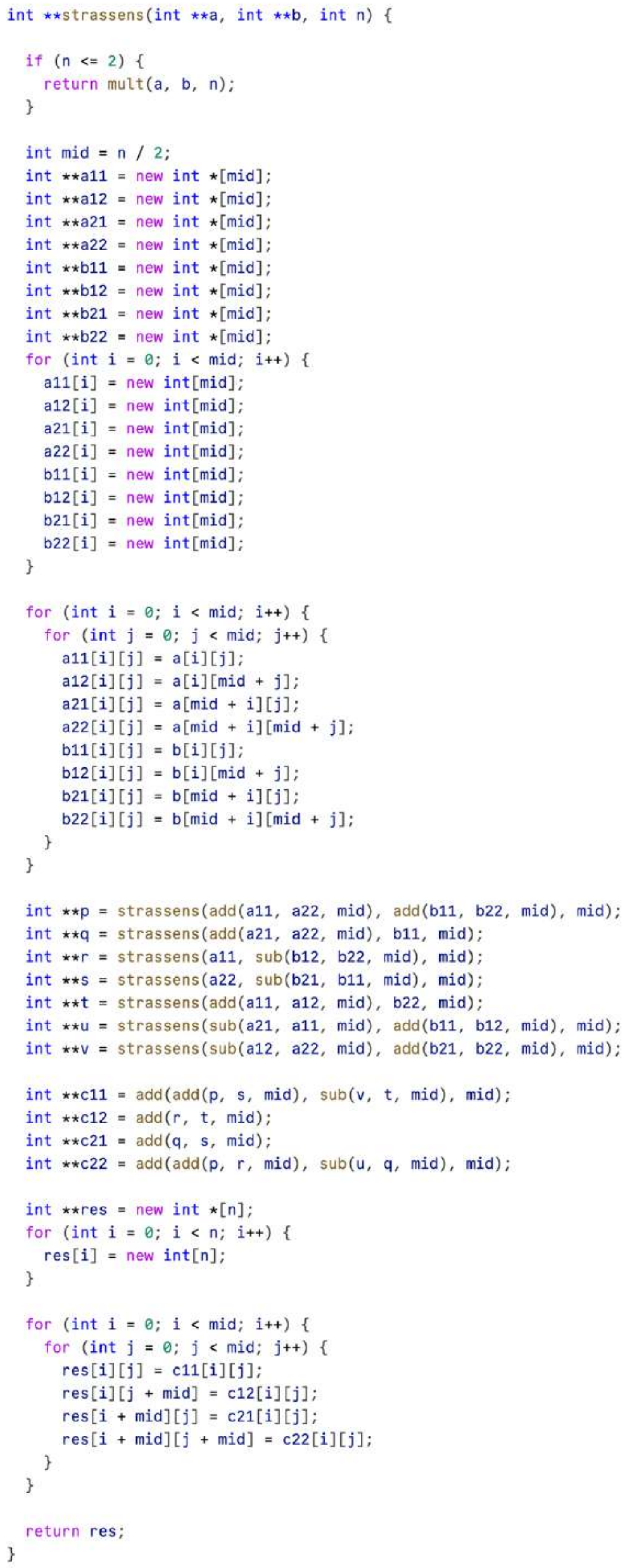
Complexity –

- Time – O(n^2.81)
- Space – O(n^2)

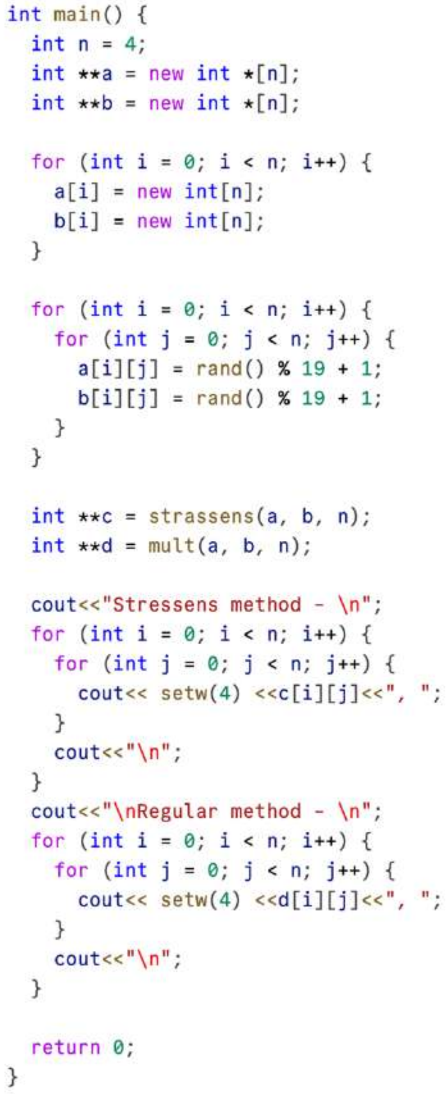
Code –



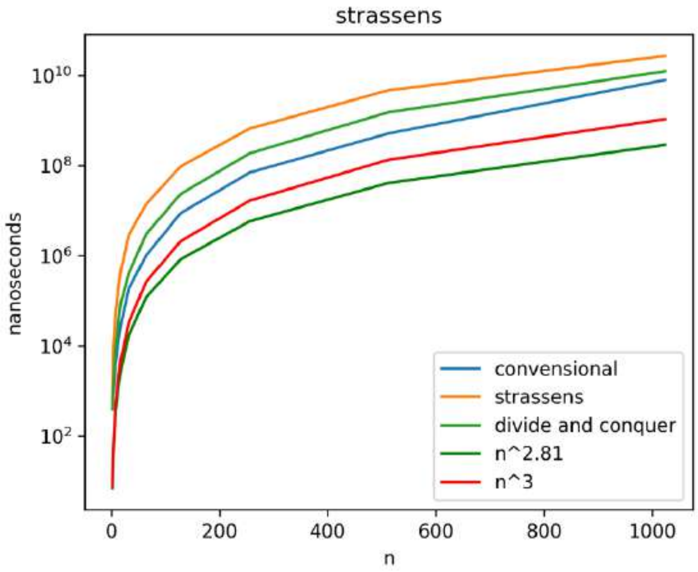
```
#include <cstdlib>
#include <iomanip>
#include <iostream>
using namespace std;

int **sub(int **a, int **b, int n) {
  int **res = new int *[n];
  for (int i = 0; i < n; i++) {
    res[i] = new int[n];
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      res[i][j] = a[i][j] - b[i][j];
    }
  }
  return res;
}
int **add(int **a, int **b, int n) {
  int **res = new int *[n];
  for (int i = 0; i < n; i++) {
    res[i] = new int[n];
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      res[i][j] = a[i][j] + b[i][j];
    }
  }
  return res;
}

int **mult(int **a, int **b, int n) {
  int **res = new int *[n];
  for (int i = 0; i < n; i++) {
    res[i] = new int[n];
    for (int j = 0; j < n; j++) {
      res[i][j] = 0;
    }
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
        res[i][j] += a[i][k] * b[k][j];
      }
    }
  }
  return res;
}
```



```
int **strassens(int **a, int **b, int n) {

  if (n <= 2) {
    return mult(a, b, n);
  }

  int mid = n / 2;
  int **a11 = new int *[mid];
  int **a12 = new int *[mid];
  int **a21 = new int *[mid];
  int **a22 = new int *[mid];
  int **b11 = new int *[mid];
  int **b12 = new int *[mid];
  int **b21 = new int *[mid];
  int **b22 = new int *[mid];
  for (int i = 0; i < mid; i++) {
    a11[i] = new int[mid];
    a12[i] = new int[mid];
    a21[i] = new int[mid];
    a22[i] = new int[mid];
    b11[i] = new int[mid];
    b12[i] = new int[mid];
    b21[i] = new int[mid];
    b22[i] = new int[mid];
  }

  for (int i = 0; i < mid; i++) {
    for (int j = 0; j < mid; j++) {
      a11[i][j] = a[i][j];
      a12[i][j] = a[i][mid + j];
      a21[i][j] = a[mid + i][j];
      a22[i][j] = a[mid + i][mid + j];
      b11[i][j] = b[i][j];
      b12[i][j] = b[i][mid + j];
      b21[i][j] = b[mid + i][j];
      b22[i][j] = b[mid + i][mid + j];
    }
  }

  int **p = strassens(add(a11, a22, mid), add(b11, b22, mid), mid);
  int **q = strassens(add(a21, a22, mid), b11, mid);
  int **r = strassens(a11, sub(b12, b22, mid), mid);
  int **s = strassens(a22, sub(b21, b11, mid), mid);
  int **t = strassens(add(a11, a12, mid), b22, mid);
  int **u = strassens(sub(a21, a11, mid), add(b11, b12, mid), mid);
  int **v = strassens(sub(a12, a22, mid), add(b21, b22, mid), mid);

  int **c11 = add(add(p, s, mid), sub(v, t, mid), mid);
  int **c12 = add(r, t, mid);
  int **c21 = add(q, s, mid);
  int **c22 = add(add(p, r, mid), sub(u, q, mid), mid);

  int **res = new int *[n];
  for (int i = 0; i < n; i++) {
    res[i] = new int[n];
  }

  for (int i = 0; i < mid; i++) {
    for (int j = 0; j < mid; j++) {
      res[i][j] = c11[i][j];
      res[i][j + mid] = c12[i][j];
      res[i + mid][j] = c21[i][j];
      res[i + mid][j + mid] = c22[i][j];
    }
  }

  return res;
}
```



```
int main() {
  int n = 4;
  int **a = new int *[n];
  int **b = new int *[n];

  for (int i = 0; i < n; i++) {
    a[i] = new int[n];
    b[i] = new int[n];
  }

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      a[i][j] = rand() % 19 + 1;
      b[i][j] = rand() % 19 + 1;
    }
  }

  int **c = strassens(a, b, n);
  int **d = mult(a, b, n);

  cout<<"Stressens method - \n";
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      cout<< setw(4) <<c[i][j]<<", ";
    }
    cout<<"\n";
  }
  cout<<"\nRegular method - \n";
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      cout<< setw(4) <<d[i][j]<<", ";
    }
    cout<<"\n";
  }

  return 0;
}
```



Quickhull

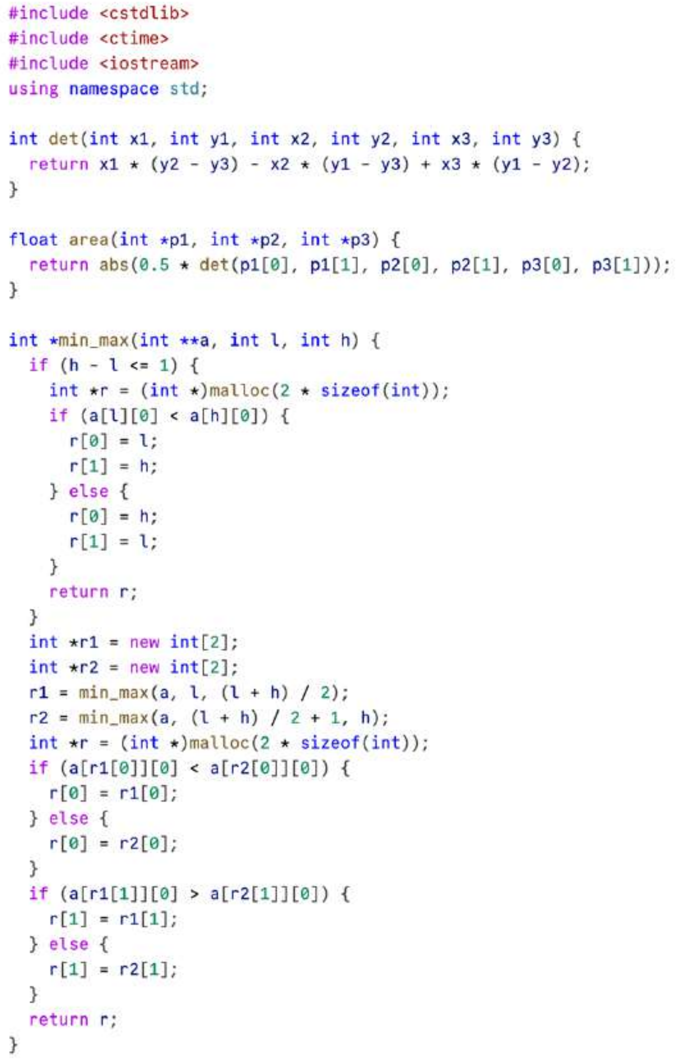
Idea –

- Find out a Convex polygon from a given set of points
- Uses a approach similar to quicksort by recursively dividing the problem into smaller parts
- Cuts the points across the min and max point on the X-axis and then find out a point which makes the largest triangle(area) with the line joining the min max points on the X-axis
- Using the 2 new sides of the triangle keep on recursively calling the algorithm till we have all the points inside the triangles

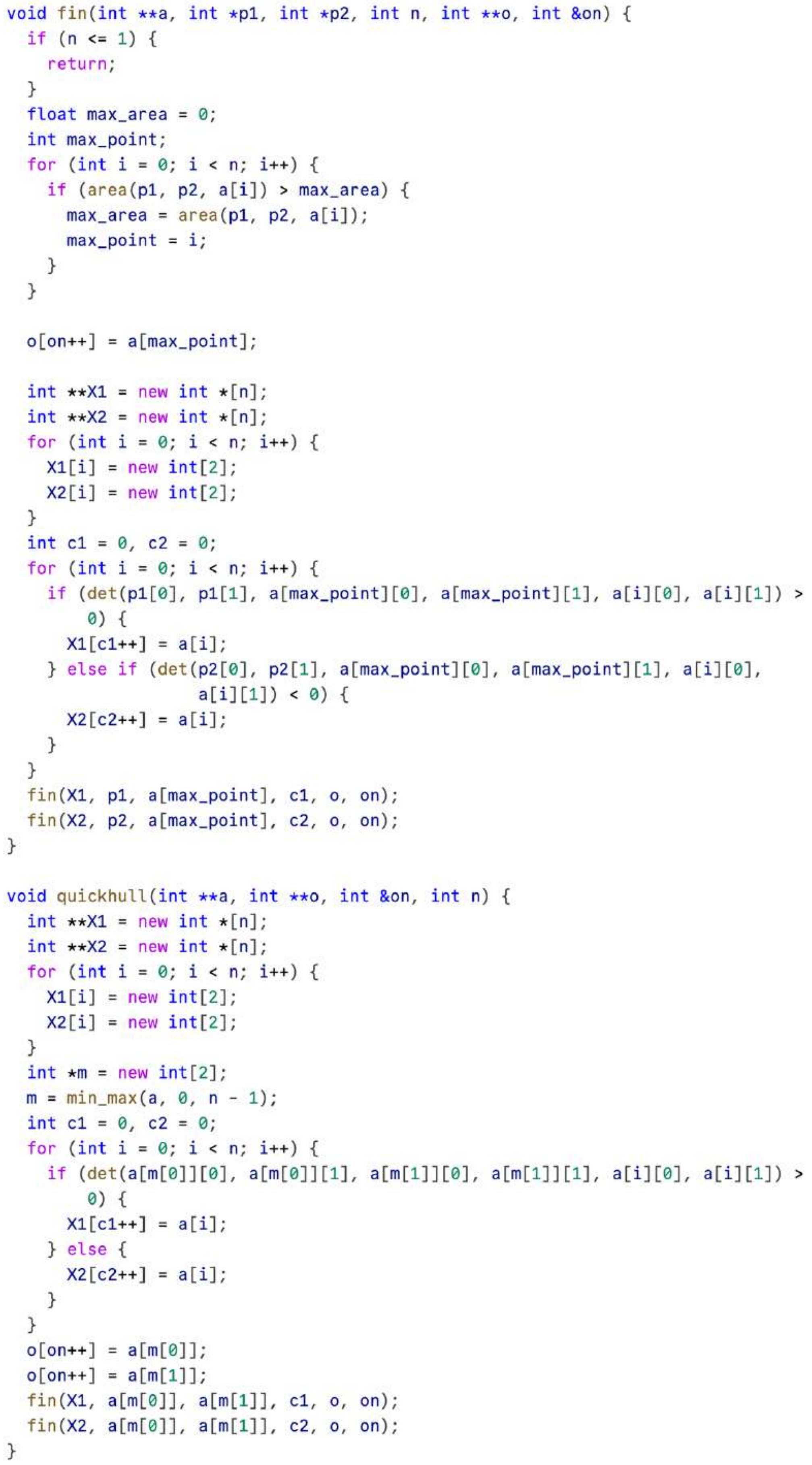
Complexity –

- Time – O(nlogn)
- Space – O(n)

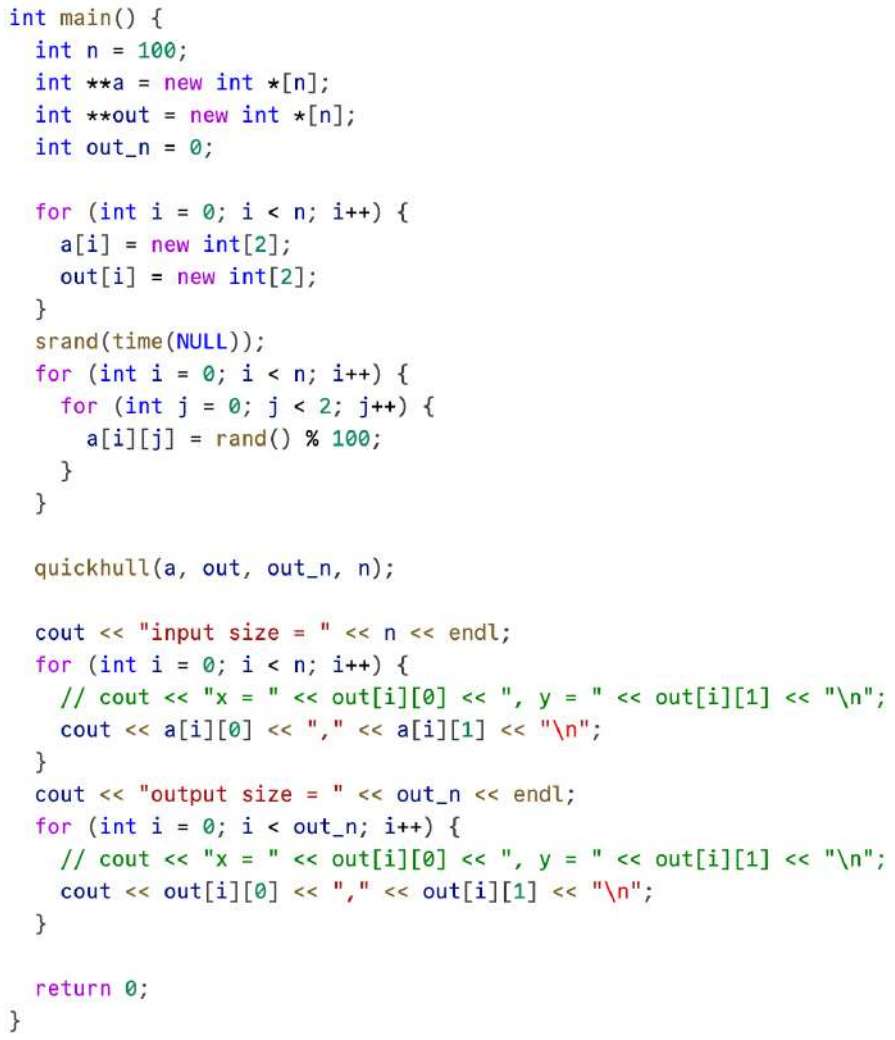
Code –



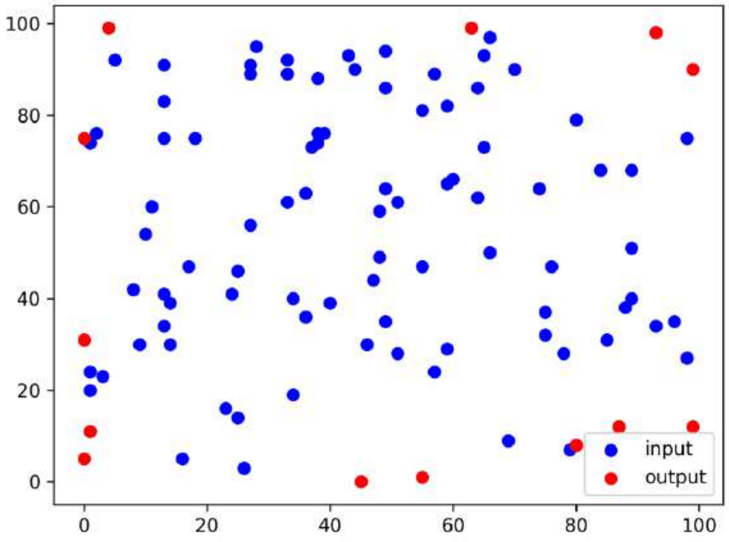
```
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

int det(int x1, int y1, int x2, int y2, int x3, int y3) {
  return x1 * (y2 - y3) - x2 * (y1 - y3) + x3 * (y1 - y2);
}

float area(int *p1, int *p2, int *p3) {
  return abs(0.5 * det(p1[0], p1[1], p2[0], p2[1], p3[0], p3[1]));
}

int *min_max(int **a, int l, int h) {
  if (h - l <= 1) {
    int *r = (int *)malloc(2 * sizeof(int));
    if (a[l][0] < a[h][0]) {
      r[0] = l;
      r[1] = h;
    } else {
      r[0] = h;
      r[1] = l;
    }
    return r;
  }
  int *r1 = new int[2];
  int *r2 = new int[2];
  r1 = min_max(a, l, (l + h) / 2);
  r2 = min_max(a, (l + h) / 2 + 1, h);
  int *r = (int *)malloc(2 * sizeof(int));
  if (a[r1[0]][0] < a[r2[0]][0]) {
    r[0] = r1[0];
  } else {
    r[0] = r2[0];
  }
  if (a[r1[1]][0] > a[r2[1]][0]) {
    r[1] = r1[1];
  } else {
    r[1] = r2[1];
  }
  return r;
}
```



```
void fin(int **a, int *p1, int *p2, int n, int **o, int &on) {
  if (n <= 1) {
    return;
  }
  float max_area = 0;
  int max_point;
  for (int i = 0; i < n; i++) {
    if (area(p1, p2, a[i]) > max_area) {
      max_area = area(p1, p2, a[i]);
      max_point = i;
    }
  }

  o[on++] = a[max_point];

  int **X1 = new int *[n];
  int **X2 = new int *[n];
  for (int i = 0; i < n; i++) {
    X1[i] = new int[2];
    X2[i] = new int[2];
  }
  int c1 = 0, c2 = 0;
  for (int i = 0; i < n; i++) {
    if (det(p1[0], p1[1], a[max_point][0], a[max_point][1], a[i][0], a[i][1]) >
        0) {
      X1[c1++] = a[i];
    } else if (det(p2[0], p2[1], a[max_point][0], a[max_point][1], a[i][0],
                   a[i][1]) < 0) {
      X2[c2++] = a[i];
    }
  }
  fin(X1, p1, a[max_point], c1, o, on);
  fin(X2, p2, a[max_point], c2, o, on);
}

void quickhull(int **a, int **o, int &on, int n) {
  int **X1 = new int *[n];
  int **X2 = new int *[n];
  for (int i = 0; i < n; i++) {
    X1[i] = new int[2];
    X2[i] = new int[2];
  }
  int *m = new int[2];
  m = min_max(a, 0, n - 1);
  int c1 = 0, c2 = 0;
  for (int i = 0; i < n; i++) {
    if (det(a[m[0]][0], a[m[0]][1], a[m[1]][0], a[m[1]][1], a[i][0], a[i][1]) >
        0) {
      X1[c1++] = a[i];
    } else {
      X2[c2++] = a[i];
    }
  }
  o[on++] = a[m[0]];
  o[on++] = a[m[1]];
  fin(X1, a[m[0]], a[m[1]], c1, o, on);
  fin(X2, a[m[0]], a[m[1]], c2, o, on);
}
```



```
int main() {
  int n = 100;
  int **a = new int *[n];
  int **out = new int *[n];
  int out_n = 0;

  for (int i = 0; i < n; i++) {
    a[i] = new int[2];
    out[i] = new int[2];
  }
  srand(time(NULL));
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < 2; j++) {
      a[i][j] = rand() % 100;
    }
  }

  quickhull(a, out, out_n, n);

  cout << "input size = " << n << endl;
  for (int i = 0; i < n; i++) {
    // cout << "x = " << out[i][0] << ", y = " << out[i][1] << "\n";
    cout << a[i][0] << "," << a[i][1] << "\n";
  }
  cout << "output size = " << out_n << endl;
  for (int i = 0; i < out_n; i++) {
    // cout << "x = " << out[i][0] << ", y = " << out[i][1] << "\n";
    cout << out[i][0] << "," << out[i][1] << "\n";
  }

  return 0;
}
```



Dijkastra

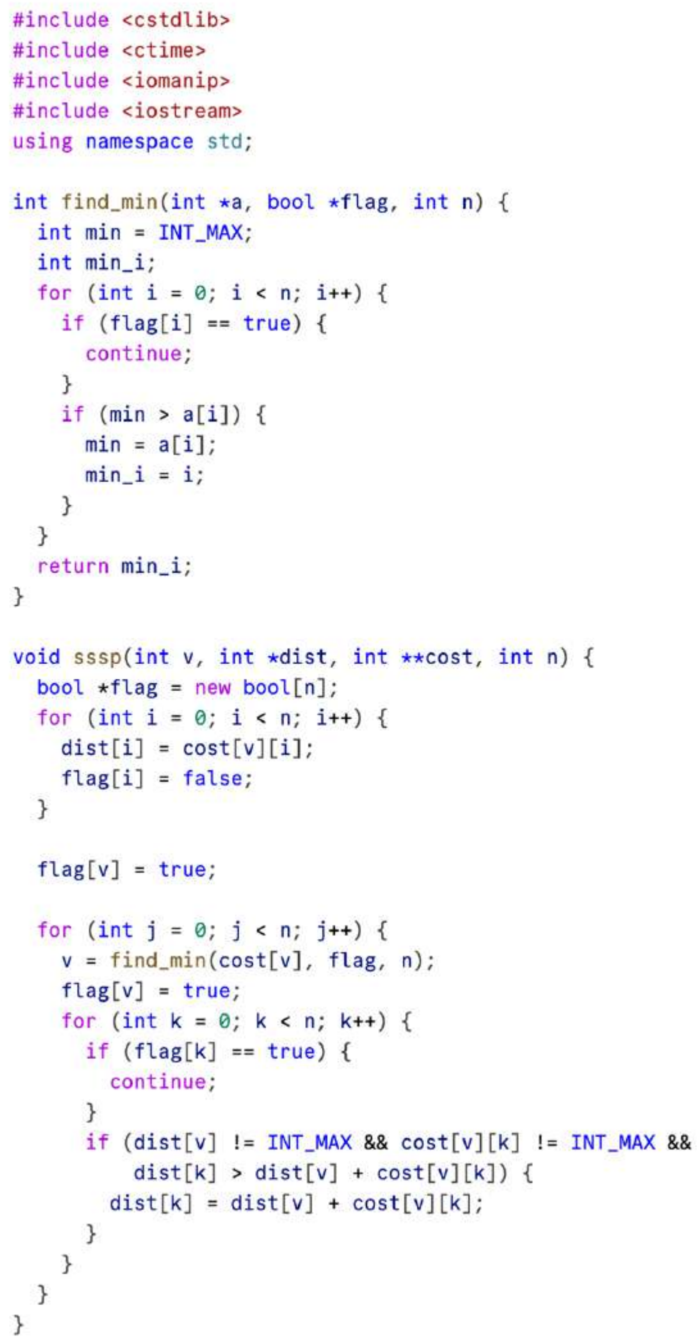
Idea –

- Find the shortest path from source to destination
- Creates an array containing all the distances from source to all of the vertexes

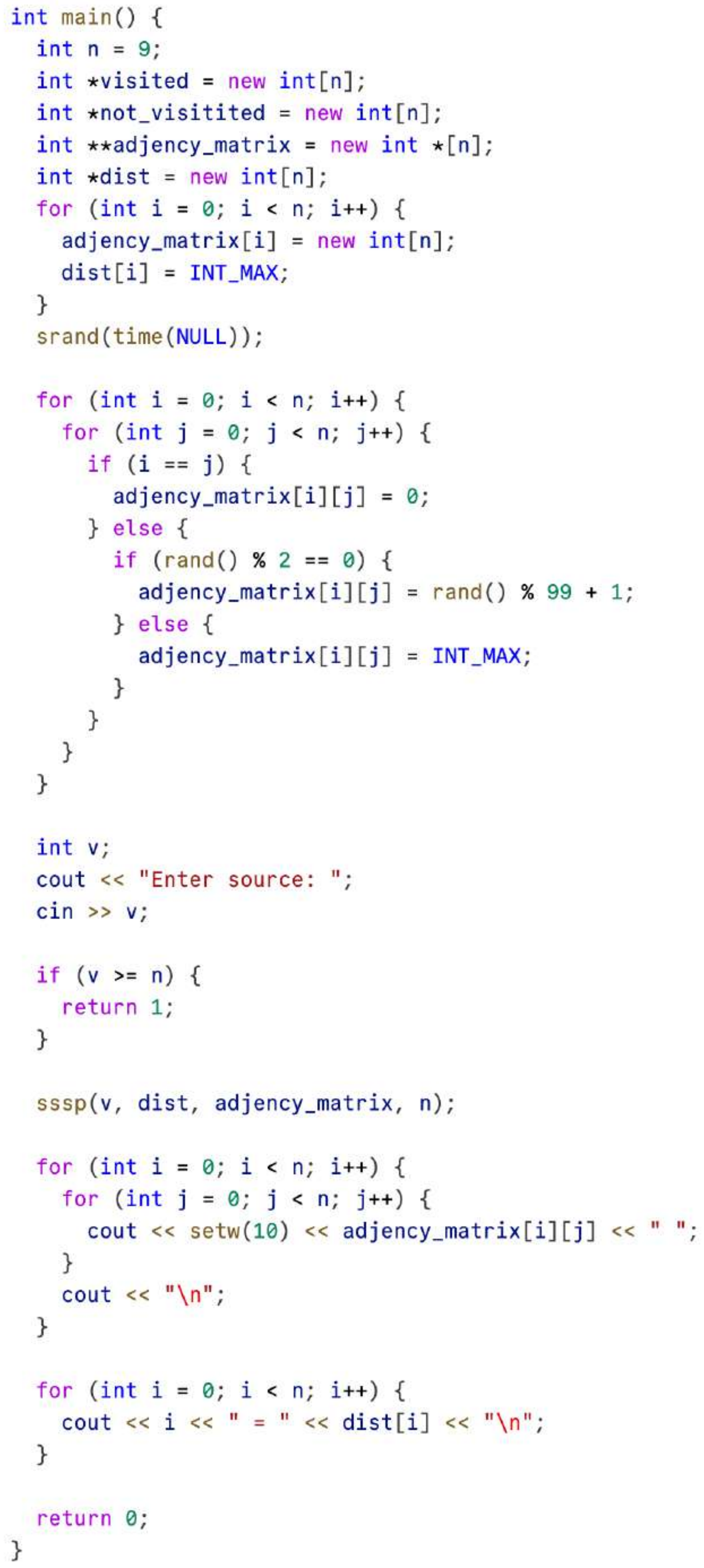
Complexity –

- Time – O(n^2)
- Space – O(n)

Code –



```
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <iostream>
using namespace std;

int find_min(int *a, bool *flag, int n) {
  int min = INT_MAX;
  int min_i;
  for (int i = 0; i < n; i++) {
    if (flag[i] == true) {
      continue;
    }
    if (min > a[i]) {
      min = a[i];
      min_i = i;
    }
  }
  return min_i;
}

void sssp(int v, int *dist, int **cost, int n) {
  bool *flag = new bool[n];
  for (int i = 0; i < n; i++) {
    dist[i] = cost[v][i];
    flag[i] = false;
  }

  flag[v] = true;

  for (int j = 0; j < n; j++) {
    v = find_min(cost[v], flag, n);
    flag[v] = true;
    for (int k = 0; k < n; k++) {
      if (flag[k] == true) {
        continue;
      }
      if (dist[v] != INT_MAX && cost[v][k] != INT_MAX &&
          dist[k] > dist[v] + cost[v][k]) {
        dist[k] = dist[v] + cost[v][k];
      }
    }
  }
}
```



```
int main() {
  int n = 9;
  int *visited = new int[n];
  int *not_visitited = new int[n];
  int **adjency_matrix = new int *[n];
  int *dist = new int[n];
  for (int i = 0; i < n; i++) {
    adjency_matrix[i] = new int[n];
    dist[i] = INT_MAX;
  }
  srand(time(NULL));

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (i == j) {
        adjency_matrix[i][j] = 0;
      } else {
        if (rand() % 2 == 0) {
          adjency_matrix[i][j] = rand() % 99 + 1;
        } else {
          adjency_matrix[i][j] = INT_MAX;
        }
      }
    }
  }

  int v;
  cout << "Enter source: ";
  cin >> v;

  if (v >= n) {
    return 1;
  }

  sssp(v, dist, adjency_matrix, n);

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      cout << setw(10) << adjency_matrix[i][j] << " ";
    }
    cout << "\n";
  }

  for (int i = 0; i < n; i++) {
    cout << i << " = " << dist[i] << "\n";
  }

  return 0;
}
```

Kruskals

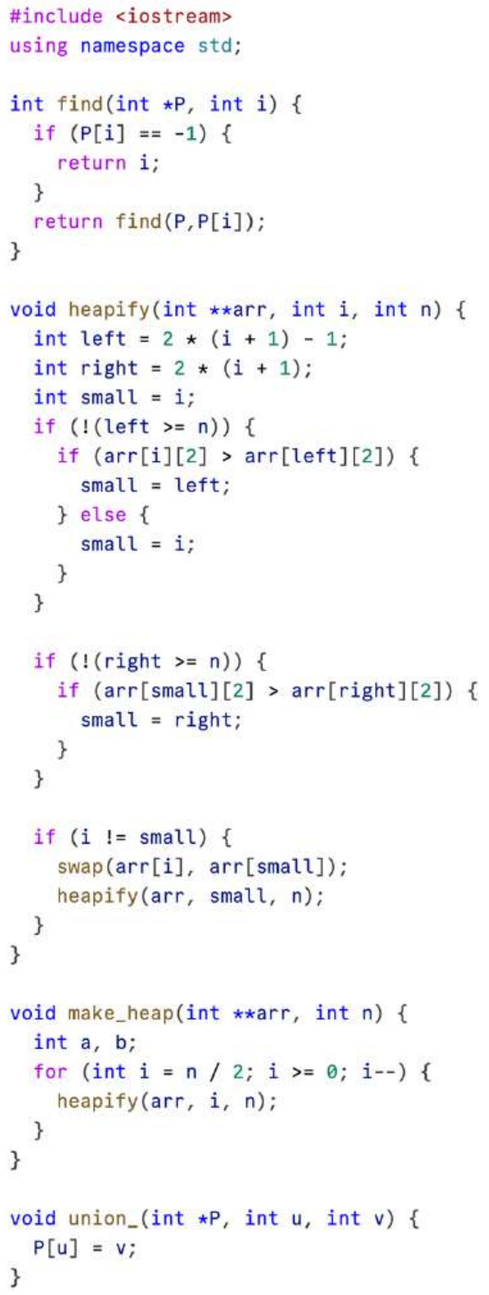
Idea –

- Creates a minimum cost spanning tree
- Picks out a minimum edge and checks if there parents are the same
    - If we get same parents we don’t use that edge as it causes a loop
    - Otherwise we use that edge in our tree

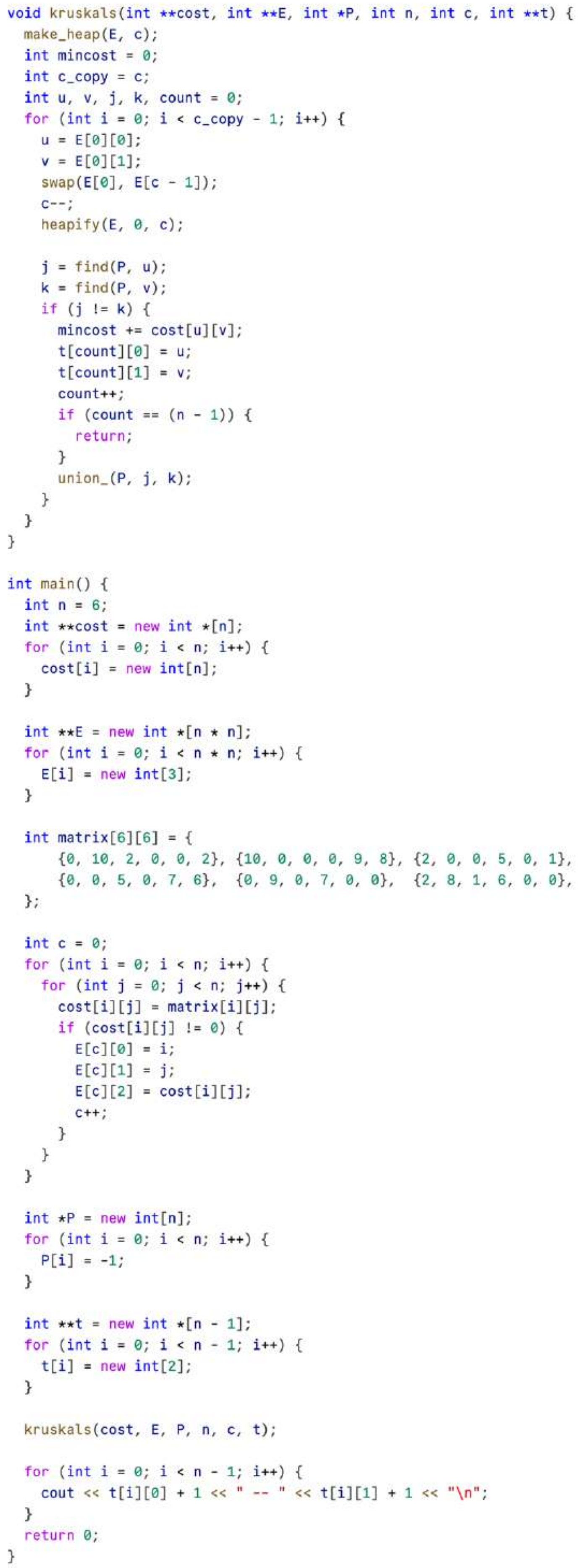
Complexity –

- Time – O(ElogE)
- Space – O(E+V)

Code –



```
#include <iostream>
using namespace std;

int find(int *P, int i) {
  if (P[i] == -1) {
    return i;
  }
  return find(P,P[i]);
}

void heapify(int **arr, int i, int n) {
  int left = 2 * (i + 1) - 1;
  int right = 2 * (i + 1);
  int small = i;
  if (!(left >= n)) {
    if (arr[i][2] > arr[left][2]) {
      small = left;
    } else {
      small = i;
    }
  }

  if (!(right >= n)) {
    if (arr[small][2] > arr[right][2]) {
      small = right;
    }
  }

  if (i != small) {
    swap(arr[i], arr[small]);
    heapify(arr, small, n);
  }
}

void make_heap(int **arr, int n) {
  int a, b;
  for (int i = n / 2; i >= 0; i--) {
    heapify(arr, i, n);
  }
}

void union_(int *P, int u, int v) {
  P[u] = v;
}
```



```
void kruskals(int **cost, int **E, int *P, int n, int c, int **t) {
  make_heap(E, c);
  int mincost = 0;
  int c_copy = c;
  int u, v, j, k, count = 0;
  for (int i = 0; i < c_copy - 1; i++) {
    u = E[0][0];
    v = E[0][1];
    swap(E[0], E[c - 1]);
    c--;
    heapify(E, 0, c);

    j = find(P, u);
    k = find(P, v);
    if (j != k) {
      mincost += cost[u][v];
      t[count][0] = u;
      t[count][1] = v;
      count++;
      if (count == (n - 1)) {
        return;
      }
      union_(P, j, k);
    }
  }
}

int main() {
  int n = 6;
  int **cost = new int *[n];
  for (int i = 0; i < n; i++) {
    cost[i] = new int[n];
  }

  int **E = new int *[n * n];
  for (int i = 0; i < n * n; i++) {
    E[i] = new int[3];
  }

  int matrix[6][6] = {
      {0, 10, 2, 0, 0, 2}, {10, 0, 0, 0, 9, 8}, {2, 0, 0, 5, 0, 1},
      {0, 0, 5, 0, 7, 6},  {0, 9, 0, 7, 0, 0},  {2, 8, 1, 6, 0, 0},
  };

  int c = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      cost[i][j] = matrix[i][j];
      if (cost[i][j] != 0) {
        E[c][0] = i;
        E[c][1] = j;
        E[c][2] = cost[i][j];
        c++;
      }
    }
  }

  int *P = new int[n];
  for (int i = 0; i < n; i++) {
    P[i] = -1;
  }

  int **t = new int *[n - 1];
  for (int i = 0; i < n - 1; i++) {
    t[i] = new int[2];
  }

  kruskals(cost, E, P, n, c, t);

  for (int i = 0; i < n - 1; i++) {
    cout << t[i][0] + 1 << " -- " << t[i][1] + 1 << "\n";
  }
  return 0;
}
```

Prims

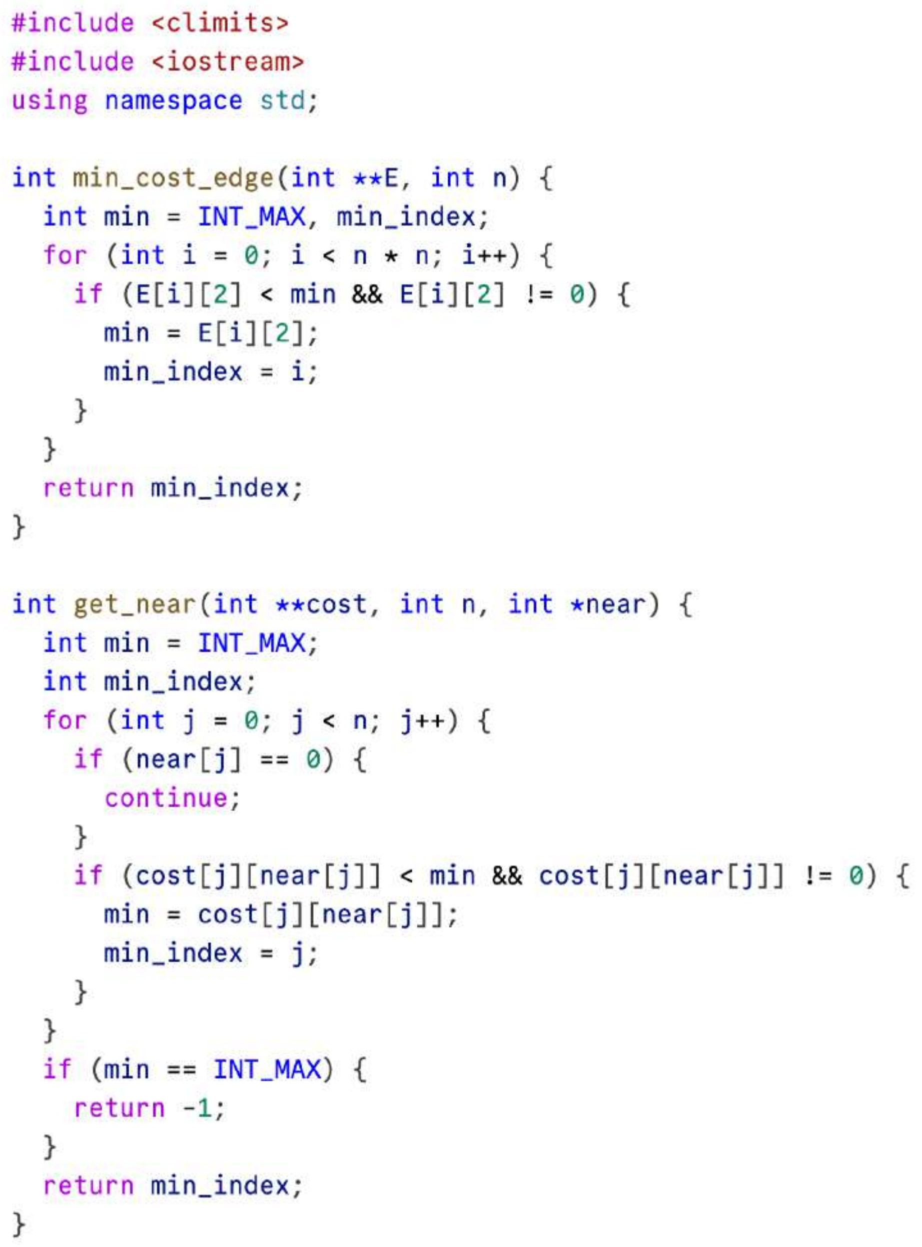
Idea –

- Creates a minimum cost spanning tree
- Finds one minimum edge using the merge sort technique and then find the shortest edge from the 2 selected vertex’s
- keeps on adding the shortest edges that extend from our visited vertex’s that don’t go to our visited vertexs

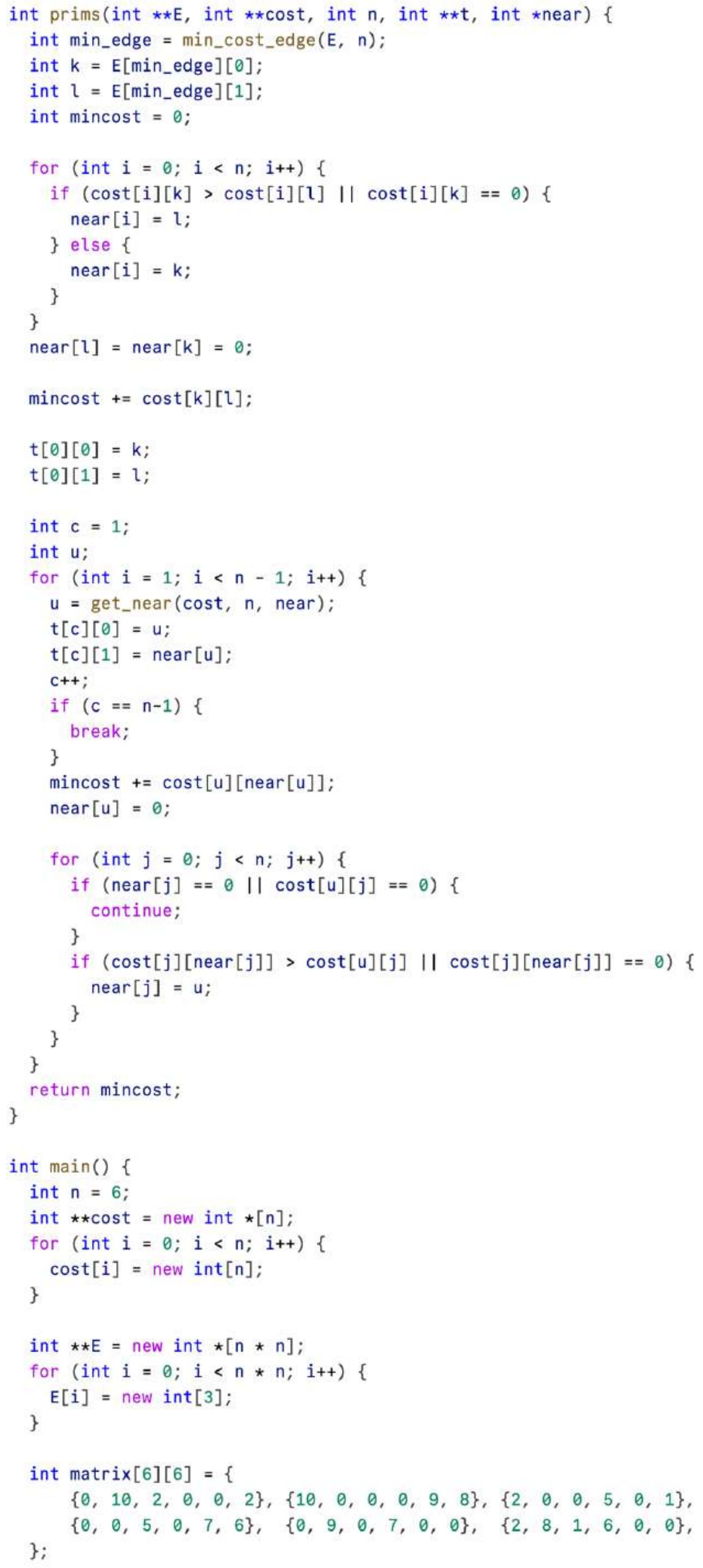
Complexity –

- Time – O(n^2)
- Space – O(n)

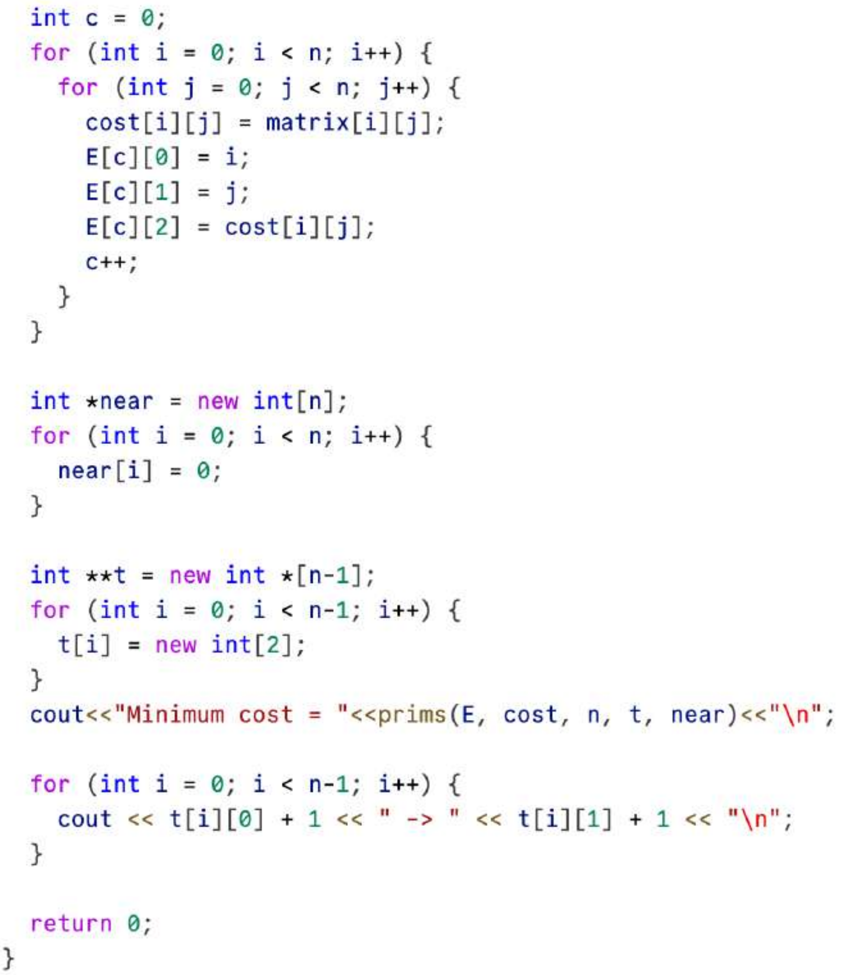
Code –



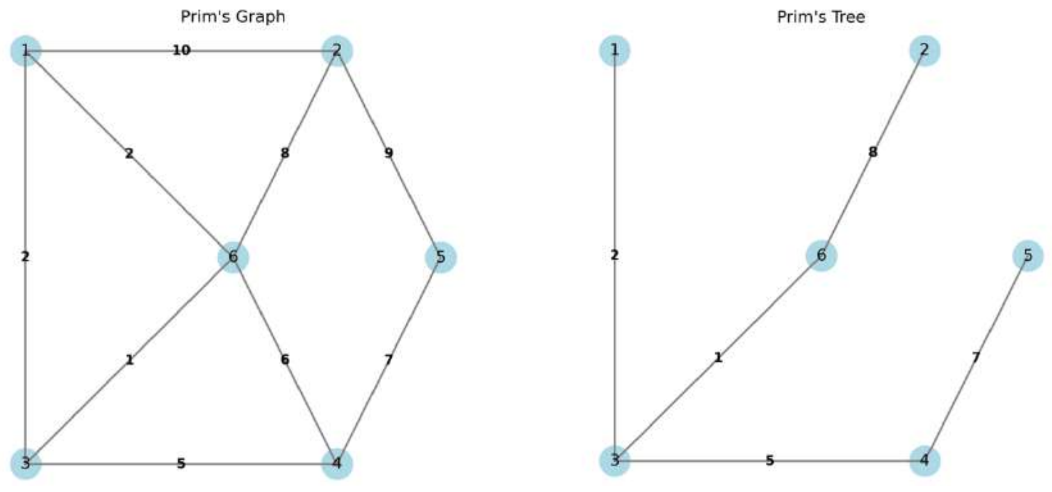
```
#include <climits>
#include <iostream>
using namespace std;

int min_cost_edge(int **E, int n) {
  int min = INT_MAX, min_index;
  for (int i = 0; i < n * n; i++) {
    if (E[i][2] < min && E[i][2] != 0) {
      min = E[i][2];
      min_index = i;
    }
  }
  return min_index;
}

int get_near(int **cost, int n, int *near) {
  int min = INT_MAX;
  int min_index;
  for (int j = 0; j < n; j++) {
    if (near[j] == 0) {
      continue;
    }
    if (cost[j][near[j]] < min && cost[j][near[j]] != 0) {
      min = cost[j][near[j]];
      min_index = j;
    }
  }
  if (min == INT_MAX) {
    return -1;
  }
  return min_index;
}
```



```
int prims(int **E, int **cost, int n, int **t, int *near) {
  int min_edge = min_cost_edge(E, n);
  int k = E[min_edge][0];
  int l = E[min_edge][1];
  int mincost = 0;

  for (int i = 0; i < n; i++) {
    if (cost[i][k] > cost[i][l] || cost[i][k] == 0) {
      near[i] = l;
    } else {
      near[i] = k;
    }
  }
  near[l] = near[k] = 0;

  mincost += cost[k][l];

  t[0][0] = k;
  t[0][1] = l;

  int c = 1;
  int u;
  for (int i = 1; i < n - 1; i++) {
    u = get_near(cost, n, near);
    t[c][0] = u;
    t[c][1] = near[u];
    c++;
    if (c == n-1) {
      break;
    }
    mincost += cost[u][near[u]];
    near[u] = 0;

    for (int j = 0; j < n; j++) {
      if (near[j] == 0 || cost[u][j] == 0) {
        continue;
      }
      if (cost[j][near[j]] > cost[u][j] || cost[j][near[j]] == 0) {
        near[j] = u;
      }
    }
  }
  return mincost;
}

int main() {
  int n = 6;
  int **cost = new int *[n];
  for (int i = 0; i < n; i++) {
    cost[i] = new int[n];
  }

  int **E = new int *[n * n];
  for (int i = 0; i < n * n; i++) {
    E[i] = new int[3];
  }

  int matrix[6][6] = {
      {0, 10, 2, 0, 0, 2}, {10, 0, 0, 0, 9, 8}, {2, 0, 0, 5, 0, 1},
      {0, 0, 5, 0, 7, 6},  {0, 9, 0, 7, 0, 0},  {2, 8, 1, 6, 0, 0},
  };
```



```
    int c = 0;
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        cost[i][j] = matrix[i][j];
        E[c][0] = i;
        E[c][1] = j;
        E[c][2] = cost[i][j];
        c++;
      }
    }

    int *near = new int[n];
    for (int i = 0; i < n; i++) {
      near[i] = 0;
    }

    int **t = new int *[n-1];
    for (int i = 0; i < n-1; i++) {
      t[i] = new int[2];
    }
    cout<<"Minimum cost = "<<prims(E, cost, n, t, near)<<"\n";

    for (int i = 0; i < n-1; i++) {
      cout << t[i][0] + 1 << " -> " << t[i][1] + 1 << "\n";
    }

    return 0;
}
```



All Pair Shortest Path

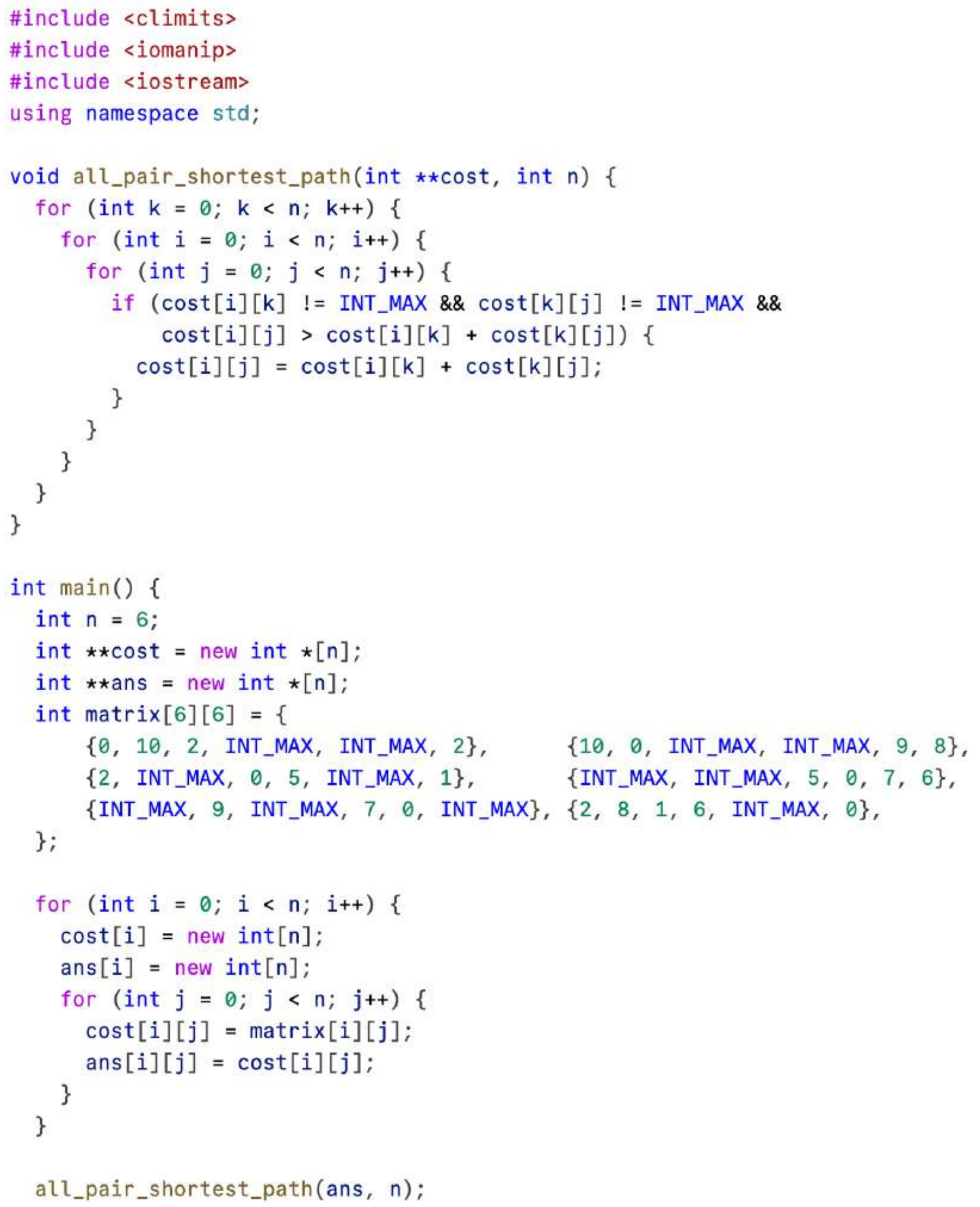
Idea –

- We find the minimum cost to go from any vertex to any other vertex
- We store the intermidiate result in the original adjency matrix or create a new one
  - The values are updated if we find a new minimum cost

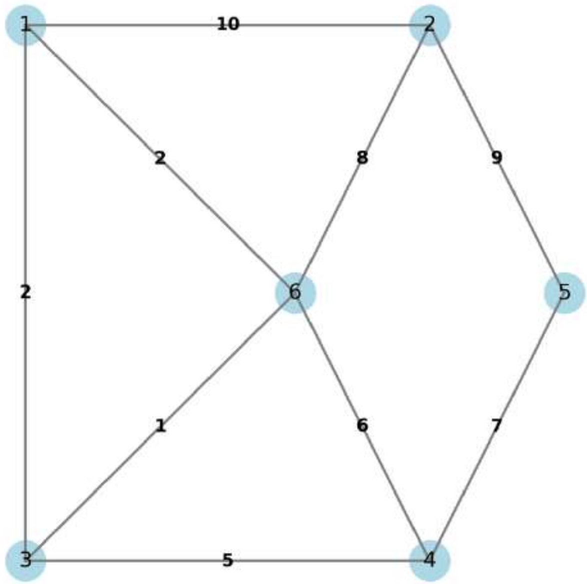
Complexity –

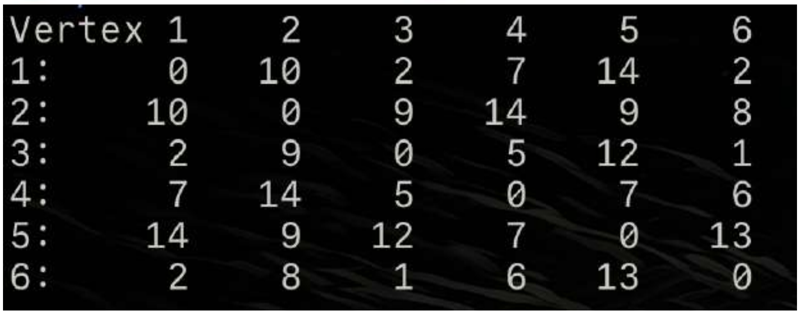
- Time – O(n^3)
- Space – O(n^2)

Code –



```
#include <climits>
#include <iomanip>
#include <iostream>
using namespace std;

void all_pair_shortest_path(int **cost, int n) {
  for (int k = 0; k < n; k++) {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        if (cost[i][k] != INT_MAX && cost[k][j] != INT_MAX &&
            cost[i][j] > cost[i][k] + cost[k][j]) {
          cost[i][j] = cost[i][k] + cost[k][j];
        }
      }
    }
  }
}

int main() {
  int n = 6;
  int **cost = new int *[n];
  int **ans = new int *[n];
  int matrix[6][6] = {
      {0, 10, 2, INT_MAX, INT_MAX, 2},      {10, 0, INT_MAX, INT_MAX, 9, 8},
      {2, INT_MAX, 0, 5, INT_MAX, 1},       {INT_MAX, INT_MAX, 5, 0, 7, 6},
      {INT_MAX, 9, INT_MAX, 7, 0, INT_MAX}, {2, 8, 1, 6, INT_MAX, 0},
  };

  for (int i = 0; i < n; i++) {
    cost[i] = new int[n];
    ans[i] = new int[n];
    for (int j = 0; j < n; j++) {
      cost[i][j] = matrix[i][j];
      ans[i][j] = cost[i][j];
    }
  }

  all_pair_shortest_path(ans, n);
  cout<<"Vertex 1    2    3    4    5    6\n";
  for (int i = 0; i < n; i++) {
    cout << i + 1 << ": ";
    for (int j = 0; j < n; j++) {
      if (ans[i][j] == INT_MAX)
        cout << setw(5) << "INF ";
      else
        cout << setw(5) << ans[i][j];
    }
    cout << endl;
  }
  return 0;
}
```





```
Vertex 1    2    3    4    5    6
1:     0   10    2    7   14    2
2:    10    0    9   14    9    8
3:     2    9    0    5   12    1
4:     7   14    5    0    7    6
5:    14    9   12    7    0   13
6:     2    8    1    6   13    0
```

Multistage Graph – Forward Method

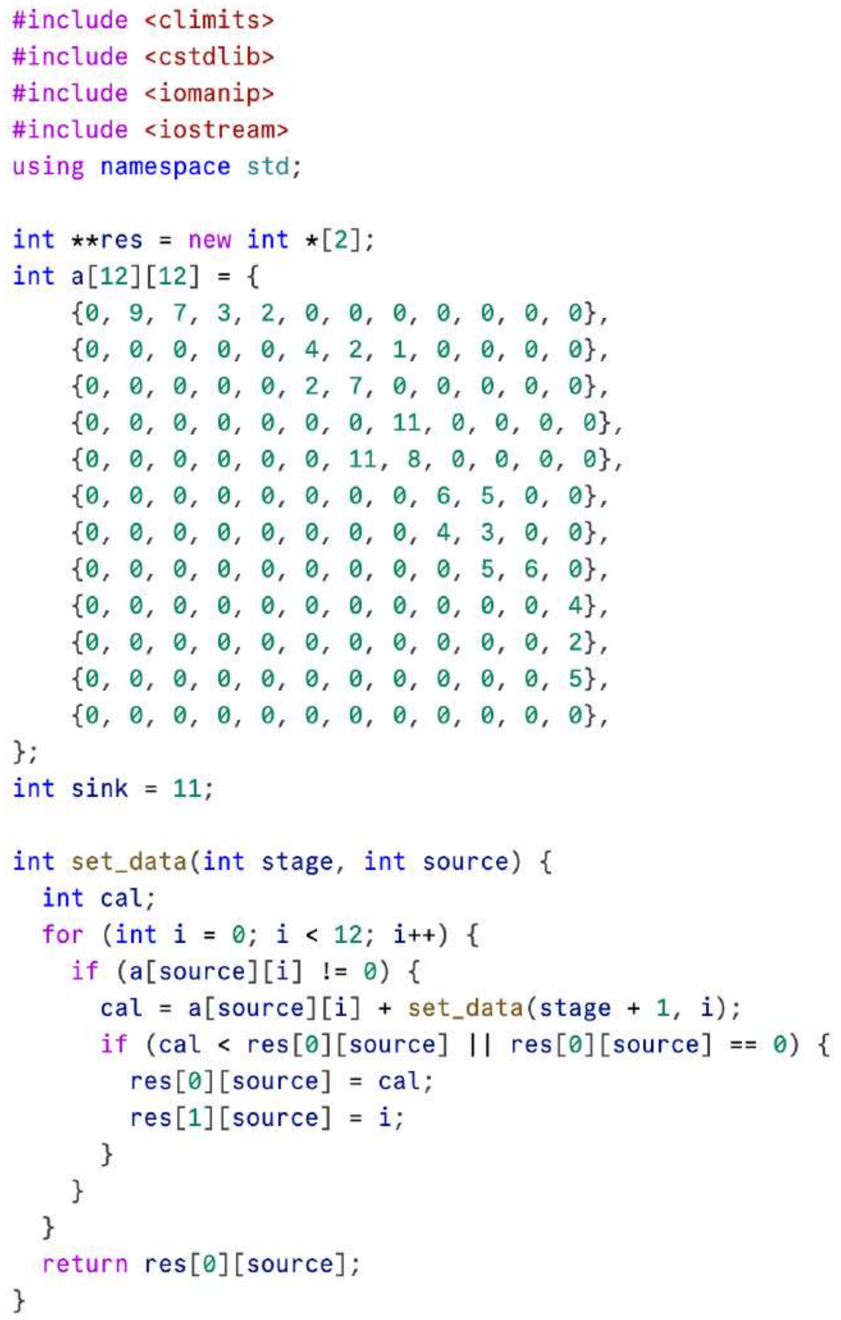
Idea –

- A graph that has a single starting node and a single ending node
- Find out the cost to go to from source to sink from any node
- Forward method goes from source to sink

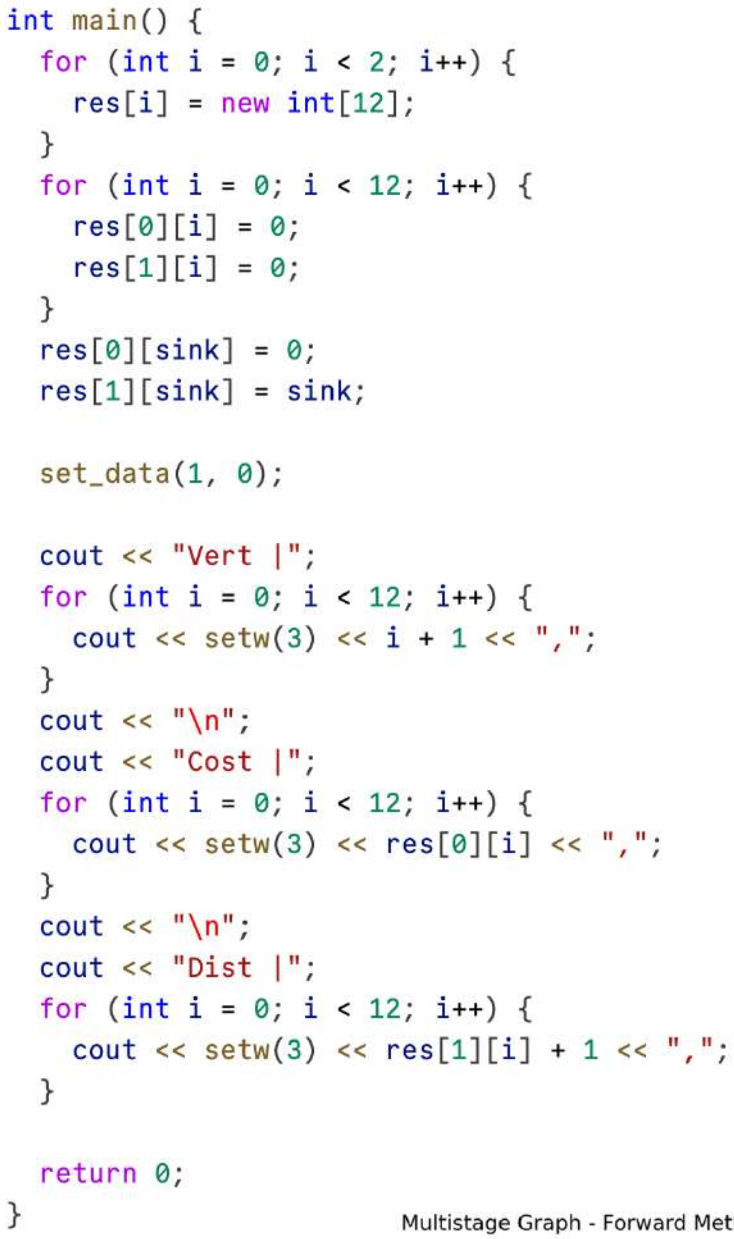
Complexity –

- Time – O(n^2)
- Space – O(n)

Code –

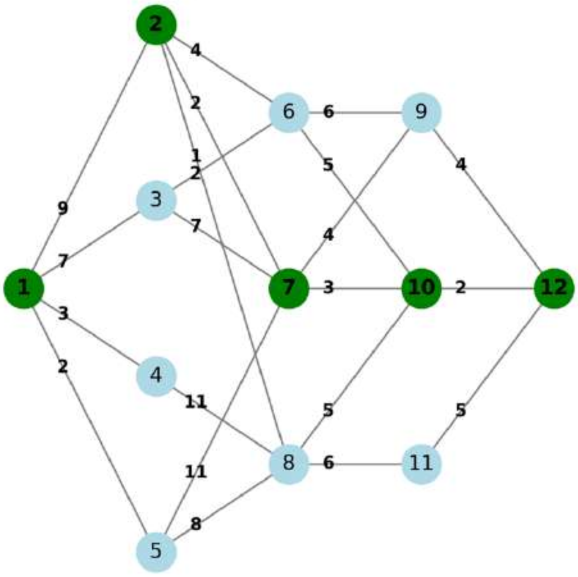


```
#include <climits>
#include <cstdlib>
#include <iomanip>
#include <iostream>
using namespace std;

int **res = new int *[2];
int a[12][12] = {
    {0, 9, 7, 3, 2, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 4, 2, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 2, 7, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 11, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 11, 8, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 6, 5, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 4, 3, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 6, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
};
int sink = 11;

int set_data(int stage, int source) {
  int cal;
  for (int i = 0; i < 12; i++) {
    if (a[source][i] != 0) {
      cal = a[source][i] + set_data(stage + 1, i);
      if (cal < res[0][source] || res[0][source] == 0) {
        res[0][source] = cal;
        res[1][source] = i;
      }
    }
  }
  return res[0][source];
}
```



```
int main() {
  for (int i = 0; i < 2; i++) {
    res[i] = new int[12];
  }
  for (int i = 0; i < 12; i++) {
    res[0][i] = 0;
    res[1][i] = 0;
  }
  res[0][sink] = 0;
  res[1][sink] = sink;

  set_data(1, 0);

  cout << "Vert |";
  for (int i = 0; i < 12; i++) {
    cout << setw(3) << i + 1 << ",";
  }
  cout << "\n";
  cout << "Cost |";
  for (int i = 0; i < 12; i++) {
    cout << setw(3) << res[0][i] << ",";
  }
  cout << "\n";
  cout << "Dist |";
  for (int i = 0; i < 12; i++) {
    cout << setw(3) << res[1][i] + 1 << ",";
  }

  return 0;
}
```

Multistage Graph - Forward Method



Multistage Graph – Backward Method

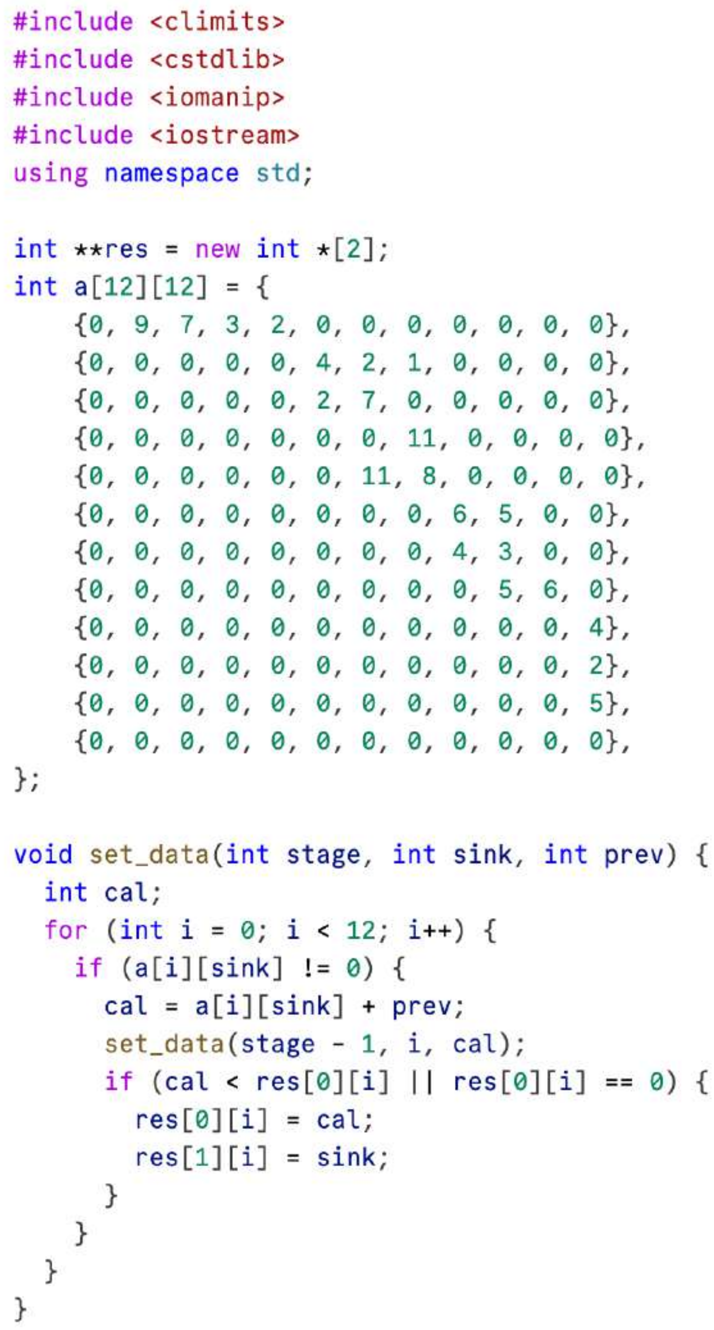
Idea –

- A graph that has a single starting node and a single ending node
- Find out the cost to go to from source to sink from any node
- Backward method goes from sink to source

Complexity –

- Time – O(n^3)
- Space – O(n)

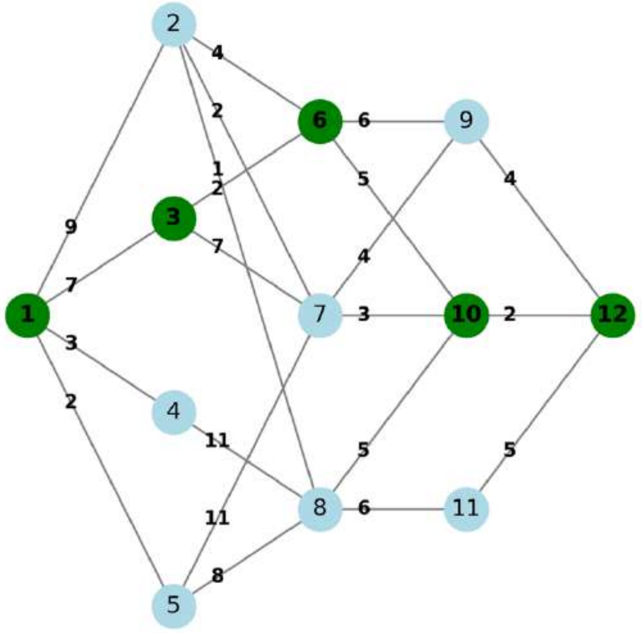
Code –



```
#include <climits>
#include <cstdlib>
#include <iomanip>
#include <iostream>
using namespace std;

int **res = new int *[2];
int a[12][12] = {
    {0, 9, 7, 3, 2, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 4, 2, 1, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 2, 7, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 11, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 11, 8, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 6, 5, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 4, 3, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 6, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
};

void set_data(int stage, int sink, int prev) {
  int cal;
  for (int i = 0; i < 12; i++) {
    if (a[i][sink] != 0) {
      cal = a[i][sink] + prev;
      set_data(stage - 1, i, cal);
      if (cal < res[0][i] || res[0][i] == 0) {
        res[0][i] = cal;
        res[1][i] = sink;
      }
    }
  }
}
```

```
int main() {
  int sink = 11;
  for (int i = 0; i < 2; i++) {
    res[i] = new int[12];
  }
  for (int i = 0; i < 12; i++) {
    res[0][i] = 0;
    res[1][i] = 0;
  }
  res[0][sink] = 0;
  res[1][sink] = sink;

  set_data(5, 11, 0);

  cout << "Vert |";
  for (int i = 0; i < 12; i++) {
    cout << setw(3) << i + 1 << ",";
  }
  cout << "\n";
  cout << "Cost |";
  for (int i = 0; i < 12; i++) {
    cout << setw(3) << res[0][i] << ",";
  }
  cout << "\n";
  cout << "Dist |";
  for (int i = 0; i < 12; i++) {
    cout << setw(3) << res[1][i] + 1 << ",";
  }

  return 0;
}
```

Multistage Graph - Backward Method



Matrix Chain Multiplications

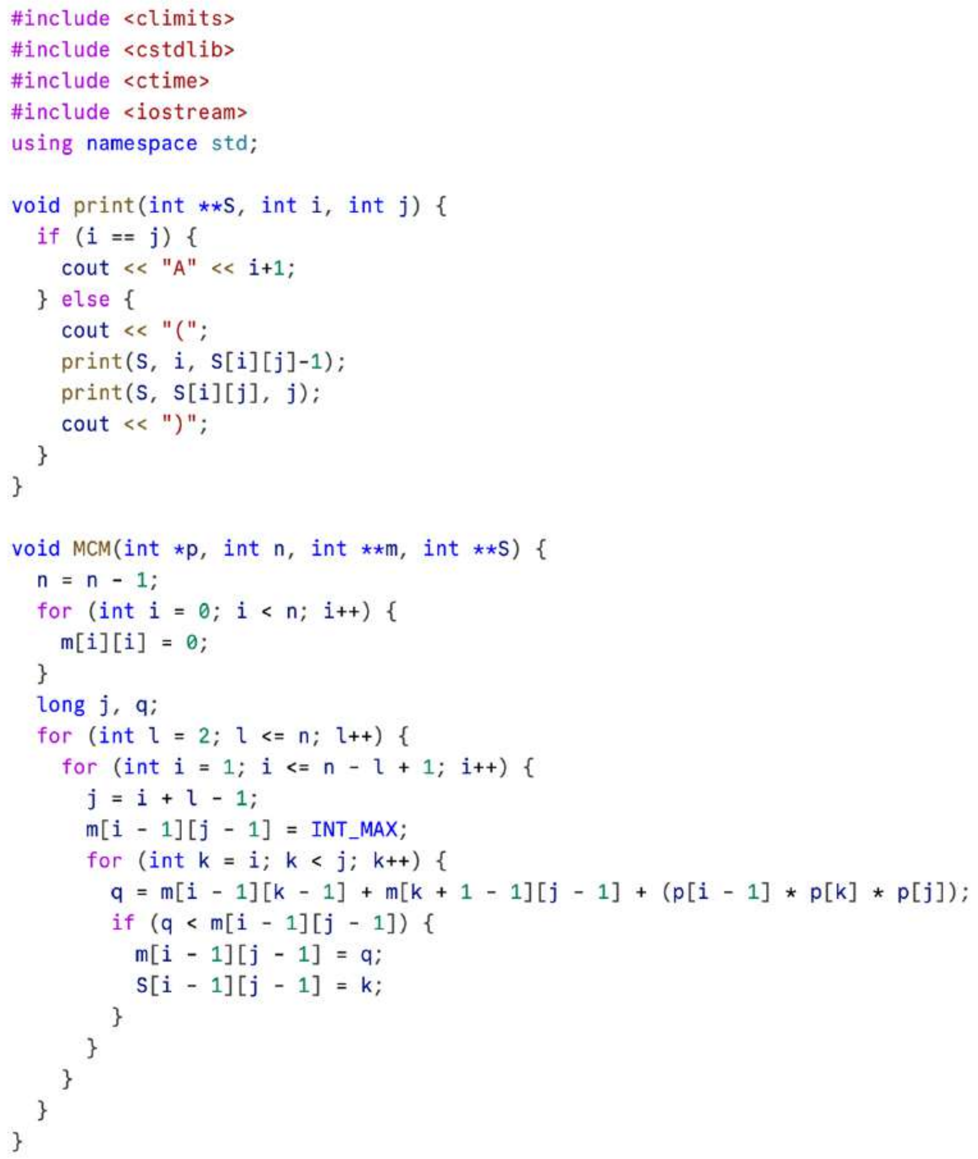
Idea –

- Find out the minimum cost of matrix multiplication by reducing the number of scalar multiplication needed
- Bundles together matrix multiplications using their sizes and shows the optimal order of multiplication
- Creates a 2D array containing the cost to multiply any combination of the matrixes

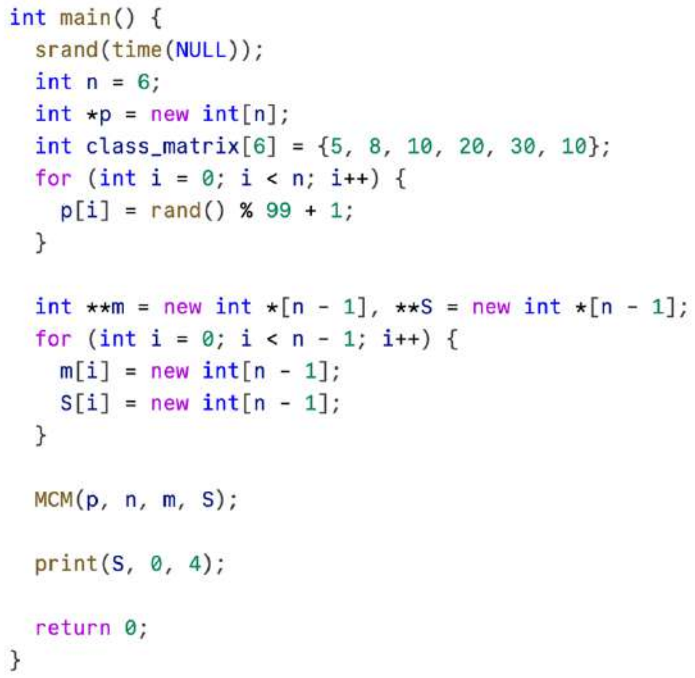
Complexity –

- Time – O(n^3)
- Space – O(n^2)

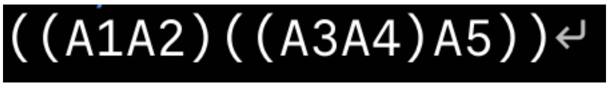
Code –



```
#include <climits>
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

void print(int **S, int i, int j) {
  if (i == j) {
    cout << "A" << i+1;
  } else {
    cout << "(";
    print(S, i, S[i][j]-1);
    print(S, S[i][j], j);
    cout << ")";
  }
}

void MCM(int *p, int n, int **m, int **S) {
  n = n - 1;
  for (int i = 0; i < n; i++) {
    m[i][i] = 0;
  }
  long j, q;
  for (int l = 2; l <= n; l++) {
    for (int i = 1; i <= n - l + 1; i++) {
      j = i + l - 1;
      m[i - 1][j - 1] = INT_MAX;
      for (int k = i; k < j; k++) {
        q = m[i - 1][k - 1] + m[k + 1 - 1][j - 1] + (p[i - 1] * p[k] * p[j]);
        if (q < m[i - 1][j - 1]) {
          m[i - 1][j - 1] = q;
          S[i - 1][j - 1] = k;
        }
      }
    }
  }
}
```



```
int main() {
  srand(time(NULL));
  int n = 6;
  int *p = new int[n];
  int class_matrix[6] = {5, 8, 10, 20, 30, 10};
  for (int i = 0; i < n; i++) {
    p[i] = rand() % 99 + 1;
  }

  int **m = new int *[n - 1], **S = new int *[n - 1];
  for (int i = 0; i < n - 1; i++) {
    m[i] = new int[n - 1];
    S[i] = new int[n - 1];
  }

  MCM(p, n, m, S);

  print(S, 0, 4);

  return 0;
}
```



```
((A1A2)((A3A4)A5))↵
```

Knapsack – Sets Method

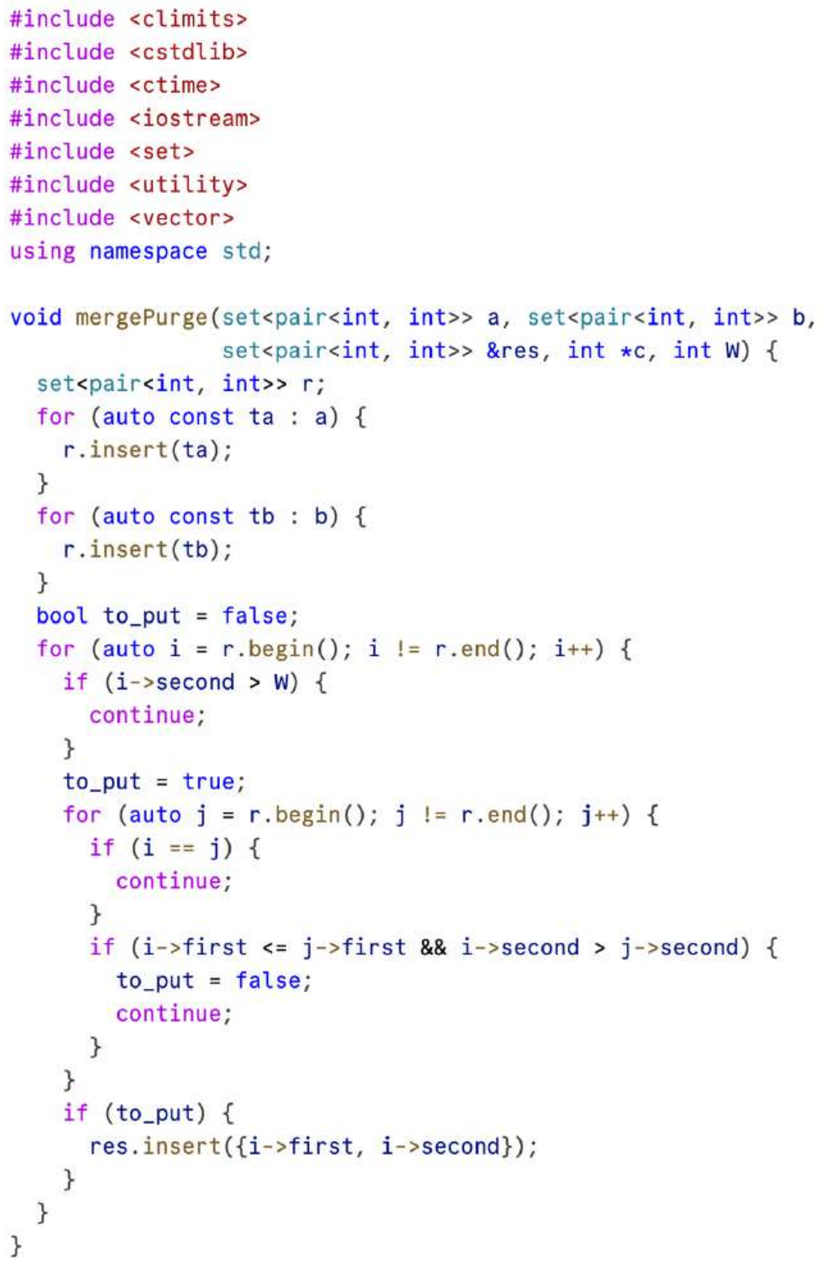
Idea –

- Solve the 0/1 knapsack problem using dynamic programming approach
- Merge complementary sets and purge elements that are clearly not needed
- Elements where their weight is more than or equal to another and their profit is lower than the same element, the element with the higher weight is removed

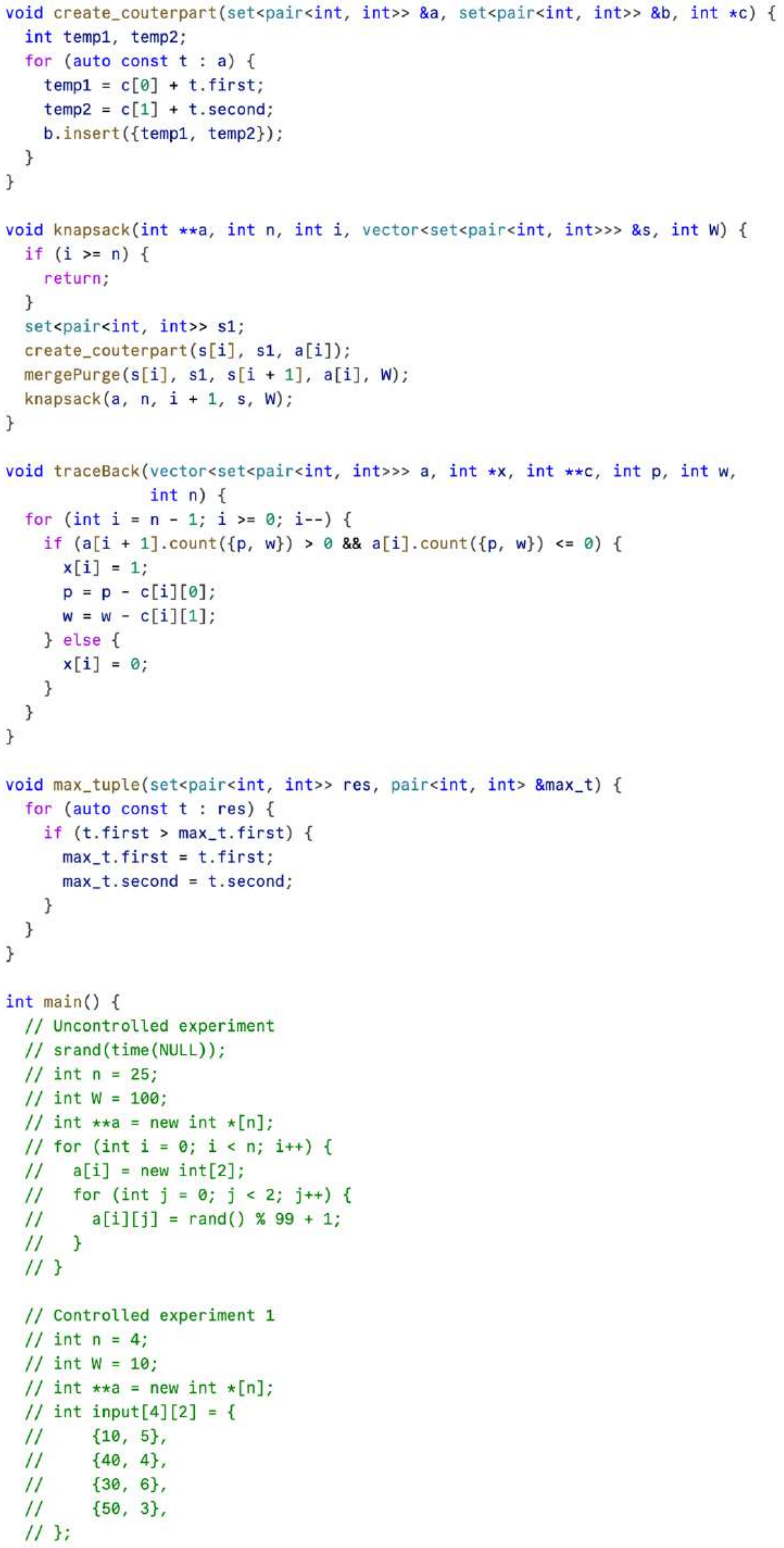
Complexity –

- Time – O(n^2)
- Space – O(n)

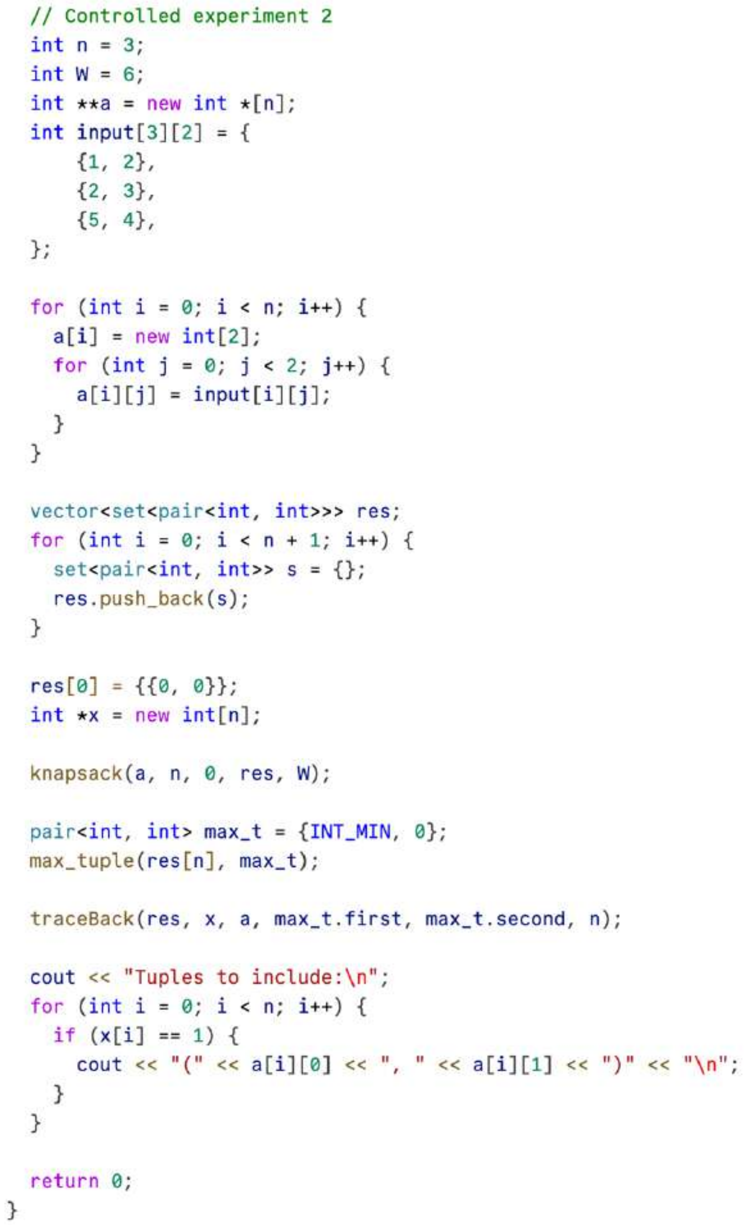
Code –



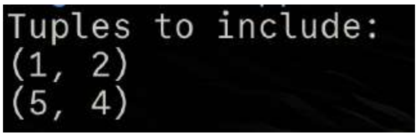
```
#include <climits>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <set>
#include <utility>
#include <vector>
using namespace std;

void mergePurge(set<pair<int, int>> a, set<pair<int, int>> b,
                set<pair<int, int>> &res, int *c, int W) {
  set<pair<int, int>> r;
  for (auto const ta : a) {
    r.insert(ta);
  }
  for (auto const tb : b) {
    r.insert(tb);
  }
  bool to_put = false;
  for (auto i = r.begin(); i != r.end(); i++) {
    if (i->second > W) {
      continue;
    }
    to_put = true;
    for (auto j = r.begin(); j != r.end(); j++) {
      if (i == j) {
        continue;
      }
      if (i->first <= j->first && i->second > j->second) {
        to_put = false;
        continue;
      }
    }
    if (to_put) {
      res.insert({i->first, i->second});
    }
  }
}
```



```
void create_couterpart(set<pair<int, int>> &a, set<pair<int, int>> &b, int *c) {
  int temp1, temp2;
  for (auto const t : a) {
    temp1 = c[0] + t.first;
    temp2 = c[1] + t.second;
    b.insert({temp1, temp2});
  }
}

void knapsack(int **a, int n, int i, vector<set<pair<int, int>>> &s, int W) {
  if (i >= n) {
    return;
  }
  set<pair<int, int>> s1;
  create_couterpart(s[i], s1, a[i]);
  mergePurge(s[i], s1, s[i + 1], a[i], W);
  knapsack(a, n, i + 1, s, W);
}

void traceBack(vector<set<pair<int, int>>> a, int *x, int **c, int p, int w,
               int n) {
  for (int i = n - 1; i >= 0; i--) {
    if (a[i + 1].count({p, w}) > 0 && a[i].count({p, w}) <= 0) {
      x[i] = 1;
      p = p - c[i][0];
      w = w - c[i][1];
    } else {
      x[i] = 0;
    }
  }
}

void max_tuple(set<pair<int, int>> res, pair<int, int> &max_t) {
  for (auto const t : res) {
    if (t.first > max_t.first) {
      max_t.first = t.first;
      max_t.second = t.second;
    }
  }
}

int main() {
  // Uncontrolled experiment
  // srand(time(NULL));
  // int n = 25;
  // int W = 100;
  // int **a = new int *[n];
  // for (int i = 0; i < n; i++) {
  //   a[i] = new int[2];
  //   for (int j = 0; j < 2; j++) {
  //     a[i][j] = rand() % 99 + 1;
  //   }
  // }

  // Controlled experiment 1
  // int n = 4;
  // int W = 10;
  // int **a = new int *[n];
  // int input[4][2] = {
  //     {10, 5},
  //     {40, 4},
  //     {30, 6},
  //     {50, 3},
  // };
```



```
  // Controlled experiment 2
  int n = 3;
  int W = 6;
  int **a = new int *[n];
  int input[3][2] = {
      {1, 2},
      {2, 3},
      {5, 4},
  };

  for (int i = 0; i < n; i++) {
    a[i] = new int[2];
    for (int j = 0; j < 2; j++) {
      a[i][j] = input[i][j];
    }
  }

  vector<set<pair<int, int>>> res;
  for (int i = 0; i < n + 1; i++) {
    set<pair<int, int>> s = {};
    res.push_back(s);
  }

  res[0] = {{0, 0}};
  int *x = new int[n];

  knapsack(a, n, 0, res, W);

  pair<int, int> max_t = {INT_MIN, 0};
  max_tuple(res[n], max_t);

  traceBack(res, x, a, max_t.first, max_t.second, n);

  cout << "Tuples to include:\n";
  for (int i = 0; i < n; i++) {
    if (x[i] == 1) {
      cout << "(" << a[i][0] << ", " << a[i][1] << ")" << "\n";
    }
  }

  return 0;
}
```



```
Tuples to include:
(1, 2)
(5, 4)
```

Longest Common Subsequence

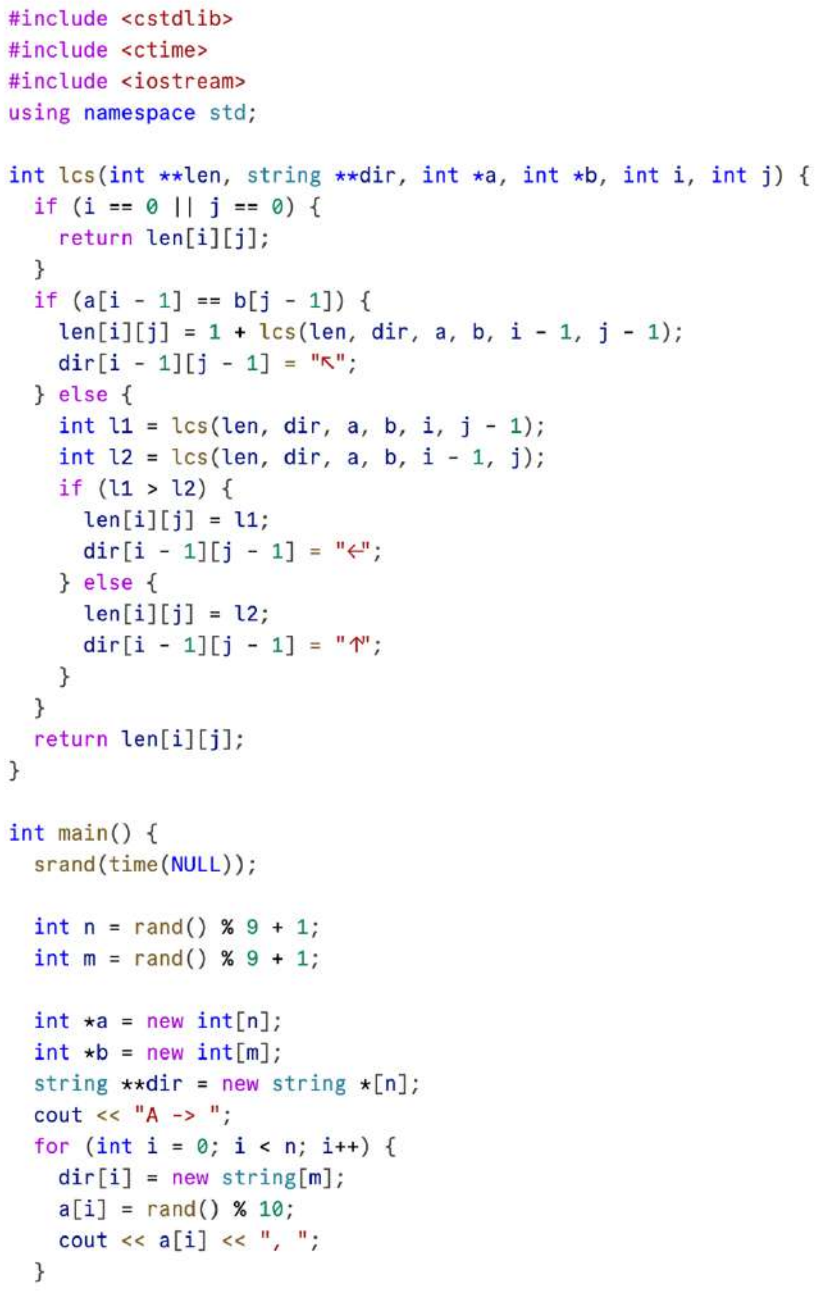
Idea –

- Find out the Longest common subsequence from a 2 given strings
- Starts from the ends of the strings and compares if there are any same elements in the other string
- Creates a 2D array containing the path to create the subsequence

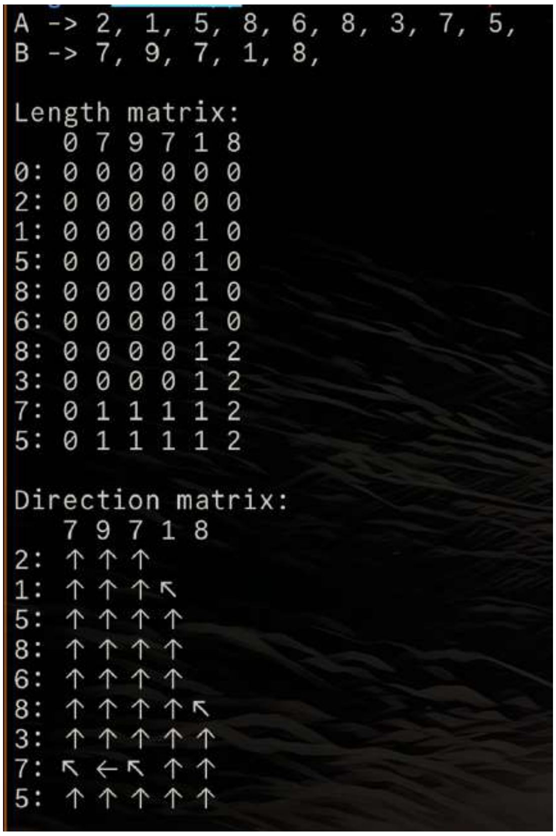
Complexity –

- Time – O(mn)
- Space – O(mn)

Code –



```
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

int lcs(int **len, string **dir, int *a, int *b, int i, int j) {
  if (i == 0 || j == 0) {
    return len[i][j];
  }
  if (a[i - 1] == b[j - 1]) {
    len[i][j] = 1 + lcs(len, dir, a, b, i - 1, j - 1);
    dir[i - 1][j - 1] = "↖";
  } else {
    int l1 = lcs(len, dir, a, b, i, j - 1);
    int l2 = lcs(len, dir, a, b, i - 1, j);
    if (l1 > l2) {
      len[i][j] = l1;
      dir[i - 1][j - 1] = "←";
    } else {
      len[i][j] = l2;
      dir[i - 1][j - 1] = "↑";
    }
  }
  return len[i][j];
}

int main() {
  srand(time(NULL));

  int n = rand() % 9 + 1;
  int m = rand() % 9 + 1;

  int *a = new int[n];
  int *b = new int[m];
  string **dir = new string *[n];
  cout << "A -> ";
  for (int i = 0; i < n; i++) {
    dir[i] = new string[m];
    a[i] = rand() % 10;
    cout << a[i] << ", ";
  }
  cout << "\n";
  cout << "B -> ";
  for (int i = 0; i < m; i++) {
    b[i] = rand() % 10;
    cout << b[i] << ", ";
  }
  cout << "\n";

  int **len = new int *[n + 1];
  for (int i = 0; i < n + 1; i++) {
    len[i] = new int[m + 1];
    for (int j=0; j<m+1; j++) {
      len[i][j] = 0;
    }
  }
  lcs(len, dir, a, b, n, m);

  cout << "\nLength matrix:\n";
  cout << "   0 ";
  for (int i = 0; i < m; i++) {
    cout << b[i] << " ";
  }
  cout << "\n";
  for (int i = 0; i < n + 1; i++) {
    i == 0 ? cout << "0: " : cout << a[i-1] << ": ";
    for (int j = 0; j < m + 1; j++) {
      cout << len[i][j] << " ";
    }
    cout << "\n";
  }

  cout << "\nDirection matrix:\n";
  cout << "   ";
  for (int i = 0; i < m; i++) {
    cout << b[i] << " ";
  }
  cout << "\n";
  for (int i = 0; i < n; i++) {
    cout << a[i] << ": ";
    for (int j = 0; j < m; j++) {
      cout << (dir[i][j] != "" ? dir[i][j] : " ") << " ";
    }
    cout << "\n";
  }
  return 0;
}
```



```
A -> 2, 1, 5, 8, 6, 8, 3, 7, 5,
B -> 7, 9, 7, 1, 8,

Length matrix:
   0 7 9 7 1 8
0: 0 0 0 0 0 0
2: 0 0 0 0 0 0
1: 0 0 0 0 1 0
5: 0 0 0 0 1 0
8: 0 0 0 0 1 0
6: 0 0 0 0 1 0
8: 0 0 0 0 1 2
3: 0 0 0 0 1 2
7: 0 1 1 1 1 2
5: 0 1 1 1 1 2

Direction matrix:
   7 9 7 1 8
2: ↑ ↑ ↑
1: ↑ ↑ ↑ ↖
5: ↑ ↑ ↑ ↑
8: ↑ ↑ ↑ ↑
6: ↑ ↑ ↑ ↑
8: ↑ ↑ ↑ ↑ ↖
3: ↑ ↑ ↑ ↑ ↑
7: ↖ ← ↖ ↑ ↑
5: ↑ ↑ ↑ ↑ ↑
```

N Queens

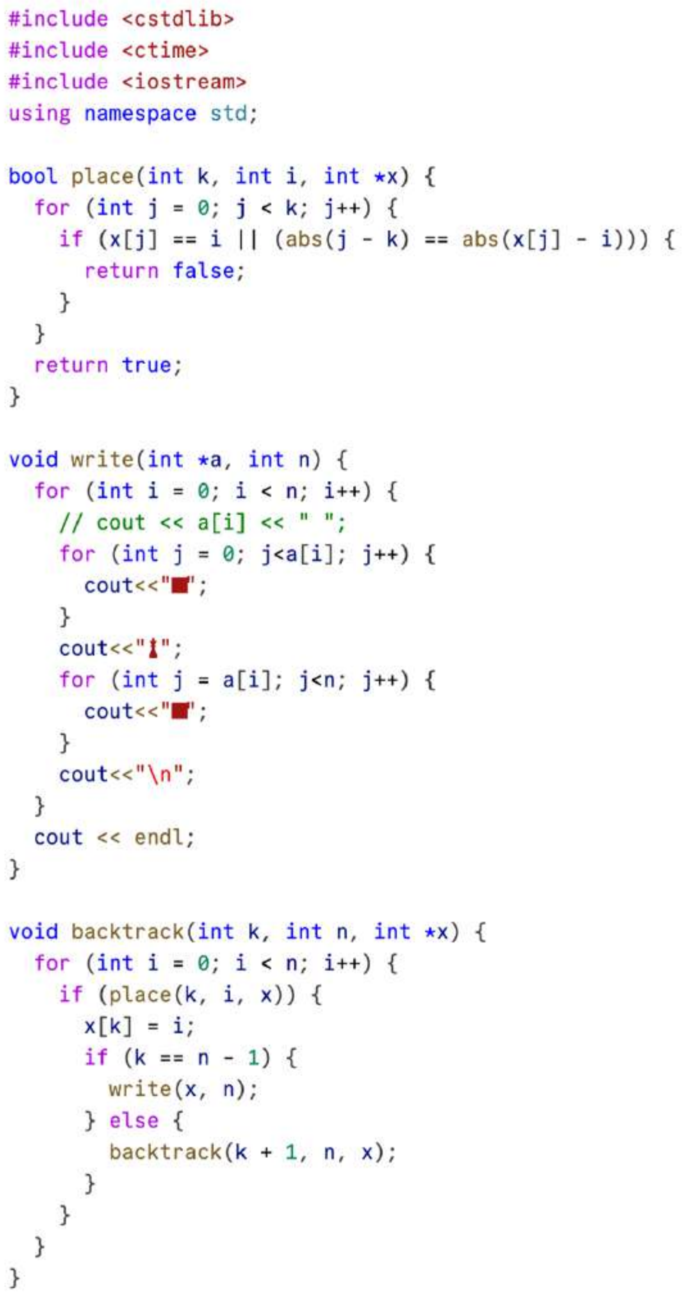
Idea –

- Place N Queens on a NxN board
- Using a backtracking approach to start filling out from the starting square
- Backtrack on the given condition that col != col and row != row and (abs(j - k) != abs(x[j] - i)) which makes sure that the queens can't see each other

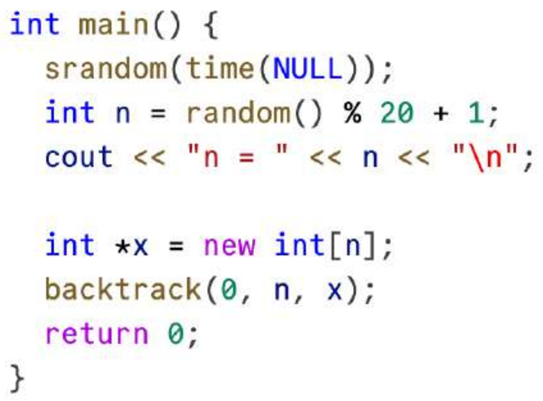
Complexity –

- Time – O(n!)
- Space – O(n)

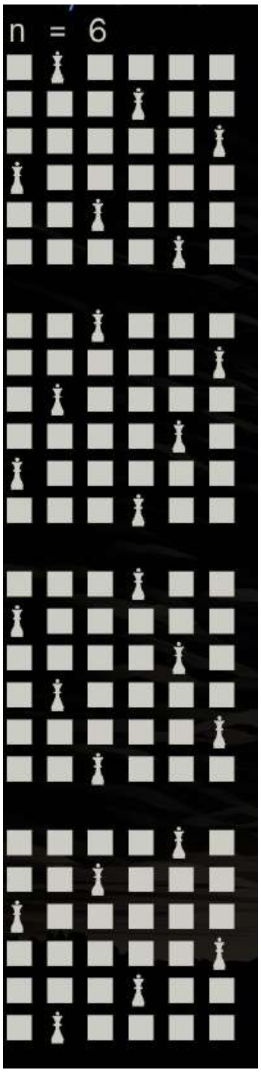
Code –



```
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

bool place(int k, int i, int *x) {
  for (int j = 0; j < k; j++) {
    if (x[j] == i || (abs(j - k) == abs(x[j] - i))) {
      return false;
    }
  }
  return true;
}

void write(int *a, int n) {
  for (int i = 0; i < n; i++) {
    // cout << a[i] << " ";
    for (int j = 0; j<a[i]; j++) {
      cout<<"🟥";
    }
    cout<<"♛";
    for (int j = a[i]; j<n; j++) {
      cout<<"🟥";
    }
    cout<<"\n";
  }
  cout << endl;
}

void backtrack(int k, int n, int *x) {
  for (int i = 0; i < n; i++) {
    if (place(k, i, x)) {
      x[k] = i;
      if (k == n - 1) {
        write(x, n);
      } else {
        backtrack(k + 1, n, x);
      }
    }
  }
}
```



```
int main() {
  srandom(time(NULL));
  int n = random() % 20 + 1;
  cout << "n = " << n << "\n";

  int *x = new int[n];
  backtrack(0, n, x);
  return 0;
}
```



```
n = 6
⬜ ♕ ⬜ ⬜ ⬜ ⬜
⬜ ⬜ ⬜ ♕ ⬜ ⬜
⬜ ⬜ ⬜ ⬜ ⬜ ♕
♕ ⬜ ⬜ ⬜ ⬜ ⬜
⬜ ⬜ ♕ ⬜ ⬜ ⬜
⬜ ⬜ ⬜ ⬜ ♕ ⬜

⬜ ⬜ ♕ ⬜ ⬜ ⬜
⬜ ⬜ ⬜ ⬜ ⬜ ♕
⬜ ♕ ⬜ ⬜ ⬜ ⬜
⬜ ⬜ ⬜ ⬜ ♕ ⬜
♕ ⬜ ⬜ ⬜ ⬜ ⬜
⬜ ⬜ ⬜ ♕ ⬜ ⬜

⬜ ⬜ ⬜ ♕ ⬜ ⬜
♕ ⬜ ⬜ ⬜ ⬜ ⬜
⬜ ⬜ ⬜ ⬜ ♕ ⬜
⬜ ♕ ⬜ ⬜ ⬜ ⬜
⬜ ⬜ ⬜ ⬜ ⬜ ♕
⬜ ⬜ ♕ ⬜ ⬜ ⬜

⬜ ⬜ ⬜ ⬜ ♕ ⬜
⬜ ⬜ ♕ ⬜ ⬜ ⬜
♕ ⬜ ⬜ ⬜ ⬜ ⬜
⬜ ⬜ ⬜ ⬜ ⬜ ♕
⬜ ⬜ ⬜ ♕ ⬜ ⬜
⬜ ♕ ⬜ ⬜ ⬜ ⬜
```

M colorable graph

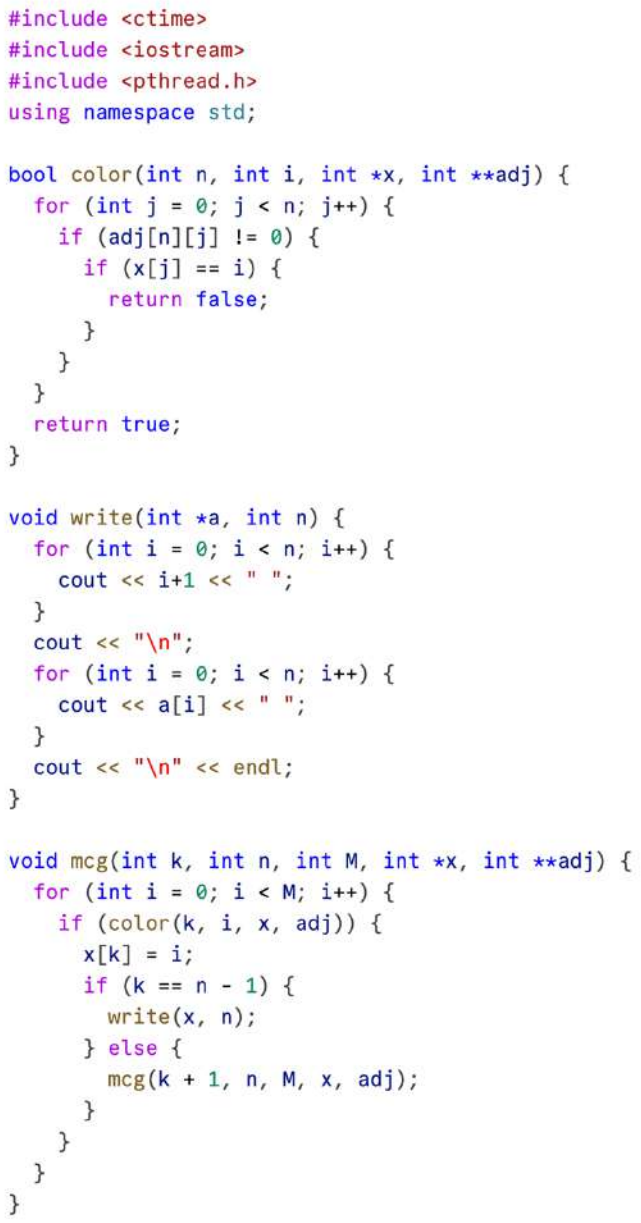
Idea –

- Find out the ways in which a graph can be colored using M colors without the adjecent nodes having the same color
- Using a backtracking approach to try out all the colors on all nodes
- Backtrack on the given condition that color appears on one of the neighbours of the node.

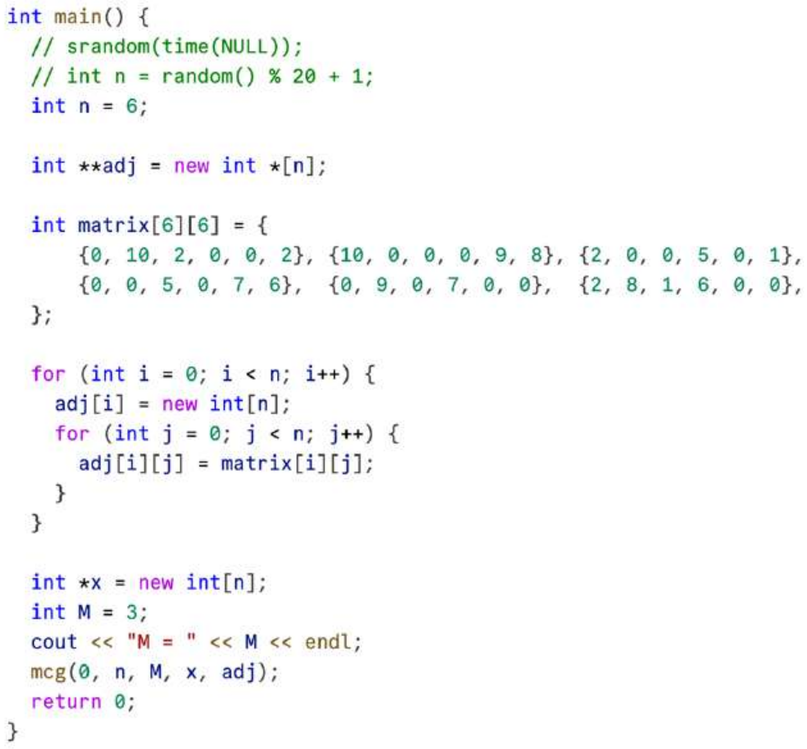
Complexity –

- Time – O(M^n)
- Space – O(n)

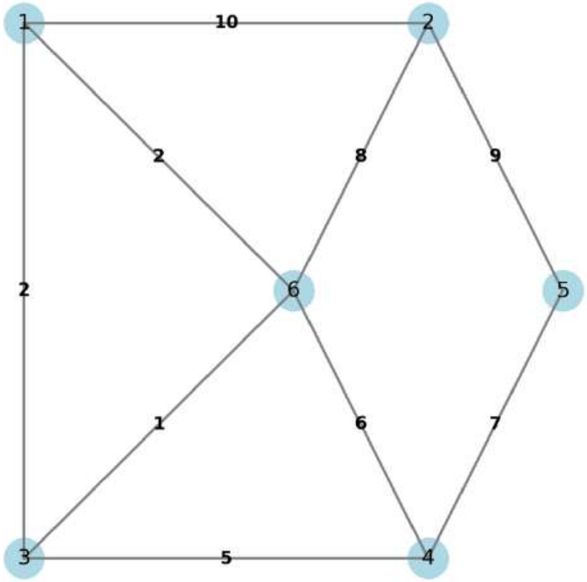
Code –

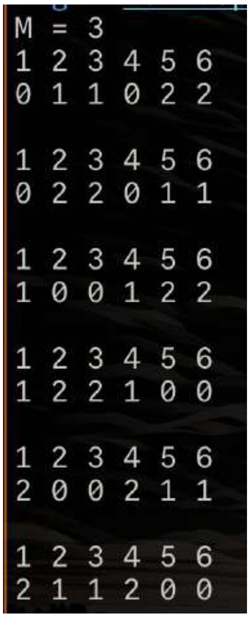


```cpp
#include <ctime>
#include <iostream>
#include <pthread.h>
using namespace std;

bool color(int n, int i, int *x, int **adj) {
  for (int j = 0; j < n; j++) {
    if (adj[n][j] != 0) {
      if (x[j] == i) {
        return false;
      }
    }
  }
  return true;
}

void write(int *a, int n) {
  for (int i = 0; i < n; i++) {
    cout << i+1 << " ";
  }
  cout << "\n";
  for (int i = 0; i < n; i++) {
    cout << a[i] << " ";
  }
  cout << "\n" << endl;
}

void mcg(int k, int n, int M, int *x, int **adj) {
  for (int i = 0; i < M; i++) {
    if (color(k, i, x, adj)) {
      x[k] = i;
      if (k == n - 1) {
        write(x, n);
      } else {
        mcg(k + 1, n, M, x, adj);
      }
    }
  }
}
```



```
int main() {
  // srandom(time(NULL));
  // int n = random() % 20 + 1;
  int n = 6;

  int **adj = new int *[n];

  int matrix[6][6] = {
      {0, 10, 2, 0, 0, 2}, {10, 0, 0, 0, 9, 8}, {2, 0, 0, 5, 0, 1},
      {0, 0, 5, 0, 7, 6},  {0, 9, 0, 7, 0, 0},  {2, 8, 1, 6, 0, 0},
  };

  for (int i = 0; i < n; i++) {
    adj[i] = new int[n];
    for (int j = 0; j < n; j++) {
      adj[i][j] = matrix[i][j];
    }
  }

  int *x = new int[n];
  int M = 3;
  cout << "M = " << M << endl;
  mcg(0, n, M, x, adj);
  return 0;
}
```





```
M = 3
1 2 3 4 5 6
0 1 1 0 2 2

1 2 3 4 5 6
0 2 2 0 1 1

1 2 3 4 5 6
1 0 0 1 2 2

1 2 3 4 5 6
1 2 2 1 0 0

1 2 3 4 5 6
2 0 0 2 1 1

1 2 3 4 5 6
2 1 1 2 0 0
```

Hamiltonian Graph

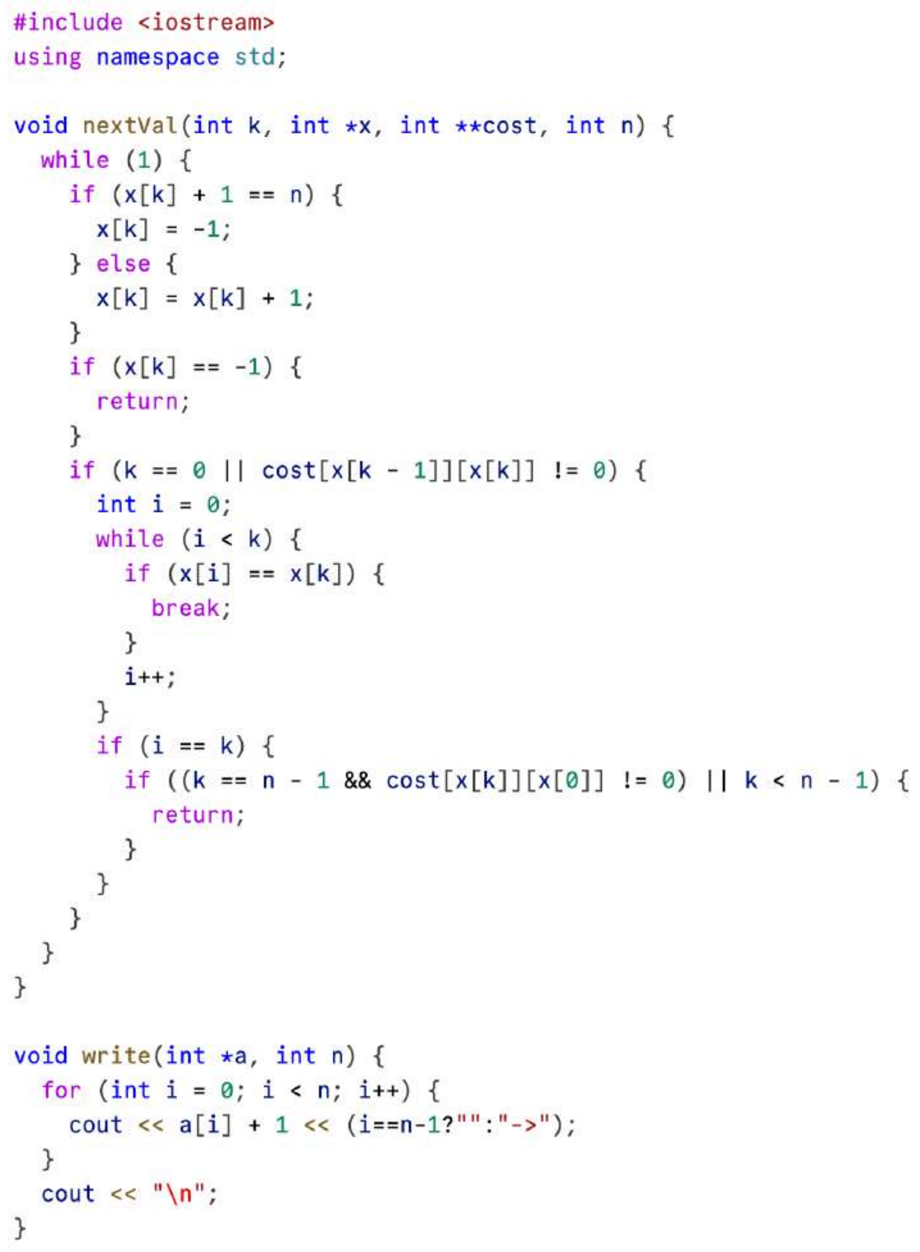
Idea –

- Find a path in the graph where we visit all the vertexs once without repition and we end up on the same vertex we started from
- Using the backtracking approach we try to explore all the path and return when we can't find a vertex to traverse to other vertexs without repeating an old one
- Gives out the correct order to traverse the graph to visit all the vertexs once

Complexity –

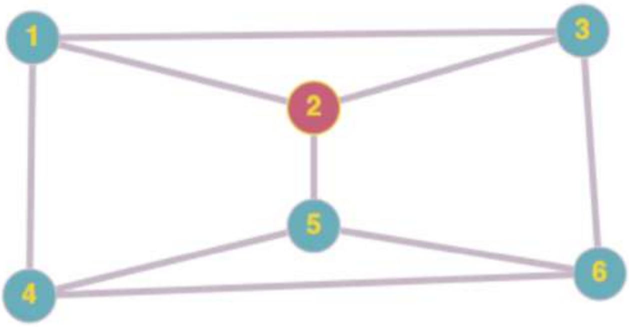
- Time – O(n!)
- Space – O(n)

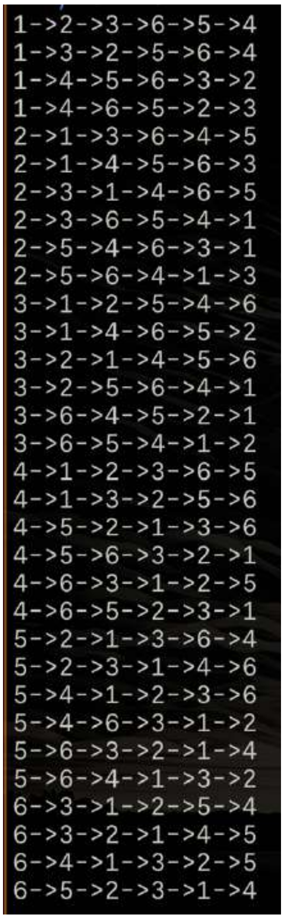
Code –



```
#include <iostream>
using namespace std;

void nextVal(int k, int *x, int **cost, int n) {
  while (1) {
    if (x[k] + 1 == n) {
      x[k] = -1;
    } else {
      x[k] = x[k] + 1;
    }
    if (x[k] == -1) {
      return;
    }
    if (k == 0 || cost[x[k - 1]][x[k]] != 0) {
      int i = 0;
      while (i < k) {
        if (x[i] == x[k]) {
          break;
        }
        i++;
      }
      if (i == k) {
        if ((k == n - 1 && cost[x[k]][x[0]] != 0) || k < n - 1) {
          return;
        }
      }
    }
  }
}

void write(int *a, int n) {
  for (int i = 0; i < n; i++) {
    cout << a[i] + 1 << (i==n-1?"":"->");
  }
  cout << "\n";
}
```

```
void backtrack(int k, int *x, int **cost, int n) {
  do {
    nextVal(k, x, cost, n);
    if (x[k] == -1) {
      return;
    }
    if (k == n - 1) {
      write(x, n);
      return;
    } else {
      backtrack(k + 1, x, cost, n);
    }
  } while (1);
}

int main() {
  int n = 6;
  int **cost = new int *[n];
  int *x = new int[n];
  for (int i = 0; i < n; i++) {
    x[i] = -1;
    cost[i] = new int[n];
  }
  int matrix[6][6] = {
      {0, 1, 1, 1, 0, 0}, {1, 0, 1, 0, 1, 0}, {1, 1, 0, 0, 0, 1},
      {1, 0, 0, 0, 1, 1}, {0, 1, 0, 1, 0, 1}, {0, 0, 1, 1, 1, 0},
  };

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      cost[i][j] = matrix[i][j];
    }
  }

  backtrack(0, x, cost, n);
  return 0;
}
```





```
1->2->3->6->5->4
1->3->2->5->6->4
1->4->5->6->3->2
1->4->6->5->2->3
2->1->3->6->4->5
2->1->4->5->6->3
2->3->1->4->6->5
2->3->6->5->4->1
2->5->4->6->3->1
2->5->6->4->1->3
3->1->2->5->4->6
3->1->4->6->5->2
3->2->1->4->5->6
3->2->5->6->4->1
3->6->4->5->2->1
3->6->5->4->1->2
4->1->2->3->6->5
4->1->3->2->5->6
4->5->2->1->3->6
4->5->6->3->2->1
4->6->3->1->2->5
4->6->5->2->3->1
5->2->1->3->6->4
5->2->3->1->4->6
5->4->1->2->3->6
5->4->6->3->1->2
5->6->3->2->1->4
5->6->4->1->3->2
6->3->1->2->5->4
6->3->2->1->4->5
6->4->1->3->2->5
6->5->2->3->1->4
```

Sum Of Subsets

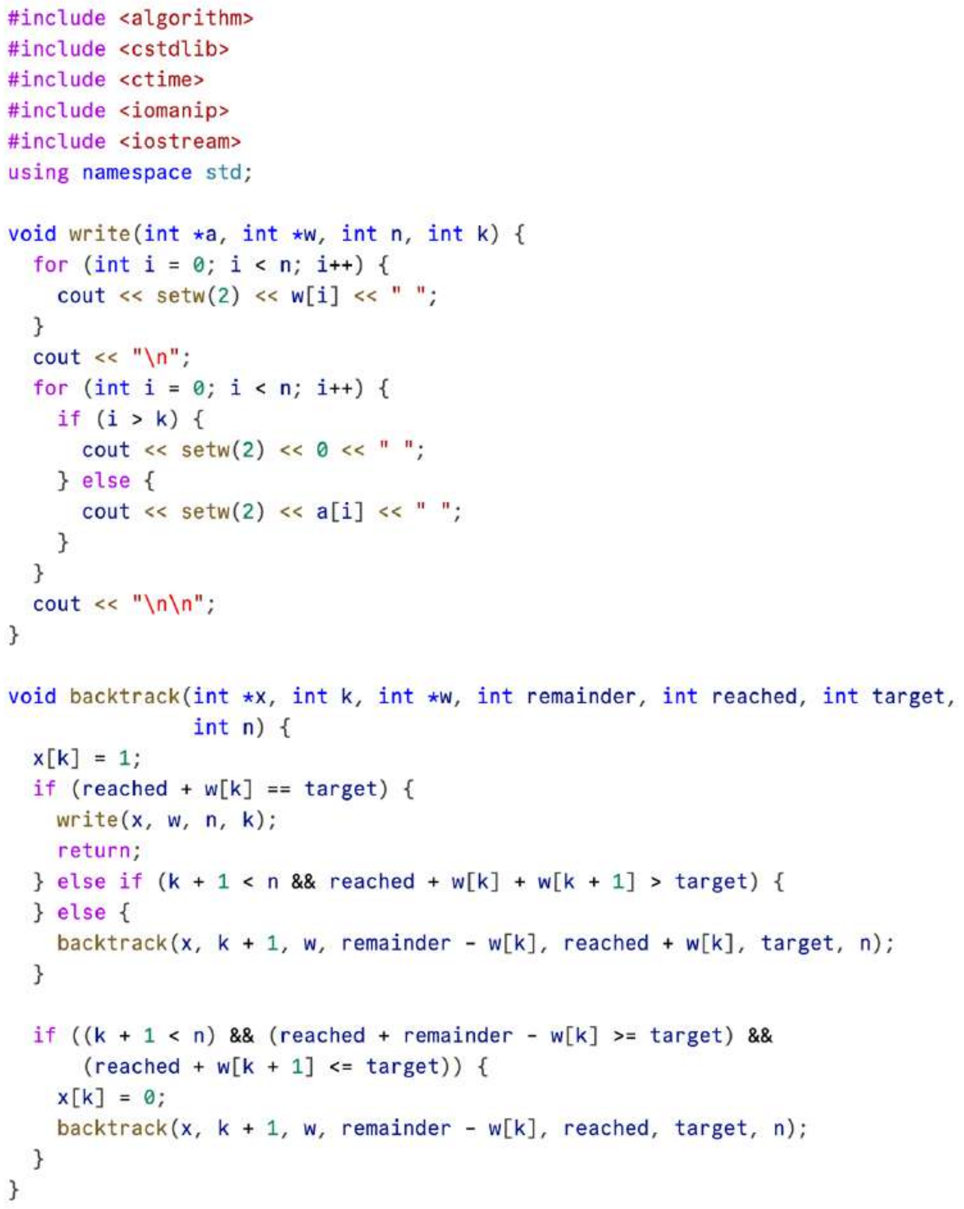
Idea –

- Finding all the subsets of a set whose sum is equal to a given value
- Assumes that we sort the starting array
- Using backtracking we try out all possible combinations for the subsets and quit when we -
    - Find that the remaining values are less than what we require
    - Current value generated + new value will tip us over the required amount
- Gives out the elements that make up that valid tuple

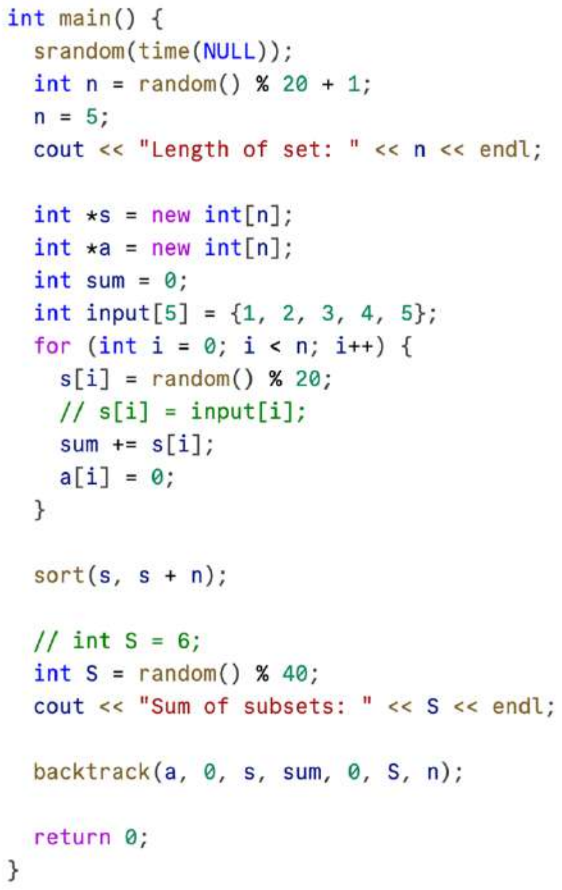
Complexity –

- Time – O(2^n)
- Space – O(n)

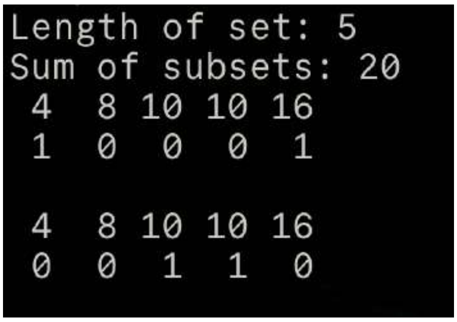
Code –



```
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <iostream>
using namespace std;

void write(int *a, int *w, int n, int k) {
  for (int i = 0; i < n; i++) {
    cout << setw(2) << w[i] << " ";
  }
  cout << "\n";
  for (int i = 0; i < n; i++) {
    if (i > k) {
      cout << setw(2) << 0 << " ";
    } else {
      cout << setw(2) << a[i] << " ";
    }
  }
  cout << "\n\n";
}

void backtrack(int *x, int k, int *w, int remainder, int reached, int target,
               int n) {
  x[k] = 1;
  if (reached + w[k] == target) {
    write(x, w, n, k);
    return;
  } else if (k + 1 < n && reached + w[k] + w[k + 1] > target) {
  } else {
    backtrack(x, k + 1, w, remainder - w[k], reached + w[k], target, n);
  }

  if ((k + 1 < n) && (reached + remainder - w[k] >= target) &&
      (reached + w[k + 1] <= target)) {
    x[k] = 0;
    backtrack(x, k + 1, w, remainder - w[k], reached, target, n);
  }
}
```



```
int main() {
  srandom(time(NULL));
  int n = random() % 20 + 1;
  n = 5;
  cout << "Length of set: " << n << endl;

  int *s = new int[n];
  int *a = new int[n];
  int sum = 0;
  int input[5] = {1, 2, 3, 4, 5};
  for (int i = 0; i < n; i++) {
    s[i] = random() % 20;
    // s[i] = input[i];
    sum += s[i];
    a[i] = 0;
  }

  sort(s, s + n);

  // int S = 6;
  int S = random() % 40;
  cout << "Sum of subsets: " << S << endl;

  backtrack(a, 0, s, sum, 0, S, n);

  return 0;
}
```



```
Length of set: 5
Sum of subsets: 20
 4  8 10 10 16
 1  0  0  0  1

 4  8 10 10 16
 0  0  1  1  0
```