Clemson Hack Pack

Clemson ACM

February 19, 2016

Copyright

10 Commandments of ACM Contests

Paraphrased from Dr. Dean

1. Thou shalt sort first and ask questions later
2. Thou shalt know the STL and use it well
3. Thou shalt know thy algorithms by heart
4. Thou shalt brute force $\leq$ 10 million items
5. Thou shalt when in doubt solve with DP
6. Thou shalt never count by 1
7. Thou shalt reinitialize thy data structures
8. Thou shalt test often and submit early
9. Thou shalt never trust a sample input
10. Thou shalt print according to the output spec

Remember what the Dr. said
"Algorithms are Cool!"

Contents

1. Data Structures

1.1. Set

Sets are data structures that are useful for determining if an element has been seen before or not. Sets are ordered collections of elements typically implemented as a balanced binary search tree. They have unique keys; that is to say that there are no duplicates in a set. However, unlike an array, elements are referenced by their ordering, not their position in the data structure.

See 'unordered_set' for a version of the set that does not have an order, but is based on hash tables. See 'multiset' for a version of the set that is not uniquely keyed. See 'unordered_multiset' for a version of the set that does not have an order and is also not uniquely keyed.

1.1.1 Reference

Listing 1.1: Set Reference

```
#include <set>
#include <iostream>
//Create a set of int
std::set<int> myset;

//iterator to a set of int
std::set<int>::iterator it;
std::set<int>::reverse_iterator rit;

int main(){

    //insert O(log N) per element or O(1) am per element for _sorted_ elements
    //For a total of O(N log N)  or O(N) for sorted inputs
    int key;
    for(key = 0; key < 10; key++){
        myset.insert(key);
    }

    //find O(log N)
    it = myset.find(3);

    //removes 3 in O(1) am post-find time
    myset.erase(it);
    //removes 4 from the set  O(log N) time
    myset.erase(4);

    //iterate the set in forward order O(1) am / O(log N)
    //for a total of O(N) total
    //Note that begin() returns an iterator to the first element
    //whereas that end() returns to a dummy element after the last element
    for(it = myset.begin(); it != myset.end(); it++){
        std::cout << *it << " " ;
    }
    std::cout << std::endl;

    //iterate the set in reverse order )O(1) am / O(log N)
    //for a total of O(N) total
    //Note that rbegin() returns an iterator to the last element
    //whereas that end() returns to a dummy element before the first element
    for(rit = myset.rbegin(); rit != myset.rend(); rit++){
        std::cout << *rit << " " ;
```

Continues on next page

Continued from previous page

```
    }
    std::cout << std::endl;

    //Find the first element greater than or equal to the current element in O(log N) time
    //In this case it returns 6
    it = myset.lower_bound(6);
    std::cout << *it << std::endl;

    //Find the first element greater than the current element in O(log N) time
    //In this case it returns 7
    it = myset.upper_bound(6);
    std::cout << *it << std::endl;

    // Empties the set O(N) time
    myset.clear();

}
```

1.1.2 Applications

- Determining how many and what items are in one set and also in another (intersection).
- Determining how many and what items are in one set but *not* in another (difference).
- Determining how many and what items are in either sets (union).
- Filtering out non-unique inputs.
- Can be useful for sweep line approaches

1.1.3 Example Contest Problem: Cow Pens

Farmer John's cows keep wandering off into the hills to learn cowculus from nomadic mathematicians. Unappreciative of refined bovine arithmetic, Farmer John decides to fence in his cows to keep them from escaping.

He wants to build the fence without crossing over any of the trees on his property (which the cows claim are valuable for studying graph theory), and he'd like the fence to be rectangular (a perfect shape, the cows say). One side of the rectangle should be formed by the river at the southern edge of Farmer John's property, so that the cows can contemplate wave-based trigonometric functions (and stay hydrated).

Please compute the maximum area that Farmer John can enclose with a fence that meets the above requirements. You can assume that the river has the position $Y = 0$ and Farmer John's property lies north of the river.

Input Format

- Line 1: One integer, N, $(N < 1000000)$, specifying the number of trees in the field.
- Lines 2..$(N + 1)$: Each line contains two integers X and Y $(0 <= X, Y <= 1000000)$. Each pair corresponds to the location of one tree in Farmer John's field.

Sample Input

Listing 1.2: Cow Pens Input

```
5
1 5
```

Continues on next page

Continued from previous page

```
2 4
3 3
4 2
5 1
```

Output Format

- Line 1: One integer that represents the area of the largest possible rectangle that Farmer John can build.

Sample Output

Listing 1.3: Cow Pens Output

```
999995000000
```

Example Solution

Listing 1.4: Cow Pens Solution

```
#include<algorithm>
#include<iostream>
#include<utility>
#include<set>
#include<vector>
using namespace std;

//Using longs because 1000000*1000000 > MAX_INT
const long MAXHEIGHT= 1000000;
typedef pair<long,long> pii;

//The default sorting of pii is v1.first < v2.first;
//SortOnY allows for sorting on ascending y values
bool SortOnY(pii v1 ,pii v2){return v1.second < v2.second;}
vector<pii> list;
set<pii> myset;

int main (){
  long N,x,y;
  cin >> N;
  for(long i = 0; i < N; i++){
    cin >> x >> y;
    list.push_back(make_pair(x,y));
  }

  //Sort the elements on increasing y values
  sort(list.begin(),list.end(),SortOnY);

  //initialize variables
  set<pii>::iterator before, after;
  long area = 0;

  //Some times it is better to make a change than code an edge case
  myset.insert(make_pair(0,0));
  myset.insert(make_pair(1000000,0));

  //Use a sweep line based on the set
  for(vector<pii>::iterator it = list.begin(); it != list.end(); it++){
    myset.insert(*it);
    before = myset.lower_bound(*it);
```

Continues on next page

Continued from previous page

```
    after = myset.upper_bound(*it);

    //lower_bound returns the pointer to the current element
    //decrement if it will be within bounds
    if(before != myset.begin()) before--;

    //compute the area of the box
    area = max(area,  it->second * (after->first - before->first));
  }


  //Check all of the plots that extend to the back of the plot
  after = myset.begin();
  for(before = myset.begin(); after != myset.end(); before++){
    after++;

    //compute the area of the long plots
    area = max(area, (after->first - before->first) * MAXHEIGHT);
  }

  //output according to the output spec
  cout << area << endl;
}
```

Lessons Learned

- The STL Set can be used as a basic binary search tree.
- Write comparison functions to change orderings of Sets, Maps, and the `sort()` function.
- `typedef` long type names to something shorter for ease of use.
- Sometimes it is better to *modify* the input than to code edge cases.
- `lower_bound` returns an iterator pointing to the first element $\leq$ the searched element.

1.1.4 Example Contest Problem: The Cows Form a Union

The cows have formed a union, and have gone on strike to protest Farmer John's new cow pen.

Each cow has been given a unique union ID number, painted on its side. The three cows with the smallest, closest ID numbers just so happen to be the union bosses.

Farmer John, in an effort to track union activity, took two photographs of cow rallies that he's sure the bosses attended, but he's not sure which cows are the bosses. Using the ID numbers of all the cows in the photographs, please determine the three cows that have the closest ID numbers so that Farmer John can attempt to negotiate with them. In case several sets of cows have equally close ID numbers, choose the set that contains the lowest ID number. You can assume that there are at least three cows between the two pictures.

Input Format

- Line 1: A space-delimited list, ending with 0, of N ID numbers $(0 \leq N \leq 1000000)$. Each number i_n $(0 < i_{0..N-1} \leq 1000000)$ indicates that a cow with that ID number is present in the *first* photograph.
- Line 2: A space-delimited list, ending with 0, of N ID numbers $(0 \leq N \leq 1000000)$. Each number i_n $(0 < i_{0..N-1} \leq 1000000)$ indicates that a cow with that ID number is present in the *second* photograph.

Sample Input

Listing 1.5: The Cows Form a Union Input

```
1 5 8 10 0
2 3 9 0
```

Output Format

- Line 1: Three space-delimited integers in ascending order, indicating the three cows who lead the union.

Sample Output

Listing 1.6: The Cows Form a Union Output

```
1 2 3
```

Example Solution

Listing 1.7: The Cows Form a Union Solution

```
#include<set>
#include<iostream>

using namespace std;

int main(){
    int i;
    set<int> s;

    //just inserting into the set will find a union because
    //it must be unique keyed
    while(cin >> i){
        //0 is not a valid input so ignore it
        if(i!=0)s.insert(i);
    }

    int min_dist,a,b, min_a, min_b, min_c;
    auto j = s.begin();

    //the first triple is a good candidate for the solution
    min_a = a = *(j++);
    min_b = b = *(j++);
    min_c = *j;
    min_dist = *j-a;

    //iterate over all triples
    for(;j != s.end(); j++){
        int dist = *j-a;
        if(dist < min_dist){
            //it it is an improvement, update the pair
            min_dist = dist;
            min_a = a;
            min_b = b;
            min_c = *j;
        }
        //update to the next entry
        a=b;
        b=*j;
    }
```

Continues on next page

Continued from previous page

```
    //Output the answer
    cout << min_a << " " << min_b << " " << min_c << endl;
}
```

1.1.5 Example Contest Problem: Cow Distances

As a result of the cows' lobbying efforts, the federal government is investigating Farmer John for possible violations of the Magnanimous Agricultural Defense of Cloven Ochlophobic Workers Statute, which states that any two cows of differing breeds (Farmer John owns Guernseys and Holsteins) must be given an inter-breed comfort zone of at least 1.000 meters. Cows of the same breed are allowed to mingle as cozily as they wish.

Assuming the regulators know the precise location of every cow in Farmer John's field, please help the federal government determine whether or not to crack down on Farmer John's gross oppression of his herd.

Input Format

- Line 1: One integer G, $(G < 500000)$, specifying the number of Guernseys to follow.
- Line 2..$(G + 1)$: Two integers X and Y, $0 \leq X, Y \leq 1000000$. Each pair corresponds to the location of one Guernsey in the field.
- Line $(G + 2)$: One integer, H, $(H < 500000)$, specifying the number of Holsteins to follow.
- Line $(G + 3)$..$(G + H + 2)$: Two integers X and Y, $0 \leq X, Y \leq 1000000$. Each pair corresponds to the location of one Holstein in the field.

Sample Input

Listing 1.8: Cow Distances Input

```
2
1 0
2 0
1
0 0
```

Output Format

- Line 1: the number 1 if Farmer John is breaking the law or 0 if he is not.

Sample Output

Listing 1.9: Cow Distances Output

```
1
```

1.2. Map

Maps are Associative, Ordered, Mapped, Uniquely Keyed, and Allocator aware.[5] Mapped means that each key corresponds to a specific value. Use it to record relationships in data.

Listing 1.10: Map Reference

```
#include <map>
std::map<key_type,value_type> mymap;

int main(){
    //insert O(N log(N)) or O(N) am for sorted inputs
    mymap.insert( std::pair<key_type, value_type>(key, value);
    mymap[key] = value;

    //iterator
    std::map<key_type, value_type>::iterator it;

    //remove O(log N) or O(1) am post-find
    mymap.erase(key);
    mymap.erase(it);

    //find O(log N)
    mymap.find(key)
}
```

1.3. Heap

Heaps are very useful data structures that support at least the following operations:

- Insert
- Either:
  - Remove the smallest element
  - Remove the largest element

Heaps that support removing the smallest element are called "Min Heaps" Heaps that support removing the largest element are called "Max Heaps" Many other implementations also implement a decrease key operation and delete operation.

The standard template library provides a two sets of functions that provide heaps. By default, both implementations are "Max Heaps" but can be made to be "Min Heaps". The first is the `priority_queue` data structure found in the queue header. The second is the `make_heap` functions in found the algorithm header. Neither of these implementations strictly implements the decrease key operation. The code sample shown below shows how to create a Binary Min Heap as well in $O(N)$ time as the Heap Sort algorithm which runs in $O(N \log N)$ time.

1.3.1 Reference

Listing 1.11: Heap Reference

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>

using namespace std;

static inline unsigned int parent(unsigned int i){return (floor(i-1)/2);}
```

Continues on next page

Continued from previous page

```
static inline unsigned int left(unsigned int i){return (2*i)+1;}
static inline unsigned int right(unsigned int i){return (2*i)+2;}

template <class T>
class binary_heap{
  typedef pair<int, T> heap_element;
  typedef typename vector<heap_element>::iterator heap_iterator;

  vector<heap_element> h;

  public:
  void insert(int key, T value){ // O(log n) time
    h.emplace_back(key, value);
    sift_up(h.size() - 1);
  }

  heap_element remove_min(){ // O(log n) time
    swap(h[h.size() - 1], h[0]);
    heap_element e = h.back();
    h.pop_back();
    sift_down(0);
    return e;
  }

  void decrease_key(int i , int delta){ // O(log n) time
    h[i].first -= delta;
    sift_up(i);
  }

  void delete_element(int i){ // O(log n) time
    swap(h[i], h[h.size() - 1]);
    sift_up(i);
    sift_down(i);
  }

  void sift_up(int i){ // O(log n) time
    while(i != 0 && h[i].first < h[parent(i)].first){
      swap(h[i],h[parent(i)]);
      i = parent(i);
    }
  }

  void sift_down(int i){ // O(log n) time
    while((left(i) < h.size() && h[i].first > h[left(i)].first) || (right(i) < h.size() && h[i].first > h[right(i)].first)){
      if (right(i) >= h.size() || h[left(i)].first < h[right(i)].first) { swap(h[left(i)],h[i]); i = left(i);}
      else { swap(h[right(i)],h[i]); i = right(i);}
    }
  }

  heap_iterator begin(){ // O(1) time
    return h.begin();
  }

  heap_iterator end(){ // O(1) time
    return h.end();
  }

  int size(){ // O(1) time
    return h.size();
  }

```

Continues on next page

Continued from previous page

```
  void make_heap(vector<heap_element>& v){ // O(N) time
    h=v;
    for(int i=v.size() - 1; i>0; i--) sift_down(i);
  }
};

int main() {

  binary_heap<int> h;
  int n;
  while(cin >> n) h.insert(n,0);

  while(h.size() > 0){
    cout << h.remove_min().first << ", ";
  }
  cout << endl;


  return 0;
}
```

1.3.2 Applications

- Dijkstra's Algorithm
- Prim's Algorithm
- Priority Based Queuing

1.4. Graph

Graphs are useful for a variety of different real world problems. Graphs are comprised of Nodes and Edges. Nodes represent a distinct state. Edges represent the possible transitions between states. Graphs can be directed(edges are one way) or undirected(edges are the same going to or from a node).

Paths are graphs that have no branches off. Cycles are graphs that connect back on them selves. Trees are graphs that contain no cycles. Forests are graphs that contain multiple disjoint trees. Bipartite Graphs have a set of "source" nodes and a set of "edge" nodes

Two nodes i, j are said to be connected if there is a path from i to j. Two nodes i, j are said to be strongly connected if there is a directed path from i to j and j to i.

1.4.1 Reference

Listing 1.12: Undirected Graph

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

typedef unordered_map<int,int> umii;
typedef unordered_map< int , umii > umiumii;
//The graph class uses c++11 extensions; use map to get O(log N) time with c++98
class sparse_graph{
  unordered_map < int, unordered_map< int, int> > graph;
  public:
    //inserts into the graph in O(1) am ex time
    void insert(int source, int destination, int edge){graph[source][destination] = edge;graph[
```

Continues on next page

Continued from previous page

```
        destination];}

    //inserts a specific edge in O(1) am ex time
    umii::iterator find(int source, int destination){
      return graph.find(source)->second.find(destination);
    }

    //returns an begin iterator for all edges leaving source; O(K) time to traverse
    umii::iterator begin(int source){ return graph.find(source)->second.begin(); }

    //returns an end iterator for all edges leaving source; O(K) time to traverse
    umii::iterator end(int source){ return graph.find(source)->second.end(); }

        //returns begin iterator over all of the nodes
        unordered_map<int, umii >::iterator begin () {return graph.begin();}

        //returns end iterator over all of the nodes
        umiumii::iterator end () {return graph.end();}

    //Constructs the sparse graph
    sparse_graph(): graph() {};
};
```

1.4.2 Applications

For all run times, V is the number of nodes, and E is the number of edges.

- Shortest Paths
  - Breath First Search — Graphs with equal weight edges $O(V + E)$
  - Dijkstra's Algorithm — Graphs with non-negative edges weights $O(E \log V)$.
  - Bellman Ford — Graphs with some negative edges $O(VE)$
  - Floyd Warshall — Graphs with negative edges but not cycles $O(V^3)$.
  - Dynamic Programming — Directed Acyclic Graphs various running times.
- Minimum Spanning Trees
  - Prim's Algorithm — For undirected graphs; if run for each component, finds the minimum spanning forest$O(E \log V)$.
  - Kruskal's Algorithm — For graphs; finds the minimum spanning forests in unconnected graphs $(E \log V)$
- Similarity/Connectivity
  - Depth First Search — Find (Strongly) Connected Components $O(V + E)$
  - Depth First Search — Find a path from i to j $O(V + E)$
- Topological Sorting
  - Depth First Search — Use start and stop times to topologically sort $O(V + E)$
- Matchings
  - Greedy — Many of these problems can be solved by a greedy max/min flow algorithm in $O(EV \log V \log F)$ where F is is the max flow.
- Flow/Routing

- Greedy — Many of these problems can be solved by a greedy max/min flow algorithm in $O(EV \log V \log F)$ where F is is the max flow.

- Clustering
- Centrality

1.4.3 Sample Contest Problem, The Cows Escape

Farmer John's Farm has become infested with Zombies! Farmer John being slightly out of shape wants to take the shortest amount of time to escape the zombies.

Farmer John's Farm is laid out in squares. Being prepared, Farmer John has his Early Zombie Detection System that will alert him to the rating of the zombies in these squares. Based off his research into the topic, Farmer John knows based on the rating how long it will take to evade the zombies in that sector.

Help Farmer John find the shortest time it will take Farmer John to escape his farm with his cows.

Input Format

- Line 1:One integer T indicating the number of test cases to evaluate
- Line 2:Three integers k, w, h representing the number zombie classes to follow and the width and height of Farmer John's Farm
- Lines 3..$(3+k)$: The letter representing the zombie class, it will not be 'F' and the time it will take to evade them, t
- Lines $(4+k)..(4+k+h)$: W capital letters representing the zombie classes in each sector of farmer John's farm. Farmer John's initial position is designated by 'F'.

Sample Input

Listing 1.13: The Cows Escape Sample Input

```
2
6 3 3
A 1
B 2
C 3
D 4
E 5
G 6
ABC
EFC
DBG
2 6 3
A 100
B 1000
BBBBBB
AAAAFB
BBBBBB
```

Output Format

- Line 1: 1 integer per line for each test case representing the minimum time it takes to escape the farm.

Sample Output

Listing 1.14: The Cows Escape Sample Output

```
2
400
```

Sample Solution

Listing 1.15: The Cows Escape Sample Solution

```
#include<iostream>
#include<iomanip>
#include<functional>
#include<unordered_map>
#include<queue>
#include<utility>
#include<limits>

using namespace std;

class graph{
    unordered_map< int , unordered_map< int, int> > g;
    public:
    void insert(int s, int d, int e){g[s][d] = e; g[d];}
    unordered_map<int,int>::iterator find(int s, int d){
        return g.find(s)->second.find(d);}
    unordered_map<int,int>::iterator begin (int s) {return g.find(s)->second.begin();}
    unordered_map<int,int>::iterator end (int s) {return g.find(s)->second.end();}
    graph(): g(){};
};
int dist[1000*1000+1]; //Array to hold the minimum distance to each grid
int farmMap[1001][1001]; //Buffer to hold the zombie locations while graph is built

int main(){
    int t;
    cin >> t;
    while(t--){
        int X,Y; //Used to hold farmer John’s location
        int k,w,h; //hold number of zombie classes, hight and width.
    //Read in the zombie classes
        unordered_map<char, int> zombie_classes;
        graph g;
        cin >> k >> w >> h;
        while(k--){
            char z; //zombie class name
            int t; // time it will take to escape them
            cin >> z >> t;
            zombie_classes[z] = t;
        }
    //Build the farmMap
        zombie_classes['F'] = 0; //Farmer John can escape the barn instantly
        for(int j=0; j<h; j++){
            for(int i=0; i < w; i++){
                char s;
                cin >> s;
                farmMap[i][j] =zombie_classes[s]; //fill the buffer with the time to pass through
                if (s == 'F')  {X=i;Y=j;} //When we find farmer john store his location
            }
        }
    //Build the graph
        for(int i=0; i < w; i++){
            for(int j=0; j < h; j++){
```

Continues on next page

Continued from previous page

```
                int id = i + w*j;
        //initialize the distance to node to the max int
                dist[id] = numeric_limits<int>::max();

        //if it does not run off the edge connect it
        //else map it to the dummy end node

                if(!(i+1 >= w) )
                    g.insert(id,id+1,farmMap[i+1][j]);
                else g.insert(id,w*h,0);

                if(!(i-1 < 0) )
                    g.insert(id,id-1,farmMap[i-1][j]);
                else g.insert(id,w*h,0);

                if(!(j+1 >= h))
                    g.insert(id,id+w,farmMap[i][j+1]);
                else g.insert(id,w*h,0);

                if(!(j-1 < 0))
                    g.insert(id,id-w,farmMap[i][j-1]);
                else g.insert(id,w*h,0);
            }
        }
    //initalize the dummy end node
        dist[w*h] = numeric_limits<int>::max();

    //This is how you make a min-queue of pair<int,int> in C++
        priority_queue<pair<int,int>, vector<pair<int,int> >, greater<pair<int, int> > > q;

        //dijkstra whohoo
        q.push(make_pair(0,X+w*Y));
        dist[X+w*Y]=0;
        while(!q.empty()){
            auto n = q.top();
            int d = n.first;
            int v = n.second;
            q.pop();
            if(d <= dist[v]){
               for(auto tmp = g.begin(v); tmp != g.end(v); tmp++){
                    if (dist[tmp->first] > d+tmp->second){
                        dist[tmp->first] = d+tmp->second;
                       q.push(make_pair((d+tmp->second), tmp->first));
                    }
               }
            }
        }
    //output the answer
        cout << dist[w*h] << endl;
    }
    return 0;
}
```

Lessons Learned

- Dijkstra can be solved using a priority queue using $O(N \log N)$ time.

1.5. Segment Tree

It is often useful to be aware of the range properties of a given array, such as: prefix sums, suffix sums, range minimum queries, range maximum queries, etc. For simpler problems, a secondary array may suffice to calculate a property like the prefix sum of an array. The prefix sum calculation only takes $O(n)$ time to perform, and any query thereafter takes only $O(1)$ time. But any updates to the source dataset will require $O(n)$ time to update the prefix sum calculations. Frequent changes will quickly show the pitfall of this approach. This precalculation approach will also not work for finding minimum or maximum values in an arbitrary range. A segment tree is a data structure for storing information about intervals that can be constructed in $O(n)$ time. Whenever the source data changes, update times stay low at $O(\log n)$ time.

9	5	4	3	8

29	14	15	9	5	4	11							3	8

Figure 1.1: The segment tree (bottom) constructed from an array (top) containing five elements. The blank entries in the segment tree array are unused.

A segment tree is represented as an array of a size that is dependent on the dataset it is sourced from. For a given array of size n, a segment tree constructed from it will use up to $2^{\lceil \log_2(n) \rceil + 1} - 1$ space. To properly represent the tree, the root is located at index 0; its left and right children are located at indices 1 and 2 respectively. To properly recurse through the tree, indices of left and right children can be calculated with $2n + 1$ and $2n + 2$ for left and right children respectively.

Construction of a segment tree begins at the root. The root node, located at index 0, queries its two children for some property (such as the maximum, the minimum, a range sum, etc...). Those two children, in turn, query their children. This process continues until a leaf node, one of the values from the source array, is reached. The recursive calls return back up the call stack with the parent nodes receiving the information they requested. These parent nodes record this information and continue returning up the call stack.

As the recursive calls are made, the starting and ending indices that each node represents is passed on. The left child of a node represents the first half of the section of the dataset the parent represents. The right child of a node represents the remaining right half. More precisely, if the parent node represents values between and including a starting index, a, and ending index b, then the left child will represent values between and including a and $a + \frac{b-a}{2}$. The right child will represent values between and including $a + \frac{b-a}{2} + 1$ and b. When a and b are equal, a leaf node has been reached, and its value is simply stored at the appropriate index in the segment tree. Due to the static nature of arrays, once built, the structure of a segment tree cannot change.

Listing 1.16: Segment Tree Reference Code

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

// dummy query returns; modify as necessary
#define ST_QUERY_DUMMY_MAX  99999999
#define ST_QUERY_DUMMY_MIN -99999999
#define ST_QUERY_DUMMY_SUM 0

// choose information to store
enum SegmentTreeType { ST_MIN, ST_MAX, ST_SUM };

class SegmentTree
{
```

Continues on next page

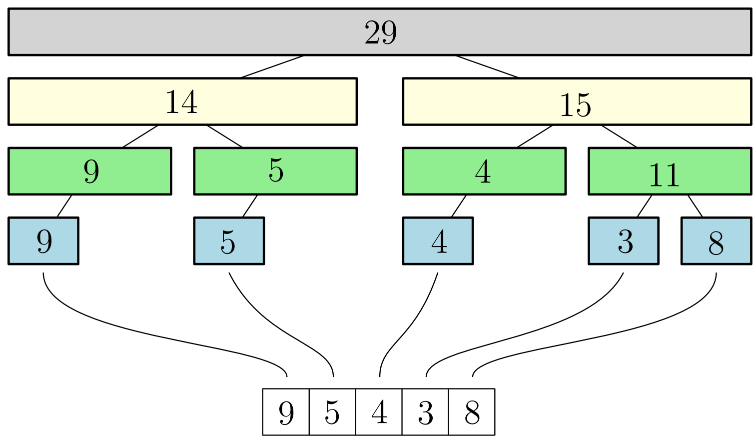

Figure 1.2: The segment tree visualized in a tree-like format. The leaves of the tree are values from the original dataset.

Continued from previous page

```
public:
  SegmentTree() : type(ST_MIN), size(0), tree(NULL) {}

  SegmentTree(SegmentTreeType type, const unsigned int size) : type(type), size(size)
  {
    tree = new int[size];
  }

  ~SegmentTree()
  {
    delete[] tree;
  }

  int build(int* const data, const unsigned int start, const unsigned int end, const unsigned int
      st_idx)
  {
    if(start == end)
    {
      tree[st_idx] = data[start];
      return data[start];
    }

    const unsigned int mid = start + ((end - start) / 2);
    const int left = build(data, start, mid, (2 * st_idx) + 1);
    const int right = build(data, mid + 1, end, (2 * st_idx) + 2);

    tree[st_idx] = helper(left, right);

    return tree[st_idx];
  }

  void update(const unsigned int start, const unsigned int end, const unsigned int changed_idx,
      const int old_val, const int new_val, const unsigned int st_idx)
  {
    if(changed_idx < start || changed_idx > end)
      return;

    if(type == ST_MIN)
    {
      if(new_val < tree[st_idx]  || start == end)
        tree[st_idx] = new_val;
    }
```

Continues on next page

Continued from previous page

```
    else if(type == ST_MAX)
    {
      if(new_val > tree[st_idx] || start == end)
        tree[st_idx] = new_val;
    }
    else if(type == ST_SUM)
    {
      int delta = new_val - old_val;
      tree[st_idx] += delta;
    }


    if(start == end)
      return;

    const unsigned int mid = start + ((end - start) / 2);
    update(start, mid, changed_idx, old_val, new_val, (2 * st_idx) + 1);
    update(mid + 1, end, changed_idx, old_val, new_val, (2 * st_idx) + 2);

    return;
  }

  int query(const unsigned int start, const unsigned int end, const unsigned int range_start, const unsigned int range_end, int val, const unsigned int st_idx)
  {
    if(start > range_end || end < range_start)
    {
      if(type == ST_MIN)
        return ST_QUERY_DUMMY_MAX;
      else if(type == ST_MAX)
        return ST_QUERY_DUMMY_MIN;
      else if(type == ST_SUM)
        return ST_QUERY_DUMMY_SUM;
    }

    if(start == end)
      return tree[st_idx];

    if(type == ST_MIN && (start >= range_start && end <= range_end))
    {
      if(tree[st_idx] < val)
        return tree[st_idx];
    }
    else if(type == ST_MAX && (start >= range_start && end <= range_end))
    {
      if(tree[st_idx] > val)
        return tree[st_idx];
    }
    else if(type == ST_SUM && (start >= range_start && end <= range_end))
    {
      return tree[st_idx];
    }
    else
    {
      const unsigned int mid = start + ((end - start) / 2);
      const int left = query(start, mid, range_start, range_end, val, (2 * st_idx) + 1);
      const int right = query(mid + 1, end, range_start, range_end, val, (2 * st_idx) + 2);
      val = helper(left, right);
    }

    return val;
  }

```

Continues on next page

Continued from previous page

```
  SegmentTreeType get_type()
  {
    return type;
  }

  unsigned int get_size()
  {
    return size;
  }

  int get_dummy_val()
  {
    if(type == ST_MIN)
      return ST_QUERY_DUMMY_MAX;
    else if(type == ST_MAX)
      return ST_QUERY_DUMMY_MIN;
    else if(type == ST_SUM)
      return ST_QUERY_DUMMY_SUM;
    else
      return -1;
  }

protected:
  SegmentTreeType type;
  unsigned int size;
  int* tree;

  int helper(const int a, const int b)
  {
    if(type == ST_MIN)
      return (a < b ? a : b);
    else if(type == ST_MAX)
      return (a > b ? a : b);
    else if(type == ST_SUM)
      return (a + b);
    else
      return -1;
  }
};

// number of entries in array
unsigned int N = 10;

int main()
{

  int src_array[N];

  // size needed for segtree is 2^{ceil(log2(n)) + 1} - 1
  const unsigned int st_size = pow(2, ceil(log2(N)) + 1) - 1;
  SegmentTree segtree(ST_MAX, st_size);
  segtree.build(src_array, 0, N - 1, 0);

  // update value in segtree
  int old = src_array[4];
  src_array[4] = 43;
  segtree.update(0, N - 1, 4, old, 43, 0);
  old = src_array[2];
  src_array[2] = 21;
  segtree.update(0, N - 1, 2, old, 21, 0);
  old = src_array[3];
  src_array[3] = 2;
  segtree.update(0, N - 1, 3, old, 2, 0);
```

Continues on next page

Continued from previous page

```
  old = src_array[9];
  src_array[9] = 1;
  segtree.update(0, N - 1, 9, old, 1, 0);

  // query segtree between indices 0 and 9, inclusive
  int result = segtree.query(0, N - 1, 0, 9, segtree.get_dummy_val(), 0);

  cout << result << endl;

  return 0;
}
```

Generally, a segment tree supports two operations: update and query. The update operation will account for changes to values in the source dataset. It is even possible to implement a range update operation to change an entire range of values. The query operation allows for fast retrieval of segment information. In the case of the prefix sum, the update operation will modify the tree's nodes such that it will contain correct information after changes, and the the query operation can return a sum for values in an arbitrary range.

1.5.1 Applications

- range sums in a frequently changing dataset
- range minimum/maximum queries

1.5.2 Example Contest Problem: Coming and Going

Farmer John is considering expanding the cows' barn, but he would first like to learn just how many of his cows spend any given time of day in the barn. In consideration of expanding the cows' barn, Farmer John decided to see just how many of his cows spent any given time of day in the barn. To do so, he installed motion sensors on each entryway/exit to the building. These sensors are able to detect both the entry and exit of a warm-blooded being. This, coupled with his cows' tags, allows him to monitor the movements of specific cows.

The night after the full day in which the sensors were installed, Farmer John sat down to do what he thought was some straightforward analysis, but quickly realized that he was in over his head with only mental math. He dusts off his old Commodore 64 and begins tabulating the data that he has collected. Among this data are the times that individual cows were inside the barn. *While* he enters these times into the computer for each cow, Farmer John wants to see how many cows were inside the barn during a specific time range.

Help him figure out how many cows were inside the barn during specific time ranges as he (slowly) tabulates the data.

Input Format

- Line 1: A single integer, C, representing the number of cows Farmer John has data on.
- The following lines are placed in C groups, each describing the movements of a single cow, detailed as follows:
  - A single line containing a single integer, k, representing the number of time ranges to follow that the cow was present in the barn.
  - k lines with two distinct integers representing times between which a cow was present in the barn. The first number is guaranteed to be strictly less than the second number.

- A single line containing two integers t_1 and t_2 representing times between which the maximum number of cows seen in the barn simultaneously should be reported since the last cow's data was added. As usual, with time ranges, the range t_1 to t_2 includes t_1 and all hours leading up to, but excluding t_2.

Sample Input

Listing 1.17: Coming and Going Sample Input

```
5

2
14 16
18 19
13 19

3
7 8
10 11
14 19
12 14

3
6 9
13 18
19 20
14 16

2
6 7
15 19
7 8

4
8 9
13 14
16 17
18 19
17 20
```

Output Format

C lines indicating the maximum number of cows seen in the barn in a single hour after adding data for the 1^{st}, 2^{nd}, ..., C^{th} cow. Each should appear on its own separate line formatted as given in the sample output.

Sample Output

Listing 1.18: Coming and Going Sample Output

```
1 of 5 cows
0 of 5 cows
3 of 5 cows
2 of 5 cows
4 of 5 cows
```

Sample Solution

Listing 1.19: Coming and Going Sample Solution

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

// number of entries in array
const unsigned int N = 14;

int build_segment_tree(int* const data, const unsigned int start, const unsigned int end, int* const
     tree, const unsigned int st_idx)
{
  // build the segment tree from source data
  //
  // data   - source array to build the segment tree from
  // start  - the start index of the array (should be 0 when first called)
  // end    - the end index of the array (size - 1)
  // tree   - the array to hold the segment tree, sized appropriately
  // st_idx - the current index of the segment tree (should be 0 when first called)
  // #ifdef hackpackpp

  // if start == end, this is a leaf node and should
  // have the original value from the array here
  if(start == end)
  {
    tree[st_idx] = data[start];
    return data[start];
  }

  // recurse into two calls to fill in the tree to the left
  // and right, splitting the array into the new range to be
  // represented by the new call
  const unsigned int mid = start + ((end - start) / 2);
  const int left = build_segment_tree(data, start, mid, tree, (2 * st_idx) + 1);
  const int right = build_segment_tree(data, mid + 1, end, tree, (2 * st_idx) + 2);

  // store the greatest value in the array in non-leaf nodes;
  // this can easily be changed to another property by storing
  // the sum, minimum of the subtrees at this node
  // sum:
  //   tree[st_idx] = left + right;
  //
  // minimum:
  //   if(left < right) tree[st_idx] = left;
  //   else tree[st_idx] = right;
  if(left > right) tree[st_idx] = left;
  else tree[st_idx] = right;

  return tree[st_idx];
}


void update_segment_tree(int* const tree, const unsigned int start, const unsigned int end, const
     unsigned int changed_idx, const int new_val, const unsigned int st_idx)
{
  // update a value in the segment tree
  //
  // tree        - the segment tree with values to be updated
  // start       - the start index of the _source_ array (should be 0)
  // end         - the end index of the _source_ array (size - 1)
  // changed_idx - the index of the element that changed
  // new_val     - the new value of the element at changed_idx
```

Continues on next page

Continued from previous page

```
  // st_idx      - the current index of the segment tree (should be 0 when first called)

  // out of range; should not be counted
  if(changed_idx < start || changed_idx > end)
    return;

  // update the value if the new value is greater or if
  // this is the leaf
  // if the tree were storing the range minimum, this would update
  // if(new_val < tree[st_idx] || start == end)
  if(new_val > tree[st_idx] || start == end)
    tree[st_idx] = new_val;

  // no need to recurse from leaf
  if(start == end)
    return;

  const unsigned int mid = start + ((end - start) / 2);
  update_segment_tree(tree, start, mid, changed_idx, new_val, (2 * st_idx) + 1);
  update_segment_tree(tree, mid + 1, end, changed_idx, new_val, (2 * st_idx) + 2);

  return;
}

int query_segment_tree(const int* const tree, const unsigned int start, const unsigned int end, const unsigned int range_start, const unsigned int range_end, int greatest, const unsigned int st_idx)
{
  // find the largest value for a range
  //
  // tree        - the segment tree to query
  // start       - the start index of the _source_ array
  // end         - the end index of the _source_ array
  // range_start - the start index of the query
  // range_end   - the end index of the query
  // greatest    - the greatest value found so far (set to zero when first calling)
  // st_idx      - the current index in the segtree (set to zero when first calling)

  // out of range; do not continue
  if(start > range_end || end < range_start)
    return -2;

  // leaf node; the greatest value one can see from here
  // is the value in the leaf node
  if(start == end)
    return tree[st_idx];

  // this node can be considered if the values in the original
  // array it represents fit in the query's range
  if(start >= range_start && end <= range_end)
  {
    if(tree[st_idx] > greatest)
      return tree[st_idx];
  }
  else
  {
    const unsigned int mid = start + ((end - start) / 2);
    const int left = query_segment_tree(tree, start, mid, range_start, range_end, greatest, (2 * st_idx) + 1);
    const int right = query_segment_tree(tree, mid + 1, end, range_start, range_end, greatest, (2 * st_idx) + 2);

    // use the greater of the two values here;
```

Continues on next page

Continued from previous page

```
    // if a summation were desired, one could simply add the values together;
    // for a minimum, simply take the lesser of the two values
    if(left > greatest)
      greatest = left;
    if(right > greatest)
      greatest = right;
  }

  return greatest;
}

#define START_OF_DAY 6

int main()
{

  int headcount[N] = { 0 };

  // size needed for segtree is 2^{ceil(log2(n)) + 1} - 1
  const unsigned int st_size = pow(2, ceil(log2(N)) + 1) - 1;
  int* segtree = new int[st_size];
  build_segment_tree(headcount, 0, N - 1, segtree, 0);

  unsigned int no_cows;
  cin >> no_cows;

  for(unsigned int i = 0; i < no_cows; i++)
  {
    unsigned int in_barn_times;
    cin >> in_barn_times;

    for(unsigned int j = 0; j < in_barn_times; j++)
    {
      unsigned int start, end;
      cin >> start >> end;
      for(unsigned int k = start; k < end; k++)
      {
        // account for times 0 to 5 we don't care about with k - 6
        headcount[k - START_OF_DAY]++;
        update_segment_tree(segtree, 0, N - 1, k - START_OF_DAY, headcount[k - START_OF_DAY], 0);
      }
    }

    int t1, t2;
    cin >> t1 >> t2;

    // print out cow count as data comes in
    const int cows_in_barn = query_segment_tree(segtree, 0, N - 1, t1 - START_OF_DAY, t2 -
         START_OF_DAY - 1, -1, 0);
    if(cows_in_barn > 0)
      cout << cows_in_barn;
    else
      cout << 0;
    cout << " of " << no_cows << " cows" << endl;
  }

  delete[] segtree;

  return 0;
}
```

Lessons Learned

- The data the segment tree keeps track of can easily be changed with a few small modifications to the build, update, and query operations.

1.5.3 USACO Contest Problem: Optimal Milking[3]

Farmer John has recently purchased a new barn containing N milking machines ($1 \leq N \leq 40,000$), conveniently numbered $1..N$ and arranged in a row.

Milking machine i is capable of extracting $M(i)$ units of milk per day ($1 \leq M(i) \leq 100,000$). Unfortunately, the machines were installed so close together that if a machine i is in use on a particular day, its two neighboring machines cannot be used that day (endpoint machines have only one neighbor, of course). Farmer John is free to select different subsets of machines to operate on different days.

Farmer John is interested in computing the maximum amount of milk he can extract over a series of D days ($1 \leq D \leq 50,000$). At the beginning of each day, he has enough time to perform maintenance on one selected milking machine i, thereby changing its daily milk output $M(i)$ from that day forward. Given a list of these daily modifications, please tell Farmer John how much milk he can produce over D days (note that this number might not fit into a 32-bit integer).

Input Format

- Line 1: The values of N and D.
- Line $2..1+N$: Line $i+1$ contains the initial value of $M(i)$.
- Lines $2+N..1+N+d$: Line $1+N+d$ contains two integers, i and m, indicating that Farmer John updates the value of $M(i)$ to m at the beginning of day d.

Sample Input

Listing 1.20: Optimal Milking Sample Output

```
5 3
1
2
3
4
5
5 2
2 7
1 10
```

Output Format

- Line 1: The maximum total amount of milk FJ can produce over D days.

Sample Output

Listing 1.21: Optimal Milking Sample Output

```
5 3
1
2
3
4
5
5 2
```

Continues on next page

Continued from previous page

```
2 7
1 10
```

2. Algorithms

2.1. Dijkstra's Algorithm

Dijkstra's Algorithm solves the single-source shortest path problem of finding the shortest paths between the source node and all other nodes in a connected graph with non-negative edge path costs. The graph can be both directed and undirected. It is commonly used to find the shortest path between a source and destination node. For graphs with negative weights, see Bellman-Ford Algorithm. It is commonly inplemented using a priority queue and runs in $O(E \log V)$.

2.1.1 Applications

- Finding the shortest paths between a source node and all other nodes in a connected graph with non-negative edge path costs.

2.1.2 Example Contest Problem: Farm Tour[9]

When Farmer John's friends visit him on the farm, he likes to show them around. His farm comprises N $(1 \leq N \leq 1000)$ fields numbered $1..N$, the first of which contains his house and the Nth of which contains the big barn. A total M $(1 \leq M \leq 10000)$ paths that connect the fields in various ways. Each path connects two different fields and has a nonzero length smaller than $35,000$.

To show off his farm in the best way, he walks a tour that starts at his house, potentially travels through some fields, and ends at the barn. Later, he returns (potentially through some fields) back to his house again.

He wants his tour to be as short as possible, however he doesn't want to walk on any given path more than once. Calculate the shortest tour possible. Farmer John is sure that some tour exists for any given farm.

Input Format

- Line 1: Two space-separated integers: N and M.
- Lines $2..M+1$: Three space-separated integers that define a path: The starting field, the end field, and the path's length.

Sample Input

Listing 2.1: Farm Tour Input

```
4 5
1 2 1
2 3 1
3 4 1
1 3 2
2 4 2
```

Output Format

- Line 1: A single line containing the length of the shortest tour.

Sample Output

Listing 2.2: Farm Tour Output

```
6
```

2.1.3 Example Contest Problem: Milk Routing[1]

Farmer John's farm has an outdated network of M pipes $(1 \leq M \leq 500)$ for pumping milk from the barn to his milk storage tank. He wants to remove and update most of these over the next year, but he wants to leave exactly one path worth of pipes intact, so that he can still pump milk from the barn to the storage tank.

The pipe network is described by N junction points $(1 \leq N \leq 500)$, each of which can serve as the endpoint of a set of pipes. Junction point 1 is the barn, and junction point N is the storage tank. Each of the M bi-directional pipes runs between a pair of junction points, and has an associated latency (the amount of time it takes milk to reach one end of the pipe from the other) and capacity (the amount of milk per unit time that can be pumped through the pipe in steady state). Multiple pipes can connect between the same pair of junction points.

For a path of pipes connecting from the barn to the tank, the latency of the path is the sum of the latencies of the pipes along the path, and the capacity of the path is the minimum of the capacities of the pipes along the path (since this is the "bottleneck" constraining the overall rate at which milk can be pumped through the path). If Farmer John wants to send a total of X units of milk through a path of pipes with latency L and capacity C, the time this takes is therefore $L + X/C$.

Given the structure of Farmer John's pipe network, please help him select a single path from the barn to the storage tank that will allow him to pump X units of milk in a minimum amount of total time.

Official Solution: `http://www.usaco.org/current/current/data/sol_mroute.html`

Input Format

- Line 1: Three space-seperated integers: $N\ M\ X\ (1 \leq X \leq 1{,}000{,}000)$.
- Line $2..1 + M$: Each line describes a pipe using 4 integers: $I\ J\ L\ C$. I and J $(1 \leq I, J \leq N)$ are the juntion points at both ends of the pipe. L and C $(1 \leq L, C \leq 1{,}000{,}000)$ give the latency and capacity of the pipe.

Sample Input

Listing 2.3: Milk Routing Input

```
3 3 15
1 2 10 3
3 2 10 2
1 3 14 1
```

Output Format

- Line 1: The minimum amount of time it will take Farmer John to send milk along a single path, rounded down to the nearest integer.

Sample Output

Listing 2.4: Milk Routing Output

Continues on next page

Continued from previous page

```
27
```

2.1.4 Example Contest Problem: Dueling GPS's[4]

Farmer John has recently purchased a new car online, but in his haste he accidentally clicked the "Submit" button twice when selecting extra features for the car, and as a result the car ended up equipped with two GPS navigation systems! Even worse, the two systems often make conflicting decisions about the route that Farmer John should take.

The map of the region in which Farmer John lives consists of N intersections $(2 \leq N \leq 10,000)$ and M directional roads $(1 \leq M \leq 50,000)$. Road i connects intersections A_i $(1 \leq A_i \leq N)$ and B_i $(1 \leq B_i \leq N)$. Multiple roads could connect the same pair of intersections, and a bi-directional road (one permitting two-way travel) is represented by two separate directional roads in opposite orientations. Farmer John's house is located at intersection 1, and his farm is located at intersection N. It is possible to reach the farm from his house by traveling along a series of directional roads.

Both GPS units are using the same underlying map as described above; however, they have different notions for the travel time along each road. Road i takes P_i units of time to traverse according to the first GPS unit, and Q_i units of time to traverse according to the second unit (each travel time is an integer in the range $1..100,000$).

Farmer John wants to travel from his house to the farm. However, each GPS unit complains loudly any time Farmer John follows a road (say, from intersection X to intersection Y) that the GPS unit believes not to be part of a shortest route from X to the farm (it is even possible that both GPS units can complain, if Farmer John takes a road that neither unit likes).

Please help Farmer John determine the minimum possible number of total complaints he can receive if he chooses his route appropriately. If both GPS units complain when Farmer John follows a road, this counts as $+2$ towards the total.

Input Format

- Line 1: The integers N and M.
- Line i describes road i with four integers: A_i B_i P_i Q_i.

Sample Input

Listing 2.5: Farm Tour Input

```
5 7
3 4 7 1
1 3 2 20
1 4 17 18
4 5 25 3
1 2 10 1
3 5 4 14
2 4 6 5
```

Output Format

- Line 1: The minimum total number of complaints Farmer John can receive if he routes himself from his house to the farm optimally.

Sample Output

Listing 2.6: Dueling GPSs Output

```
1
```

Lessons Learned

- Trickier problems may require adjusting the graph or, in this case, creating a new one
- Numbering the nodes starting with 0 instead of 1 allows for mapping with vector indeces

2.2. Sieve of Eratosthenes

The sieve of Eratosthenes is a simple, yet effective algorithm for generating primes less than around 10 million. It works by iteratively eliminating (”sifting”) multiples of primes. Numbers that are left are prime. The algorithm runs in $O(n \log \log n)$ time and requires $O(n)$ memory to generate all primes up to n.

2.2.1 Applications

- Finding prime numbers below ~10 million

2.2.2 Example Contest Problem: All or Nothing

After many hard days and nights, Farmer John has completed the construction of a larger barn to house his cows. Though both parties would like to begin migration to the new barn, the cows want to perform the migration in a fair manner so that no cow gets to enjoy the new barn before another.

Each day, Farmer John plans to move an equal number of cows over. He is not willing to do any *more* work one day, nor any *less* work another day, and he is *certainly* not willing to move a single cow a day. The cows, after consulting amongst themselves, have decided to trick Farmer John into thinking that he has a prime number of cows. This would, in turn, force him to move all of the cows at once. Currently, they are exploring their options of how many different prime head counts they could give Farmer John.

Find out the number of possible prime head counts the cows can give Farmer John such that he is forced to move all of the cows to the new barn early tomorrow.

Input

- Line 1: Two integers A, B,$(1 \leq A < B \leq 10000000)$, separated by spaces indicating head counts.

Sample Input

Listing 2.7: All or Nothing Input

```
2 5
```

Output

- Line 1: One integer representing the number of head counts between A and B inclusive where Farmer John has to move all the cows at once.

Sample Output

Listing 2.8: All or Nothing Output

```
3
```

Example Solution

Listing 2.9: All or Nothing Solution

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

const unsigned int N = 10000001;
bitset<N> sieve;
unsigned int prefix[N];

int main()
{
  unsigned int a, b;
  cin >> a >> b;

  // Begin at index 1. Starting the sift at index 0 (1) will
  // cause all numbers to be marked composite.
  sieve[0] = 1; // 0 is not prime.
  sieve[1] = 1; // 1 is not prime.
  for(unsigned int i = 1; i < N; i++)
  {
    // If this number has been marked, do not use it.
    if(sieve[i]) continue;
    // Mark composites of the prime.
    for(unsigned int j = (i * i); j <= N; j += i)
      sieve[j] = 1;
  }

  // Compute prefix sum of the number of primes.
  unsigned int count = 0;
  for(unsigned int i = 0; i < N; i++)
  {
    if (!sieve[i]) count++;
    prefix[i] = count;
  }

  // Remaining numbers != 0 are prime.
  if(a < 2) cout << prefix[b];
  else cout << prefix[b] - prefix[a - 1];
  cout << endl;

  return 0;
}
```

Lessons Learned

- The sieve is a simple tool for finding primes $< 10,000,000$.
- Requires a sequence of at least size N where N is equal to the upper bound (can be made more efficient by excluding even numbers).
- Aligning the sequence of numbers with the array indices eliminates quite a bit of ± 1 confusion, leading to cleaner code

2.2.3 ACM Contest Problem: Ping![7]

Suppose you are tracking some satellites. Each satellite broadcasts a ’ping’ at a regular interval, and the intervals are unique (that is, no two satellites ping at the same interval). You need to know which satellites you can hear from your current position. The problem is that the pings cancel each other out. If an even number of satellites ping at a given time, you won’t hear anything, and if an odd number ping at a given time, it sounds like a single ping. All of the satellites ping at time 0, and then each pings regularly at its unique interval.

Given a sequence of pings and non-pings, starting at time 0, which satellites can you determine that you can hear from where you are? The sequence you’re given may, or may not, be long enough to include all of the satellites’ ping intervals. There may be satellites that ping at time 0, but the sequence isn’t long enough for you to hear their next ping. You don’t have enough information to report about these satellites. Just report about the ones with an interval short enough to be in the sequence of pings.

Input

- There will be several test cases in the input.
- Each test case will consist of a single string on its own line, with from 2 to 1,000 characters. The first character represents time 0, the next represents time 1, and so on.
- Each character will either be a 0 or a 1, indicating whether or not a ping can be heard at that time (0 = No, 1 = Yes).
- Each input is guaranteed to have at least one satellite that can be heard.
- The input will end with a line with a single 0.

Sample Input

Listing 2.10: Ping! Input

```
01000101101000
1001000101001000
0
```

Output

- For each test case, output a list of integers on a single line, indicating the intervals of the satellites that you know you can hear.
- Output the intervals in order from smallest to largest, with a single space between them.
- Output no extra spaces, and do not separate answers with blank lines.

Sample Output

Listing 2.11: Ping! Output

```
1 2 3 6 8 10 11 13
3 6 7 12 14 15
```

2.3. Knuth-Morris-Pratt String Matching

This algorithm is a method that improves upon string searches by using information about the keyword itself to determine where a failed search should continue. Prior to beginning a search, a table of values is computed. In this table (called the partial match table) are the lengths of the longest proper prefixes that match the longest proper suffixes up to the given permutation of characters. They also determine the number of indices the algorithm should advance should the very **next** character match fail. Because these prefixes and suffixes match, and the prefix is always the first characters of the keyword, the positions of the suffixes are where the algorithm can begin yet another matching sequence.

For example, let us consider the string 'abababcd'. Throughout the construction of the table, we consider the first N characters of the string to yield the substring we want to analyze.

$N = 1$ **'a'** This substring contains only one character and can contain a neither a proper prefix nor a proper suffix, therefore, we set the first index to 0.

$N = 2$ **'ab'** There is only one proper prefix ('a') and one proper suffix ('b'), and they do not match, therefore, this one is set to 0 as well.

$N = 3$ **'aba'** Now, we have two prefixes, 'a' and 'ab', and two suffixes, 'a' and 'ba'. While, 'ab' and 'ba' do not match, 'a' and 'a' do. So, this time, we can set the value to 1, because that is the length of the longest match. Now, upon failing to match the next character, the algorithm will refer to this value and jump to matching this character by simply subtracting this number from the sum of the length of the partial match and the starting index of the partial match.

$N = 4$ **'abab'** Prefixes: 'a', 'ab', and 'aba'. Suffixes: 'b', 'ab', and 'bab'. Perusing the substrings in decreasing length, 'ab' provides a match. The value is set to 2.

$N = 5$ **'ababa'** Prefixes: 'a', 'ab', 'aba', and 'abab'. Suffixes: 'a', 'ba', 'aba', and 'baba'. The longest match here is 'aba', with a length of 3.

$N = 6$ **'ababab'** Prefixes: 'a', 'ab', 'aba', 'abab', and, 'ababa'. Suffixes: 'b', 'ab', 'bab', 'abab', and, 'babab'. This time, the longest match is 'abab', so we set a value of 4.

$N = 7$ **'abababc'** Prefixes: 'a', 'ab', 'aba', 'abab', 'ababa', and 'ababab'. Suffixes: 'c', 'bc', 'abc', 'babc', 'ababc', and 'bababc'. In this case, there are no matches, so the value is zero.

$N = 8$ **'abababcd'** Prefixes: 'a', 'ab', 'aba', 'abab', 'ababa', 'ababab', and 'abababc' Suffixes: 'd', 'cd', 'bcd', 'abcd', 'babcd', 'ababcd', and 'bababcd' There are no matching substrings; the value here is zero.

i	0	1	2	3	4	5	6	7
W[i]	a	b	a	b	a	b	c	d
T[i]	0	0	1	2	3	4	0	0

Table 2.1: The computed partial match table for the string. W is the string for which the table is computed and T is the partial match table itself.

As mismatches occur, and by consulting this table of values, the KMP algorithm dictates where to resume the search for the desired keyword again. Suppose that the algorithm is currently matching against the string 'ababaccabcdefg'. Immediately, it will attempt to match the first eight characters against our chosen keyword 'abababcd'. Of course, during this process, it will realize that this is a mismatch when matching the string's 6$^{\text{th}}$ character ('c') against the keyword's (b). Instead of resuming the search at the second character of the string, the algorithm consults the table. If, during the search, the algorithm passed by the start of another possible match, the table can tell exactly where the start of that match begins at relative to what index the initial mismatch occurred.

In this case, our partial match got as far as five characters before encountering a problem. Using the table, we discover how far the matching should backtrack with $P[l-1]$ where P is the partial match table and l is the length of the partial match. In this case, the search resumes the matching process after moving $l - P[l-1]$ characters from where the match started, where l is the length of the partial match

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td></tr>
<tr><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>c</td><td>c</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td></tr>
<tr><td>|</td><td>|</td><td>|</td><td>|</td><td>|</td><td>X</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>b</td><td>c</td><td>d</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

and P is the partial match table. So, the algorithm will move its search $5-P[5-1]=5-P[4]=5-3=2$ characters from where it is currently.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td></tr>
<tr><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>c</td><td>c</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td></tr>
<tr><td></td><td></td><td>|</td><td>|</td><td>|</td><td>X</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>b</td><td>c</td><td>d</td><td></td><td></td><td></td><td></td></tr>
</table>

Matching resumes at index 2 and this time fails after getting a partial match length of 3. Therefore, we advance by $3-P[3-1]=3-1=2$ characters.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td></tr>
<tr><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>c</td><td>c</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td></tr>
<tr><td></td><td></td><td></td><td></td><td>|</td><td>X</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>b</td><td>c</td><td>d</td><td></td><td></td></tr>
</table>

The match soon fails after only obtaining a partial match length of one. Because $1-P[1-1]=0$, this means we move forward only as far as the partial match extended.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td></tr>
<tr><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>c</td><td>c</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td>X</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>b</td><td>c</td><td>d</td><td></td></tr>
</table>

No matches here.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td></tr>
<tr><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>c</td><td>c</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td>X</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td>a</td><td>b</td><td>a</td><td>b</td><td>a</td><td>b</td><td>c</td><td>d</td></tr>
</table>

Finally, there is no match at this point here. Because the rest of the text is shorter than the search query, we stop searching. We have determined that the keyword is not present.

The KMP algorithm (as it is commonly known as) has two distinct parts. The first, constructing the partial match table, as exemplified above, requires $O(m)$ time. The second, the actual string matching portion, takes $O(n)$ time. Therefore, the running time of the algorithm can be described as $O(m+n)$.

2.3.1 Applications

- More efficient string searches.

2.3.2 Example Contest Problem: The Fine Print

Due to his excessive milking of the cows without appropriate compensation, Farmer John has, unsurprisingly, received an ultimatum from the cows. If the two parties cannot come to an agreement, Farmer

John risks internal insurgency. Though he is willing to reduce his demands and compensate them with more grazing time, the document he has received is unbearably lengthy.

Farmer John can recall that, lately, the cows have been nagging him to build a swimming pool. Therefore, it is likely that a condition has been added to force him to concede to building this pool.

To save Farmer John from a long night (he works early mornings) find out if anything about a 'pool' has been added anywhere.

Input

- Line 1: Text from standard input representing the legal document terminated with an EOF.

Sample Input

Listing 2.12: The Fine Print Input

```
Farmer John,

Cras sit amet mauris. Curabitur a quam. Aliquam neque. Nam nunc nunc, lacinia
sed, varius quis, iaculis eget, ante. Nulla dictum justo eu lacus. Phasellus
sit amet quam. Nullam sodales. Cras non magna eu est consectetuer
fauci. Compensation for excess labor MUST include an olympic-size swimming pool. Sed
tellus velit, ullamcorper ac, fringilla vitae, sodales nec, purus. Morbi
aliquet risus in mi.

We hope an agreement can be reached.

The Cows
```

Output

- Line 1:
  - The sentence containing 'pool' if it exists. All sentences within the text end in a period.
  - The string "The agreement does not mention a pool." if a sentence containing 'pool' doesn't exist.

Sample Output

Listing 2.13: The Fine Print Output

```
Compensation for excess labor MUST include an olympic-size swimming pool.
```

Example Solution

Listing 2.14: The Fine Print Solution

```
#include <fstream>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

vector<unsigned int> match_idxs;

// Create the partial match table.
int const* build_table(const string& s)
```

Continues on next page

Continued from previous page

```
{
  int* const table = new int[s.length()]();
  fill(table, table + s.length(), 0);
  for(unsigned int str_sz = 1; str_sz <= s.length(); str_sz++)
  {
    if(str_sz == 1) continue;
    for(int curr_len = str_sz - 1; curr_len > 0; curr_len--)
    {
      // Take the first and last 'curr_len' characters.
      string prefix = s.substr(0, curr_len);
      string suffix = s.substr(str_sz - curr_len, curr_len);

      // Keep the length of longest matching prefix and suffix.
      if(prefix == suffix) { table[str_sz - 1] = curr_len; break; }
    }
  }

  return table;
}

int main()
{
  // Read input.
  string text = "", buffer = "";
  while(!cin.eof()) { getline(cin, buffer); text += buffer; }

  // KMP algorithm to find keyword.
  const string keyword = "pool";
  int const* const pm_table = build_table(keyword);
  for(unsigned int i = 0; i <= text.length() - keyword.length() && text.length() >= keyword.length()
       ; i++)
  {
    // Consider a possible match
    if(text[i] == keyword[0])
    {
      for(unsigned int m = 1; m <= keyword.length(); m++)
      {
        // Complete match
        if(m == keyword.length()) { match_idxs.push_back(i); break; }

        // Mismatch
        if(text[i + m] != keyword[m])
        {
          // Consult table to see where to start 'i' at.
          i += m - pm_table[m - 1];

          // It will be necessary to negate the next loop's i++.
          i--;
          break;
        }
      }
    }
  }

  if(match_idxs.size() == 0) cout << "The agreement does not mention a pool." << endl;
  else // Pick out the sentence.
  {
    unsigned int beg = match_idxs[0];
    unsigned int end = match_idxs[0] + keyword.length();
    while(text[beg - 2] != '.' && beg > 0) beg--;
    while(text[end] != '.') end++;
    cout << text.substr(beg, end - beg + 1) << endl;
  }
```

Continues on next page

Continued from previous page

```
  return 0;
}
```

Lessons Learned

- $O(m)$ is needed to build the partial match table.
- Just the partial match table can be a useful addition when solving problems that involve finding partial matches themselves.

2.3.3 Example Contest Problem: DNA Splicing

The Nobonez alien race has descended upon Farmer John's beloved cows! Rather than abducting them though, they have begun to experiment on them genetically, dabbling with their DNA. With the help of the local geneticist, Farmer John can save all of his cows. To do so, he must locate all occurrences of changed DNA.

DNA sequences are composed of different combinations of nucleotides, abbreviated as 'A', 'T', 'C', and 'G'. After careful analysis, the geneticist has concluded that only one specific pattern of DNA has been changed from its original sequence. And, fortunately, the Nobonez have only experimented with changing no more than two nucleotides at a time.

Find this corrupted DNA to successfully save all of Farmer John's cows.

Input

- Line 1: Text from standard input representing the original subsequence of DNA that was targeted.
- Line 2: Text from standard input representing the cow's DNA sequence after the modification.

Sample Input

Listing 2.15: DNA Splicing Input

```
GCCGTTCCGCGC
ATTCGTATTAATATATTGCCGTTACGCGCTATAGCTGCAGGAATATATTGCCGTGCCGCGCAGGAACA
```

Output

- With each occurrence on its own line, in the following order: the index of the modification in the string representation, what it should be, and what the nucleotide was changed to, as formatted below.

Sample Output

Listing 2.16: DNA Splicing Output

```
23: C changed to A
54: T changed to G
```

Lessons Learned

- Besides using the KMP string matching algorithm, another common way of approaching string matching is hashing.

2.3.4 ACM Contest Problem: Tandem Repeats[7]

Tandem repeats occur in DNA when a pattern of one or more nucleotides is repeated, and the repetitions are directly adjacent to each other. For example, consider the sequence:

ATTCGATTCGATTCG
This contains nine tandem repeats:
ATTCGATTCGATTCG
ATTCGATTCGATTCG
ATTCGATTCGATTCG
ATTCGATTCGATTCG
ATTCGATTCGATTCG
ATTCGATTCGATTCG
ATTCGATTCGATTCG
ATTCGATTCGATTCG
ATTCGATTCGATTCG

Given a nucleotide sequence, how many tandem repeats occur in it?

Input

- There will be several test cases in the input. Each test case will consist of a single string on its own line, with 1 to 100,000 capital letters, consisting only of A, G, T, and C.
- This represents a nucleotide sequence. The input will end with a line with a single 0.

Sample Input

Listing 2.17: Tandem Repeats Input

```
AGGA
AGAG
ATTCGATTCGATTCG
0
```

Output

- For each test case, output a single integer on its own line, indicating the number of tandem repeats in the nucleotide sequence.
- Output no spaces, and do not separate answers with blank lines.

Sample Output

Listing 2.18: Tandem Repeats Output

```
1
1
9
```

2.4. Computational Geometry

Basic geometric algorithms are an essential part of many programs. These algorithms are provided for quick transcription to code.

2.4.1 Cross Product

The cross product is an operation on 3-dimensional vectors that finds a perpendicular vector.

Listing 2.19: Cross Product

```
#include<iostream>
void cross_product(double A[3], double B[3], double cross[3]){
  //Input one and two, then output, all taken to be size-3 double arrays
  cross[0] = ((A[1]*B[2])-(A[2]*B[1]));
  cross[1] = ((A[2]*B[0])-(A[0]*B[2]));
  cross[2] = ((A[0]*B[1])-(A[1]*B[0]));//The cross vector is equal to the cross product of A and B
}
```

2.4.2 Dot Product

The dot product is a vector operation that takes two vectors of equal length and returns the sum of the corresponding elements. For example, [1, 2] dot [3, 4] is 1*3 + 2*4, or 11. The dot product is alternately equal to the product of the vectors times the cosine of the angle between them.

Listing 2.20: Dot Product

```
#include<iostream>
#include<vector>

double dot_product(double *v, double *v2, int len) {//Can convert all ints to doubles
  double result=0;
  for(int c=0; c < len; c++) result += (v[c] * v2[c]);
  return result;
}
```

2.4.3 Arctangent

The arctangent function takes the ratio between the opposite and adjacent sides of a right triangle and returns the angle (between -pi/2 and pi/2 (or -tau/4 and tau/4)). This will need to be corrected if the answer is required to be in quadrant 2 or 3. The other trigonometric functions are included in cmath as well. These do use radians, so convert to degrees by multiplying by 180/pi if necessary.

Listing 2.21: Arctangent

```
#include<iostream>
#include <cmath>    /* atan */
#define PI 3.14159265

int main() {
  double param, result;
  param = 1.0;
  result = atan (param) * 180 / PI; //Remove the " * 180 / PI" to convert to degrees.
  fprintf(stdout, "The arc tangent of %.1f is %.1f degrees\n", param, result );
  return 0;
}
```

2.4.4 Area of Triangle

The area of a triangle given three points is most easily computed by taking half the absolute value of the determinant of two of its rows, as done here. This could also be computed via length*height/2 or Heron's formula, which takes is the square root of the product of the semiperimeter and the semiperimeter minus each side.

Listing 2.22: Area of Triangle

```
#include<iostream>
#include<cmath>

int main(){
  double p1[2]={0,0}, p2[2]={0,3}, p3[2]={4,0}; //Three 2d points to measure the area of the triangle between
  double v1[2], v2[2];
  v1[0] = p2[0]-p1[0];
  v2[0] = p3[0]-p1[0];
  v1[1] = p2[1]-p1[1];
  v2[1] = p3[1]-p1[1];

  int det = (v1[0]*v2[1]) - (v1[1]*v2[0]);
  int area = det/2;
}
```

2.4.5 Area of Polygon

This function returns the area of the polygon defined by the input list of points. It does work for concave polygons, though not self-intersecting or self-crossing polygons- they must be able to be traced by a non-intersecting line. This algorithm adds the area between the segment and the y axis if the segment goes up, otherwise it subtracts it.

Listing 2.23: Area of Polygon

```
#include<iostream>

double poly_area(double x[], double y[], int size){
  int i=0, j=size-1;  //Set J to next-to-last point
  double area = 0;

  for(i=0; i < size; i++){
    area+=(x[j] + x[i]) * (y[j]-y[i]);
    j = i;
  }

  return abs(area/2);
}
```

2.4.6 Side of a Line

This function returns true or false based on whether two points are both above or both below a line. To calculate which side of a line a point is on, take the value of the line at the x-value of the point and compare it to the y-value of the point, as is done in the function. This function takes in two points in double[2] format, as well as doubles for the slope and y-intercept of the line.

```
#include<iostream>

bool points_line_side(double p1[2], double p2[2], double slope, double intercept){
  double diff1 = slope * p1[0] + intercept - p1[1];
  double diff2 = slope * p2[0] + intercept - p2[1];
  if((diff1*diff2) >= 0)
    return true;
  return false;
}
```

2.4.7 Distance from point to line in 3 Dimensions

This function will resturn the distance between a point and a line as defined by any two points on the line. Input format is 3 3-element double arrays, the first two being points on the line and the third the point from which to measure the distance.

Listing 2.25: Distance from point to line in 3 Dimensions

```
#include<iostream>
#include<cmath>

double p_l_dist(double *li, double *lii, double* point){
  double a[3], b[3], axb[3];
  for(int c=0; c<3; c++) a[c]=-1*li[c]+lii[c];
  for(int c=0; c<3; c++) b[c]=-1*li[c]+point[c];

  axb[0] = ((a[1]*b[2])-(a[2]*b[1]));//Cross product of a and b
  axb[1] = ((a[2]*b[0])-(a[0]*b[2]));//alternatively, use cross product funciton
  axb[2] = ((a[0]*b[1])-(a[1]*b[0]));

  return (sqrt(axb[0]*axb[0] + axb[1]*axb[1] + axb[2]*axb[2])/
    sqrt(a[0]*a[0]+a[1]*a[1]+a[2]*a[2]));
}
```

2.4.8 Point inside Polygon

This function tests whether a point is contained in a polygon defined by a number of vertices and the arrays for the x and y coordinates of the vertices in addition to an x and a y coordinate.

Listing 2.26: Point inside Polygon

```
#include<iostream>

int poly_contains(int vertices, double *vertx, double *verty, double tx, double ty)
{
  int c=0, d=vertices-1, r = 0;
  for (; c < vertices; d = c++) {
    if ( ((verty[c]>ty) != (verty[d]>ty)) &&
      (tx < (vertx[d]-vertx[c]) * (ty-verty[c]) / (verty[d]-verty[c]) + vertx[c]) )
      r = !r;
  }
  return r;
}
```

Continues on next page

Continued from previous page

2.4.9 Polygon Convexity

To calculate whether a polygon is convex or not.

Listing 2.27: Polygon Convexity

```
#include<iostream>

bool is_convex(double *xes, double *yes, int len){
if(len < 3) return false;
double prev, next;
prev = xes[0]*yes[1 ] - xes[1]*yes[0];

for(int c=1; c<len; c++){
  next = xes[c]*yes[c+1] - xes[c+1]*yes[c];
  if((prev*next) < 0) return false;
  prev=next;
}
return true;
}
```

2.5. Flood Fill

This algorithm is a four or eight way recursive method that checks a start node on a graph for and old value, then updates the start node value with a new value and recursively calls in four or eight directions. When the method is called a start node location, the current node value to be changed, and the new value to update the current node value are all passed. Inside the flood fill method the start node value is checked via if statement to not equal the old value. Passing the if statement executes a return call ending the iteration of the method. When the if statement is failed the start node value is updated to the new value. Next a series of four recursive flood fill method calls are executed in north south west and east directions from the start node location. The flood fill method can have eight recursive flood fill method calls to execute in all eight directions from the start node. The flood fill method can be implemented with both and array or a stack, as shown in the example code. It should be noted that a target node location can also be passed to the method to have the method return located. This implementation creates and execution closely relating breath first search on a graph.

2.5.1 Flood Fill

Listing 2.28: Flood Fill

```
#include<fstream>
#include<string>
#include<time.h>
#include<cmath>
#include<stdlib.h>
#include<iostream>

int graph[20][50];

using namespace std;

```

Continues on next page

Continued from previous page

```
void floodfill4(int Sy,int Sx,int oldnumber,int newnumber){

  if(graph[Sy][Sx]!= oldnumber) return;
  graph[Sy][Sx] = newnumber;
  floodfill4(Sy-1, Sx, oldnumber,newnumber);
  floodfill4(Sy+1, Sx, oldnumber,newnumber);
  floodfill4(Sy, Sx-1, oldnumber,newnumber);
  floodfill4(Sy, Sx+1, oldnumber,newnumber);
}
int main(){

  int y=0,z=0;
  ifstream file;
  file.open ("input.in");
  string s;
  cout << "\n";

  while(!file.eof()){
    getline(file,s);
    z = s.length();
    for(int x=0; x < z ; x++){
     graph[y][x]= s[x] - '0';
   }
   y++;
   if(y==20) break;
  }
  file.close();
  cout << "\n";

  int oldnumber=2;
  int newnumber=1;
  int Sx,Sy,Tx,Ty;
  Tx=  3;
  Ty= 18;
  Sx= 24;
  Sy=  9;
  floodfill4(Sy,Sx,oldnumber,newnumber);

  for(int i=0; i < 20; i++){
    for(int j=0; j < 50; j++){
     cout << graph[i][j];
   } cout << "\n";
  }
  cout << "\n";
  cout << "\n";

  return 0;
}
```

2.5.2 Flood Fill with Stack

Listing 2.29: Flood Fill with Stack

```
#include<time.h>
#include<cmath>
#include<stdlib.h>
#include<iostream>
#include<stack>
#include<fstream>
#include<string>

```

Continues on next page

Continued from previous page

```
using namespace std;

typedef pair<int,int> pii;
stack<pii> S;
int graph[20][50]; //global graph

void floodfill(int oldval, int newval){
   while (!S.empty()){
      int i = S.top().first, j = S.top().second;
      S.pop();

      if(graph[i][j] == oldval) {
         graph[i][j] = newval;
         S.push(pii(i-1,j));
         S.push(pii(i+1,j));
         S.push(pii(i,j-1));
         S.push(pii(i,j+1));
}}}

int main(){

  int y=0,z=0;
  ifstream file;
  file.open ("input.in");
  string s;
  cout << "\n";

  while(!file.eof()){
    getline(file,s);
    z = s.length();

    for(int x=0; x < z ; x++){
     graph[y][x]= s[x] - '0';
   }

   y++;
   if(y==20) break;
  }

  file.close();
  cout << "\n";

  int oldnumber=2;
  int newnumber=1;
  S.push(pii(9, 24));
  floodfill(oldnumber,newnumber);

  for(int i=0; i < 20; i++){
    for(int j=0; j < 50; j++){
     cout << graph[i][j];
   } cout << "\n";
  }
  cout << "\n";

  return 0;
}
```

2.5.3 Flood Fill with Target Node

Listing 2.30: Flood Fill with Target Node

Continues on next page

Continued from previous page

```
#include<fstream>
#include<string>
#include<time.h>
#include<cmath>
#include<stdlib.h>
#include<iostream>

int graph[20][50];
using namespace std;

void floodfill4(int Sy,int Sx, int Ty, int Tx, int oldnumber,int newnumber){
  if(graph[Ty][Tx]== newnumber) return;
  if(graph[Sy][Sx]!= oldnumber) return;
  graph[Sy][Sx] = newnumber;
  floodfill4(Sy-1, Sx,Ty,Tx, oldnumber,newnumber);
  floodfill4(Sy+1, Sx,Ty,Tx, oldnumber,newnumber);
  floodfill4(Sy, Sx-1,Ty,Tx, oldnumber,newnumber);
  floodfill4(Sy, Sx+1,Ty,Tx, oldnumber,newnumber);
}

int main(){

  int y=0,z=0;
  ifstream file;
  file.open ("input.in");
  string s;
  cout << "\n";

  while(!file.eof()){
    getline(file,s);
    z = s.length();
    for(int x=0; x < z ; x++){
     graph[y][x]= s[x] - '0';
   }
   y++;
   if(y==20) break;
  }
  file.close();
  cout << "\n";

  int oldnumber=2;
  int newnumber=1;
  int Sx,Sy,Tx,Ty;
  Tx=  3;
  Ty= 18;
  Sx= 24;
  Sy=  9;

  floodfill4(Sy,Sx,Ty,Tx,oldnumber,newnumber);

  for(int i=0; i < 20; i++){
    for(int j=0; j < 50; j++){
     cout << graph[i][j];
   } cout << "\n";
  }
  cout << "\n";

  return 0;
}
```

2.5.4 Input Example

Listing 2.31: Input Example

```
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222200000000000002222222222222222222
22222222222222222202222222222202222222222222222222
22222222222222222202222222222202222222222222222222
22222222222200000002222222222200000002222222222222
22222222222202222222222222222222222202222222222222
22222222222202222222222222222222222202222222222222
22222222222202222222222222222222222202222222222222
22222222222200000002222222222200000002222222222222
22222222222222222202222222222202222222222222222222
22222222222222222202222222222202222222222222222222
22222222222222222200000000000002222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
```

2.5.5 Output Example

Listing 2.32: Output Example

```
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222200000000000002222222222222222222
22222222222222222201111111111102222222222222222222
22222222222222222201111111111102222222222222222222
22222222222200000001111111111100000002222222222222
22222222222201111111111111111111111102222222222222
22222222222201111111111111111111111102222222222222
22222222222201111111111111111111111102222222222222
22222222222200000001111111111100000002222222222222
22222222222222222201111111111102222222222222222222
22222222222222222201111111111102222222222222222222
22222222222222222200000000000002222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222222222
```

2.6. Breadth-first Search

Breadth-first search is a method of searching an unweighted graph of nodes. Ultimately, the search is implemented using a queue to visit nodes in the immediate neighborhood of the current node (nodes that are separated by only one edge) before branching out to explore nodes further away. As each node is seen, it is marked as visited. Only nodes that have not been marked as visited are added to the queue. Using this technique, one can find the shortest path from a given node to another node. The running time of the algorithm can be described as $O(m + n)$ for a given graph containing m nodes and n edges.

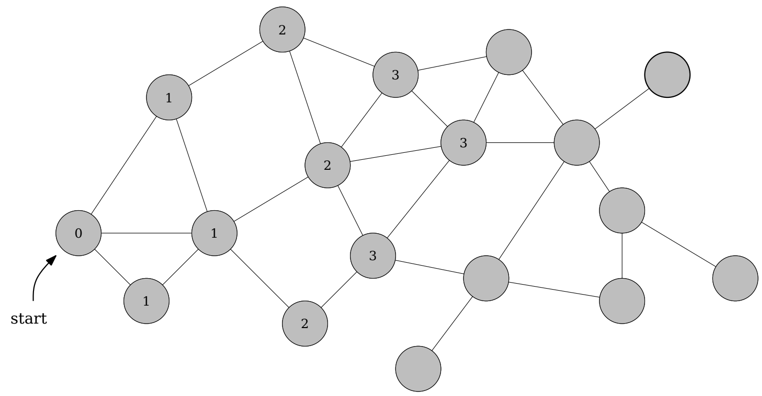

Figure 2.1: A partially completed breadth-first search on a graph with no particular target node.

2.6.1 Applications

- Searching graphs with unit edges. Graphs with weighted edges should use the Dijkstra or Floyd-Warshall algorithm.
- Finding the shortest path between two given nodes.
- Testing a given graph for bipartiteness. By applying arbitrary, alternating labels to nodes as they are visited, one can determine if a graph is bipartite if a break in the alternation occurs.

2.6.2 Example Contest Problem: Hopping Stones

Contemplating wave-based trigonometric functions down by the river forming the southern edge of Farmer John's property may have kept the cows busy for a little while, but studying cowculus with their mathematics mentors is still on their mind. They must escape the bounds of the fences. The problem is that not all of them are able to hop the fence, and none would desire to destroy Farmer John's hard work.

There are a number of large rocks in the river that the cows could use to go around the edge of the fencing. However, many of the cows are queasy about being over the deep waters that it holds and would rather hop around as little as possible.

Reassure the hesitating cows by finding the length of the shortest route possible for them to cross.

Input

- Line 1: An unsigned integer representing the starting position of the cows.
- Line 2: An unsigned integer representing the number of the destination to be hopped to.
- Line 3 to EOF: A pair of unsigned integers representing the stones that can be hopped between.

Sample Input

Listing 2.33: Hopping Stones Input

```
0
20
0 1
0 3
1 8
1 2
8 13
```

Continues on next page

Continued from previous page

```
13 14
8 7
2 7
3 4
4 19
4 5
19 5
19 9
5 6
9 6
9 10
6 11
11 12
10 12
7 15
15 16
16 17
16 18
18 17
12 20
17 20
2 19
```

Output

Text formatted as in the sample output stating the number of hops that must be made to reach the specified destination.

Sample Output

Listing 2.34: Hopping Stones Output

```
It will take 7 hops.
```

Example Solution

Listing 2.35: Hopping Stones Solution

```
#include <iostream>
#include <fstream>
#include <queue>
#include <utility>
#include <vector>
using namespace std;

typedef pair<unsigned int, unsigned int> Puiui;

struct Node
{
  unsigned int id;
  unsigned int dist;
  vector<Node*> neighbors;

  Node() : id(0), dist(0) {}
  Node(const unsigned int id) : id(id), dist(0) {}
  ~Node() {};
};

int main()
{
```

Continues on next page

Continued from previous page

```
  Node* nodes = nullptr;
  vector<Puiui> edges;

  unsigned int Start;
  unsigned int Target;
  cin >> Start;
  cin >> Target;

  // Read in all edges.
  unsigned int a, b;
  unsigned int max_node_id = 0;
  while(cin >> a)
  {
    cin >> b;
    if(a > max_node_id) max_node_id = a;
    if(b > max_node_id) max_node_id = b;
    edges.push_back(make_pair(a, b));
  }

  // Make nodes.
  nodes = new Node[max_node_id + 1];
  for(unsigned int i = 0; i <= max_node_id; i++)
  {
    nodes[i].id = i;
  }

  // Connect nodes by the specified edges.
  for(unsigned int i = 0; i < edges.size(); i++)
  {
    nodes[edges[i].first].neighbors.push_back(&nodes[edges[i].second]);
    nodes[edges[i].second].neighbors.push_back(&nodes[edges[i].first]);
  }

  // Breadth-first search from start to target.
  queue<Node*> q;
  q.push(&nodes[Start]);
  while(!q.empty())
  {
    // Get current node.
    Node* current = q.front();
    q.pop();

    // See if this is the target node.
    if(current->id == Target)
    {
      cout << "It will take " << current->dist << " hops." << endl;
      break;
    }
    else
    {
      // Update neighbors' distances.
      for(unsigned int i = 0; i < current->neighbors.size(); i++)
      {
        Node* const neighbor = current->neighbors[i];
        if(neighbor->dist == 0)
        {
          neighbor->dist = current->dist + 1;
          q.push(neighbor);
        }
      }
    }
  }

```

Continues on next page

Continued from previous page

```
  delete[] nodes;

  return 0;
}
```

Lessons Learned

- If no target node is specified, the algorithm will completely propogate to all reachable nodes. This will result in shortest path calculation for all nodes.
- Representing the edges as a pair of unsigned integers is much quicker than using a dedicated struct.

2.6.3 ACM Contest Problem: Word Ladder[8]

A *word ladder* is a puzzle in which you transform one word into another by changing one letter at a time. But, there's a catch: every word that you form in each step must be in the dictionary! Here's an example of how to transform **CAT** into **GAS**:

CAT → **CAR** → **WAR** → **WAS** → **GAS**

Of course, you want to use the fewest number of transitions possible. These puzzles can be tough, and often you'll think to yourself: "Darn it! If only [*some word*] was in the dictionary!"

Well, now is your chance! Given a dictionary, and a starting and ending word, what ONE single word could you add to the dictionary to minimize the number of steps to get from the starting word to the ending word, changing one letter at a time, and making sure that every word at every step is in the dictionary?

Input

Each input will consist of a single test case. Note that your program may be run multiple times on different inputs. Each test case will start with a line with a single integer n $(2 \leq n \leq 1000)$ which indicates the number of words in the dictionary. The dictionary will follow on the next n lines , with one word per line. All words will consist of between 1 and 8 capital letters only, and all words in a test case wil be the same length. The first word in the list will be the starting word of the word ladder, and the second word will be the ending word of the word ladder.

Listing 2.36: Word Ladder Input

```
3
CAT
DOG
COT

2
CAT
DOG

4
CAT
DOG
COT
COG
```

Output

Output exactly two lines. The first line holds the one single word that you would add to the dictionary, and the second holds an integer indicating the minimum number of steps to get from the starting word to the ending word, adding your word. Output no spaces.

It is possible that there's more than one word you can add that will make your path as short as possible. In this case, output the solution word that comes first alphabetically.

It is possible that there's no word you can add that makes the solution possible. In this case, output 0 (zero) as the word, and -1 as the number of steps.

2.7. Depth-first Search

Depth-first search is a method of searching a graph for the possibility reaching a specified node. As its name suggests, the algorithm will proceed from the starting node to the farthest node of a graph before branching to other nodes. Because of this, depth-first search is **not** guaranteed to find the shortest path. Rather, it will find **a** path if it exists. Traversing an entire graph takes $\Theta(m+n)$ for a graph of m vertices and n edges.

2.7.1 Applications

- Testing if two graphs are connected by some common node.
- Discovering whether a specified state is possible with certain steps.
- Finding connected and strongly connected components in $O(m+n)$ time.

2.7.2 Example Contest Problem: Shaky Stones

For a while now, the cows have been circumventing the new fence Farmer John built by traversing large rocks that are embedded in the river by the farm. Some of the stones have become unstable after the cows have used them a number of times. The ones that are likely to move or sink away have been pointed out by the perceptive cows and given an estimate of the number of times left the stones can be used.

Once again, it's time for the cows to meet their mathematics mentors, and, to do so this time, they need to use the large rocks to get around the fencing. The cows know which rocks should not be used after a number of times.

Inform the cows of whether they'll all be able to make it, or whether a few need will need to stay behind and be tutored later.

Input

- Line 1: The number of cows needing to cross.
- Line 2: The number of stones that can be used for crossing.
- Line 3: The assigned number of the stone the cows start at.
- Line 4: The assigned number of the stone the cows finish at.
- Line 5: The number of stones, r, that have restrictions on them.
- Line 6 to $6+r-1$: Two integers, the first representing the assigned number of the stone with the restriction on it, and the second representing the number of hops that are deemed safe for it.
- Line $6+r$ to EOF: Two integers representing stones that can be safely hopped between.

Sample Input

Listing 2.37: Shaky Stones Input

```
25

21

0
20

4
12 9
11 40
10 32
17 11

0 1
0 3
1 8
1 2
8 13
13 14
8 7
2 7
3 4
4 19
4 5
19 5
19 9
5 6
9 6
9 10
6 11
11 12
10 12
7 15
15 16
16 17
16 18
18 17
12 20
17 20
2 19
```

Output

- Line 1: Print the text 'It is possible.' followed by a newline if the specified number of cows can cross. Otherwise, print the text 'It is not possible. Only x can cross' where x is the number of cows that can cross.

Sample Output

Listing 2.38: Shaky Stones Output

```
It is not possible. Only 20 can cross.
```

Example Solution

Listing 2.39: Shaky Stones Solution

```cpp
#include <cassert>
#include <iostream>
#include <vector>
using namespace std;

struct Node
{
  bool visited;
  unsigned int id;
  int hops_left;
  vector<Node*> neighbors;

  Node() : visited(false), id(0), hops_left(-1) {}
};

bool dfs(Node& n, const unsigned int target)
{
  // Mark this node as visited.
  n.visited = true;
  if(n.hops_left > 0) n.hops_left--;

  // If we are currently visiting the node we have been looking
  // for, then the search is finished.
  if(n.id == target) return true;
  else
  {
    // Iterate through all neighbors of the current node and launch
    // a recursive search on each of them given that they haven't
    // been visited and, for this problem, are still traversable.
    for(vector<Node*>::iterator it = n.neighbors.begin();
        it != n.neighbors.end(); it++)
    {
      if(((*it)->hops_left < 0 || (*it)->hops_left > 0) && !(*it)->visited)
        if(dfs(**it, target)) return true; // A return value of true means the algorithm is complete.
    }
  }

  // Reset the state back to what it was to 'undo' the step.
  if(n.hops_left != -1) n.hops_left++;

  // Returns false if the node that's being search for cannot be
  // reached from this node.
  return false;
}

int main()
{
  unsigned int a, b;
  unsigned int node_count;
  Node* nodes = nullptr;

  // Get number of crossings.
  cin >> a;
  const unsigned int crossings = a;

  // Create Node array.
  cin >> node_count;
  nodes = new Node[node_count];
  assert(nodes);
```

Continues on next page

Continued from previous page

```
  for(unsigned int i = 0; i < node_count; i++)
    nodes[i].id = i;

  cin >> a >> b;
  const unsigned int Start = a;
  const unsigned int Target = b;

  // Apply constraints
  unsigned int constrained_nodes;
  cin >> constrained_nodes;
  for(unsigned int i = 0; i < constrained_nodes; i++)
  {
    cin >> a >> b;
    nodes[a].hops_left = b;
  }

  // Read and connect edges.
  while(cin >> a >> b)
  {
    nodes[a].neighbors.push_back(&nodes[b]);
    nodes[b].neighbors.push_back(&nodes[a]);
  }

  // Simulate the crossing of 'crossings' cows.
  for(unsigned int i = crossings; i > 0; i--)
  {
    if(!dfs(nodes[Start], nodes[Target].id))
    {
      cout << "It is not possible. Only " << crossings - i << " can cross." << endl;
      return 0;
    }

    // Reset visited status.
    for(unsigned int i = 0; i < node_count; i++)
      nodes[i].visited = false;
  }

  cout << "It is possible." << endl;

  delete[] nodes;

  return 0;
}
```

Lessons Learned

- Depth-first search could also be implemented using a stack in place of where breadth-first search would use a queue.
- The algorithm can be written using a while-loop instead of recursion.

2.8. Prim's Algorithm

Prim's algorithm is a greedy algorithm that allows for finding a minimum spanning tree/forest in $O(E \log V)$ time. Prim's algorithm is implemented by doing the following:

1. Choose an arbitrary unvisited node in the graph.
2. Grow the tree by choosing the node that can be added to the tree by least cost.
3. Repeat step 2 until all connected nodes have been visited. If there are still nodes, return to step

1.

2.8.1 Applications

- Finding a minimum spanning tree

2.8.2 Example Contest Problem: Cow Connection

Farmer John has expanded his real estate holdings and now owns several farms. These farms are connected by an old, unmaintained road network.

At night, these roads are unlit and are very dangerous. Farmer John wants to improve safety by installing lights along the roads. However, Farmer John does not have much extra money to spend on this effort.

Help Farmer John determine a minimum cost set of roads that he could light to connect all of the farms by lit roads.

Input Format

- Line 1: One integer, N, $(N < 1000000)$ specifying the number of paths in the corn field.
- Line 2..$(N + 1)$: Each line contains three integers S, D and C where:
  - S and D represent the farm id numbers that a given road connects. $0 \leq S < N$ and $0 \leq D < N$.
  - C is the cost of lighting that road. $(0 < C < 1000000$

Sample Input

Listing 2.40: Cowconnection Sample Input

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

Output Format

- Line 1: One integer, representing the minimum cost.
- Line 2..$(N + 1)$: Two integers, that represent the farms at the end of either road Farmer John should choose to light. These integers should be in order from least to greatest.

Sample Output

Listing 2.41: Cowconnection Sample Output

```
18
0 1
0 2
2 3
2 4
3 5
```

Example Solution

Listing 2.42: Cowconnection Example Solution

```
#include<queue>
#include<unordered_map>
#include<limits>
#include<functional>
#include<iostream>
#include<algorithm>
#include<set>
using namespace std;


//Taken from the section on graphs
typedef unordered_map<int,int> umii;
typedef umii::const_iterator edge_iter;
typedef unordered_map<int,umii> umiumii;
typedef umiumii::const_iterator graph_iter;
class graph{
  umiumii g;
  public:
    void insert(int s, int d, int e){g[s][d] = e;g[d];}
    umii::iterator find(int s, int d){ return g.find(s)->second.find(d);}
    edge_iter cbegin(int s)const{return g.find(s)->second.cbegin();}
    edge_iter cend(int s)const{return g.find(s)->second.cend(); }
        graph_iter begin ()const {return g.cbegin();}
        graph_iter end () const {return g.cend();}
        int size()const{return g.size();}
    graph(): g() {};
};

//Convenience class that improves readably of code.
//endif
class edge{
  public: int source,dest,cost;
  //defining the operator> function allows us to use std::greater<edge>
  bool operator> (const edge& rhs) const{ return cost > rhs.cost ;}
  edge(int source, int dest, int cost):source(source),dest(dest),cost(cost){};
};

//Finds a undirected minimum spanning forest in O(E log V) time
//For connected graphs, this produces a minimum spanning tree
graph prims(const graph& g){

  priority_queue<edge, vector<edge>, greater<edge> > Q; //queue of edges
  vector<bool> in_tree(g.size(), false); //tracks if an element is in the tree
  graph mst; //Graph that holds the minimum spanning forest

  //chose an arbitrary first node and queue all its edges
  for(auto node = g.begin(); node!= g.end(); node++){

    //Skip nodes that have been visited
    if (!in_tree[node->first]) {

      //Visit an arbitrary node
      in_tree[node->first] = true;
      for(auto i = g.cbegin(node->first); i != g.cend(node->first); i++){
        Q.emplace(node->first, i->first, i->second);
      }

      while(Q.size() > 0){

```

Continues on next page

Continued from previous page

```
        //Remove an edge from the queue.
        edge e = Q.top();
        Q.pop();

        //if the edge is in the tree ignore it
        if (!in_tree[e.dest]) {
          in_tree[e.dest] = true;
          mst.insert(e.source, e.dest, e.cost);
          mst.insert(e.dest, e.source, e.cost);
          for(auto i = g.cbegin(e.dest); i != g.cend(e.dest); i++){
            Q.emplace(e.dest, i->first, i->second);
          }
        }
      }
    }
  }
  return mst;

}

int main(){
  //Assumes nodes are numbered 0..N-1
  graph g;
  int n,s,d,e;

  cin >> n;
  while(n--){
    cin >> s >> d >> e;
    g.insert(s,d,e);
    g.insert(d,s,e);
  }

  //Solve the problem
  graph mst = prims(g);

  //Prepare the graph for printing
  int cost=0;
  set<pair<int,int> > edges;
  for (auto i = mst.begin(); i != mst.end(); i++){
    for(auto j = mst.cbegin(i->first); j != mst.cend(i->first); j++){
      int a=i->first,b=j->first;
      cost += j->second;
      edges.emplace(min(a,b),max(a,b));
    }
  }
  //Edges were double counted so divide by 2.
  cost/=2;

  //Print a minimum spanning tree
  cout << cost << endl;
  for(auto i = edges.begin(); i != edges.end(); i++)
    cout << i->first << " " << i->second << endl;

  return 0;
}
```

2.9. Max Flows

This section does not attempt to define basic terms regarding graphs. For these definitions, please review the section entitled graphs in the data structures section of the hackpack.

There are several types of problems that involve finding maximum flow in a graph. Many of these problems can easily tackled by making small transformations to the graph.

The implementation below solves the max flow problem for a single source and sink. If there is more than one source or sink, one can add a "super" source and or sink that connects to all of the sources and sinks. Then find the flow from the super source to the super sink.

The implementation below solves the maximum flow problem for directed graphs. If the graph is undirected, simply insert every edge twice in both directions.

The implementation below places the weights on the edges instead of the nodes. If the flow is limited through the nodes instead of the edges, simply split every node into an "in" node and an "out" node and place the weight on the path between them.

In a general directed graph, this can be done in $O(EV \log V \log F)$ where E is the number of edges V is the numbr of nodes, and time where F is the maximum flow.

- While there is a path from source to sink
  - Greedily find the widest path — the single path with greatest capacity.
  - Reduce the capacity of the edges on the widest path by the capacity of the widest path.
  - Increase the capacity of the edges on the widest path by the capacity of the widest path going in the opposite direction creating them if necessary.

2.9.1 Applications

- Finding the maximum flow in a graph
- Finding a minimum set of edges required to disconnect source and sink
- Finding a maximal matching

2.9.2 Example Contest Problem: Cow-Ex

The cows have opened up a package distribution system.

The cows have several routes by which they distribute packages. Each route has been rated with a positive integer referring to the amount of packages that the route can carry in 1 day.

Today is Farmer John's birthday, and the cows wish to send him as many packages as possible. Help the cows determine the how many packages they can send to Farm John's barn.

Input Format

- Line 1: A positive integer N that represents the number of routes that the cows have in place. $0 < N \leq 10000$
- Line 2..$(N + 1)$: Three non-negative integers, S, D, E representing a route going from location S to location D with a capacity of E. $0 \leq S, D < N$
- Line $(N + 2)$: A non-negative integer A, that represents location where packages are sent.
- Line $(N + 3)$: A non-negative integer B, that represents location of the barn.

Sample Input

Listing 2.43: Cowex Sample Input

```
8
4 0 9
```

Continues on next page

Continued from previous page

```
4 3 4
0 2 5
3 0 5
3 5 7
5 1 8
2 5 6
2 1 2
4
1
```

Output Format

- Line 1: A Positive integer representing the maximum capacity of the network. This should be 0 if it is not possible to transport any packages.

Sample Output

Listing 2.44: Cowex Sample Output

```
9
```

Example Solution

Listing 2.45: Cowex Sample Solution

```
#include<iostream>
#include<algorithm>
#include<limits>
#include<unordered_map>
#include<vector>
using namespace std;

//Taken from the section on graphs
typedef unordered_map<int,int> umii;
typedef umii::const_iterator edge_iter;
typedef unordered_map<int,umii> umiumii;
typedef umiumii::const_iterator graph_iter;
class graph{
  umiumii g;
  public:
  void insert(int s, int d, int e){g[s][d] = e;g[d];}
  umii::iterator find(int s, int d){ return g.find(s)->second.find(d);}
  edge_iter cbegin(int s)const{return g.find(s)->second.cbegin();}
  edge_iter cend(int s)const{return g.find(s)->second.cend(); }
  graph_iter begin ()const {return g.cbegin();}
  graph_iter end () const {return g.cend();}
  int size()const{return g.size();}
  graph(): g() {};
};
int max_flow(graph g, int source, int sink){
  //If the source and sink are the same node, then max flow
  if (source == sink) return numeric_limits<int>::max();

  //initalize max flow
  int total_flow = 0;

  while(true){
    //initialize data structures
    vector<int> flow(g.size() , 0);
    vector<bool> visited(g.size() , false);
```

Continues on next page

Continued from previous page

```
    vector<int> previous_node(g.size(), -1);

    //initialize the source
    flow[source] = numeric_limits<int>::max();

    //Find the widest path in the graph (path of maximal flow)
    int max_flow, max_loc;
    while(true){
      max_flow = 0;
      max_loc = -1;

      // find unvististed node with highest capacity O(V) time
      for(auto i = g.begin(); i!=g.end(); i++){
        if( flow[i->first] > max_flow && ! visited[i->first]){
          max_flow = flow[i->first];
          max_loc = i->first;
        }
      }

      //No path from source to sink
      if ( max_loc == -1 ){
        break;
      }

      //Done! the best node is the sink
      if ( max_loc == sink ){
        break;
      }
      //Update the edges leaving the node with the most flow O(1) in sparce graph
      visited[max_loc] = true;
      for (auto i = g.cbegin(max_loc); i != g.cend(max_loc); i++){
        if (flow[i->first] < min(max_flow, i->second)){
          previous_node[i->first] = max_loc;
          flow[i->first] = min(max_flow, i->second);
        }
      }
    }

    //There was no remaining path from source to sink
    if (max_loc == -1){
      break;
    }

    // update reverse flows
    int pathcapactiy = flow[sink];
    total_flow += pathcapactiy;
    int current_node = sink;
    while (current_node != source){
      int next_node= previous_node[current_node];
      int rev_cap;
      if(g.find(current_node, next_node) == g.cend(current_node)){
        rev_cap = 0;
      } else {
        rev_cap = g.find(current_node,next_node)->second;
      }
      g.insert(next_node, current_node, g.find(next_node, current_node)->second - pathcapactiy);
      g.insert(current_node,next_node, rev_cap + pathcapactiy);
      current_node = next_node;
    }
  }
  return total_flow;
}

```

Continues on next page

Continued from previous page

```

int main(int argc, char *argv[])
{

  int n,s,d,e;
  graph g;
  cin >> n;
  while(n--){
    cin >> s >> d >> e;
    g.insert(s,d,e);
  }
  cin >> s >> d;

  int m = max_flow(g, s, d);

  cout << m << endl;

  return 0;
}
```

Lessons Learned

- The innermost while loop is a solution to the widest most path problem that runs in $O(E \log V)$ time.

3. Approaches

3.1. Hash Window Approaches

Hashing is a mapping of objects to bytes. These bytes are often stored in Strings or numeric data types. Normally, hash functions attempt to map each object to a *unique* set of bytes. Also a *slight* change in the object should cause a *large* change in the output. When two objects map to the same set of bytes, it is referred to as a hash collision.

3.1.1 Applications

- Identify substrings quickly and cleanly.
- Hash collisions to detect certain features of the input.

3.1.2 Fine Print Returns!

The cows have prepared another legal document for Farmer John to review.

The Cows are still championing the installation of an Olympic style swimming pool. However, this time Farmer John only cares about how many times, ‘pool’ appears in the text.

Help Farmer John determine how many times the text ‘pool’ appears in the document.

Input

- A stream of text terminated by an EOF representing the legal agreement

Sample Input

Listing 3.1: Fine Print Sample Input

```
pool pool apoole
```

Output

- 1 integer representing the number of references to the word pool in the text.

Sample Output

Listing 3.2: Fine Print Sample Output

```
3
```

Sample Solution

Listing 3.3: Fine Print Sample Solution

```
#include <iostream>
#include <string>
#include <cmath>
using namespace std;
#define PRIME 31
#define LENGTH_TARGET 4

```

Continues on next page

Continued from previous page

```
int main()
{
  //Initialize the strings
  string target = "pool";
  string text = "", buffer ="";
  int target_hash = 0;
  int hash = 0;
  int count = 0;

  //Read in the string into memory
  while(!cin.eof()) {getline(cin,buffer); text +=buffer;}

  //Calculate the starting hashes
  for(unsigned int i=0; i<target.length(); i++){
    target_hash = target_hash * PRIME + target[i];
    hash = hash * PRIME + text[i] ;
  }

  //Calculate the hashes as the windows slides
  for(unsigned int i=0,j=target.length();j<text.length();i++,j++){
    if(hash == target_hash) count++;
    hash = (hash * PRIME) + text[j] - text[i]*pow(PRIME,LENGTH_TARGET);
  }
  //don't forget the possible ending point match
  if(hash == target_hash) count++;

  cout << count << endl;

  return 0;
}
```

Lessons Learned

- There is often more than one way to solve a problem, checkout the KMP string matching algorithm for another way to solve this problem.

3.2. Dynamic Programming

Dynamic Programming is a powerful tool that can be applied to several different types of algorithms.[2] The basic idea is to save the results of smaller problems and use the results to solve larger problems.

3.2.1 Applications

- Improving runtimes of some other algorithms
- Solving the knapsack in $O(nm)$ time
- Solving the integer knapsack in $O(nm)$ time
- Solving the largest increasing subsequence in $O(n \log n)$ time
- Solving the maximum value sub-array problem in $O(n)$
- Solving the maximum value continuous sub-array problem

3.2.2 Example Contest Problem: A Knapsack Full of Fireworks

The cows on Farmer John's Farm are planning on putting on a fireworks show for Farmer John's birthday.

They have pooled all of their loose change, and hope to purchase a collection of fireworks that will maximize Farmer John's amazement during the show so that he will be more likely to build them a new barn. Each firework's label helpfully includes a "wow factor" rating explicitly for this purpose. A high "wow factor" is more desirable than a low one.

Please help the cows determine the maximum "wow factor" they can get for their loose change.

Input Format

- Line 1: One integer, N $(1 \leq N \leq 100)$, the number of fireworks in the catalog.
- Line 2: One integer, C $(1 \leq C \leq 10000)$, the total amount of change that the cows have to spend.
- Lines 3..$(N + 2)$ Two integers P, W representing the price and wow factor for the fireworks.

Sample Input

Listing 3.4: A Knapsack Full of Fireworks Input

```
4
3
1 1
2 2
3 5
```

Output Format

- Line 1: A single integer representing the maximum wow factor.

Sample Output

Listing 3.5: A Knapsack Full of Fireworks Output

```
6
```

Example Solution

Listing 3.6: A Knapsack Full of Fireworks Solution

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

//Create a item class to represent the items
class item{
  public:
    int size,value;
    //This is an initializer list; it makes initializing easy
    item(int s, int v): size(s), value(v) {}
};

//Create a vector of items to hold all of the items
vector<item> items;
vector<int> A;

int main(int argc, char *argv[])
{
  //Read in the number of items, capacity, and items
```

Continues on next page

Continued from previous page

```
  int N,C,s,v;
  cin >> C >> N;
  for (int i = 0; i < N; i++){
    cin >> s >> v;
    items.push_back(item(s,v));
  }

  //Solve the problem
  //Be certain to include the base case M(0) = 0
  A.push_back(0);

  //Iteratively solve the rest
  for (int j = 1; j <= C; j++){
    int m=INT_MIN;
    for(vector<item>::iterator i = items.begin(); i != items.end(); i++){
      if(j - i->size >= 0) m = max(A[j - i->size] + i->value, m);
    }
    A.push_back( max(A[j-1], m));
  }
  cout << A[C] << endl;

  return 0;
}
```

Lessons Learned

The optimal solution is of the form:

$$W(j) = \max\{W(j-1), \max\{W(j-p_i) + v_i\}\}$$

Where $W(0) = 0$

3.2.3 Example Contest Problem: A Few Fireworks More

The cows have reconsidered their original plan of buying just the fireworks with the greatest total "wow factor". Instead, they want to incorporate "wow factor" *and* diversity, so the cows have decided to purchase a collection of fireworks that optimizes "wow factor" and includes no more than one of each kind of firework in the catalog.

Please help the cows determine the maximum "wow factor" they can get for their loose change, on the condition that they purchase no more than one of each kind of firework in the catalog.

Input

- Line 1: One integer, N, $(1 \leq N \leq 100)$ the number of fireworks in the catalog.
- Line 2: One integer, C, $(1 \leq C \leq 10000)$ the number of cents that the cows found.
- Lines 3..$(N+2)$ Two integers P, W representing the price and wow factor for the fireworks.

Sample Input

Listing 3.7: A Few Fireworks More Input

```
3
4
1 2
2 4
6 1
```

Output Format

- Line 1: A single integer representing the maximum wow factor using each firework at most once

Sample Output

Listing 3.8: A Few Fireworks More Output

```
6
```

Example Solution

Listing 3.9: A Few Fireworks More Solution

```cpp
#include<iostream>
#include<utility>
#include<algorithm>
using namespace std;

//Assumes that no more than 1000 items considered
//and that no more than 10000 capacity is used
#define MAX_NUM_OF_ITEMS 1001
#define MAX_CAPACITY 10001
typedef pair<int,int> pii;

int A[MAX_NUM_OF_ITEMS][MAX_CAPACITY];
pii items[MAX_NUM_OF_ITEMS];

int main(int argc, char *argv[])
{
  //Read in the input
  int N,C, size, value;
  cin >> N >> C;
  for (int i = 1; i <= N; i ++){
    cin >> size >> value ;
    items[i] = make_pair(size, value);
  }
  //No need to set up base cases since global arrays of ints
  //initialize to 0

  //Solve the problem;  Can be optomized to use O(C) memory
  for(int i = 1; i <= N; i++){
    for(int j = 1; j <= C; j++){
      if( j - items[i].first >= 0)
        A[i][j] = max( A[i-1][j], A[i-1][j- items[i].first] + items[i].second );
      else A[i][j] = A[i-1][j];
    }
  }
  cout << A[N][C] << endl;
  return 0;
}
```

Lesson Learned

A similar problem to the knapsack, except each item can be used at most once. The solution here is to expand the state space. The optimal solution is of the form

$$M(i,j) = \max\{M(i-1,j), M(i-1,j-s_i)+v_i\}$$

Where $M(0,j)=0$ and $M(i,0)=0$

3.2.4 Example Contest Problem: The Good, the Bad, the Cowy

Farmer John's birthday party went off without a hitch, but the cows are worried that Farmer John isn't yet convinced that he should build the cows a new barn. Just in case, they have decided to put it to a vote whether or not they should bake him a cake as well. Unfortunately, the cows are all experts in the school of bovine politics, and think that a simple majority vote will simply not do because of the dangers of vote rigging.

Instead, the cows have resorted to a rather odd voting system: each cow votes in some arbitrary order with an integer value, and if the length of largest increasing subsequence of all the votes is greater than half the number of cows, then the cows will bake Farmer John a cake.

Help the cows determine the results of their vote.

Input

- Line 1: Several integers, separated by spaces, representing the votes of the cows.

Sample Input

Listing 3.10: The Good, the Bad, the Cowy Input

```
100 1 -10 2 -3 3 -9 4 8 10 16
```

Output Format

- Line 1: "1" if the cows have decided to bake a cake, and "0" otherwise.

Sample Output

Listing 3.11: The Good, the Bad, the Cowy Output

```
1
```

Example Solution

Listing 3.12: The Good, the Bad, the Cowy Solution

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

int main(int argc, char *argv[])
{
  int vote;
  vector<int> votes; // the votes of the cows
  vector<int> lenght; // the length of the longest sequence
  //Read in the votes of the cows
  while(cin >> vote) {
    votes.push_back(vote);
    lenght.push_back(1);
  }

  //solve the problem
  //for each stopping place
  for(int i=0; i<votes.size(); i++){
    //extend the sequence if possible
    for(int j=0; j<i; j++){
      if (votes[j] < votes[i]) lenght[i] = max(lenght[i],lenght[j]+1);
```

Continues on next page

Continued from previous page

```
    }
  }

  //The largest increasing subsequence could end with any vote
  int k =0;
  for(int i=0; i< votes.size(); i++){
    k = max(k,lenght[i]);
  }

  //if the length of the largest increasing subsequence is
  //greater than the number of votes/2 then the cows bake a
  //cake else they do not
  if (k > votes.size()/2) cout << "1" << endl;
  else cout << "0" << endl;


  return 0;
}
```

Lesson Learned

- This problem can be solved in $O(n \log n)$ time.
- Sometimes you have to check the entire array to find the solution.
- $while(cin >> val)$ can be used to read in an uncertain number of values.

4. Appendix

4.1. C IO Functions

Occasionally it is far easier to use the C IO functions to meet an output spec. It is also possible to set the precision and width options via numbers after the present, but before the specifier. The general form of a specifier is:

Listing 4.1: Format Codes for printf()

```
%[flags][width][.precision][length]specifier
```

Table 4.1: Format Specifier Codes[5]

Format Code	Output	Example
d	Signed Int	314
u	Unsigned Int	314
o	Unsigned Octal	472
x	Unsigned hex	13a
X	UNSIGNED HEX	13A
f	floating point	3.140000
e	Scientific notation	3.140000e+00
c	character	A
s	string	ACM
p	pointer address	0x40060c
l	Used with other specifiers to indicate a long	314
%%	Prints a literal %	%

Table 4.2: Modifier Flags[5]

Format Code	Output	Example
-	Left-justify	314
+	Force-sign character	+314
#	Show prefix	0x13a
	Show decimal point	314.
0	Left pad field with 0	0314

4.1.1 Examples

./general/ciofunctions/ciofunctions.cpp

```
#include <stdio.h>
int main (){
    //To stdout
    double f = 3.14;
    int    i = 314;
    char*  s = "ACM";
    printf("%5f,%5i,%5s\n", f, i, s);

    //Same thing to a file
    FILE * outputfile;
```

Continues on next page

Continued from previous page

```
    outputfile = fopen("outputfile.txt","w");
    fprintf(outputfile,"%5f,%5i,%5s\n", f, i, s);
    fclose(outputfile);
}
```

4.2. Some Basic VIMRC Settings

Listing 4.2: vimrc

```
set nocp      " prefer vim mode to vi
set mouse=a   " Turn the mouse on
imap jj <ESC> " Make it easy to escape
set ai        " turn on auto-indention
set cin       " turn on c-style auto indentation
set nu        " turn on numbers
set ru        " turn on ruler
set showmode  " turn on modeline
set scs       " turn on smart case search
set bs=2      " turn on better backspacing
set ts=4      " Set tabstop to 4 spaces
set hidden    " Allow files to be opened in background
sy on         " turn on syntax highlighting
colo elflord  " pick a nice color scheme
set bg=dark   " make it better for black terminals
```

4.3. Some Basic Emacs Settings

The configuration file for emacs can be placed in a few different places under a few different names:

- ~/.emacs
- ~/.emacs.el
- ~/.emacs.d/init.el

Listing 4.3: .emacs

```
;; Better indenting for C/C++
(setq c-default-style "linux"
      c-basic-offset 4)

;; Place backup files somewhere out-of-the-way
(setq backup-directory-alist ‘(("." . "~/.saves")))

;; Compile more quickly
(global-set-key (kbd "C-c c") ’compile)

;; Line numbering
(setq linum-format "%4d \u2502")
(global-linum-mode t)

;; Get rid of GUI elements.
(menu-bar-mode -1)
(scroll-bar-mode -1)
(tool-bar-mode -1)

```

Continues on next page

Continued from previous page

```
;; OOB theming is a new Emacs thing (versions >= 24)
;; If that’s available, use a comfortable color scheme
(if (>= emacs-major-version 24)
    (load-theme ’misterioso))
```

4.4. Makefile (and Helper)

Building and testing your code should be easy – the competition is about algorithms, after all. Here’s a helpful `Makefile` that will build `./probname` from invoking `make probname`:

Listing 4.4: Makefile

```
# ’make prob’ creates ./prob from prob.cpp or prob.py;
# you don’t need to copy any rules you won’t be using.

%: %.cpp
  g++ $< -g -Wall -o $@

PYTHON=python # Change this if you need the python3 or python2 executable
%: %.py
  echo -e "#!/usr/bin/env ${PYTHON}\n" > $@
  cat $< >> $@
  chmod +x $@
```

For easy testing over multiple input files, save any input or output samples to `probname.<input-id>.in` and `probname.<matching-id>.out` (those files for output checking are optional) and use this script:

Listing 4.5: Makefile Helper Script ’t’

```
#!/usr/bin/env bash

# NOTE: ’make probname’ must create an executable called ’probname’.
# USAGE: assuming this file is saved as ./t (be sure to run chmod +x ./t),
# ’./t probname’ will test probname with any files matching ’probname.*.in’,
# comparing output with any similar files named ’probname.*.out’. It may help
# to add an ’alias t=./t’ to your bashrc so that you can invoke ’t <probname>’.

make $1
for t in $(ls $1.*.in 2> /dev/null); do
  h=">> running with $t"
  o=$(./$1 < $t)

  # This chunk handles output checking -- it’s removable.
  p="${t%%.in}.out"
  if [ -a $p ]; then
    diff -q <(echo -e $o) $p > /dev/null
    if [ $? -ne 0 ]; then
      h="$h (output differs)"
      o="$o\n--- doesn’t match $p: ---\n$(cat $p)"
    fi
  fi

  echo -e $h
  echo -e $o
done
```

5. C++ Standard Library

This chapter was pulled from cppreference.com.[6]

5.1. Utilities library

5.1.1 std::pair

NAME

std::pair - std::pair is a struct template that provides a way to store two heterogeneous objects as a single unit. A pair is a specific case of a std::tuple with two elements.

SYNOPSIS

#include <utility>

```
 template<
    class T1,
    class T2
> struct pair;
```

MEMBER FUNCTIONS

std::pair::pair(3) - constructs new pair (public member function) std::pair::operator=(3) - assigns the contents (public member function) std::pair::swap(3) [C++11] - swaps the contents (public member function)

NON-MEMBER FUNCTIONS

make_pair(3) - creates a pair object of type, defined by the argument types (function template) operator==(3), operator!=(3), operator<(3), operator<=(3), operator>(3), operator>=(3) - lexicographically compares the values in the pair (function template) std::swap(std::pair)(3) [C++11] - specializes the std::swap algorithm (function template) std::get(std::pair)(3) [C++11] - accesses an element of a pair (function template)

5.1.2 std::tuple

NAME

std::tuple - Class template std::tuple is a fixed-size collection of heterogeneous values. It is a generalization of std::pair.

SYNOPSIS

#include <tuple>

```
 template< class... Types >
class tuple; [since C++11]
```

MEMBER FUNCTIONS

std::tuple::tuple(3) - constructs a new tuple (public member function) std::tuple::operator=(3) - assigns the contents of one tuple to another (public member function) std::tuple::swap(3) - swaps the contents of two tuples (public member function)

NON-MEMBER FUNCTIONS

make_tuple(3) - creates a tuple object of the type defined by the argument types (function template) tie(3) - creates a tuple of lvalue references or unpacks a tuple into individual objects (function template) forward_as_tuple(3) - creates a tuple of rvalue references (function template) tuple_cat(3) - creates a tuple by concatenating any number of tuples (function template) std::get(std::tuple)(3) - tuple accesses specified element (function template) operator==(3), operator!=(3), operator<(3), operator<=(3), operator>(3), operator>=(3) - lexicographically compares the values in the tuple (function template) std::swap(std::tuple)(3) [C++11] - specializes the std::swap algorithm (function template)

5.2. Strings library

5.2.1 std::basic_string

NAME

std::basic_string - The class template basic_string stores and manipulates sequences of char-like objects. The class is dependent neither on the character type nor on the nature of operations on that type. The definitions of the operations are supplied via the Traits template parameter - a specialization of std::char_traits or a compatible traits class.

SYNOPSIS

#include <string>

```
 template<
    class CharT,
    class Traits = std::char\_traits<CharT>,
    class Allocator = std::allocator<CharT>
> class basic\_string;
```

MEMBER FUNCTIONS

std::basic_string::basic_string(3) - constructs a basic_string (public member function) std::basic_string::operator=(3) - assigns values to the string (public member function) std::basic_string::assign(3) - assign characters to a string (public member function) std::basic_string::get_allocator(3) - returns the associated allocator (public member function)

Element access std::basic_string::at(3) - access specified character with bounds checking (public member function) std::basic_string::operator[](3) - access specified character (public member function) std::basic_string::front(3) [C++11] - accesses the first character (public member function) std::basic_string::back(3) [C++11] - accesses the last character (public member function) std::basic_string::data(3) - returns a pointer to the first character of a string (public member function) std::basic_string::c_str(3) - returns a non-modifiable standard C character array version of the string (public member function)

Iterators std::basic_string::begin(3), std::basic_string::cbegin(3) [C++11] - returns an iterator to the beginning (public member function) std::basic_string::end(3), std::basic_string::cend(3) [C++11] - returns an iterator to the end (public member function) std::basic_string::rbegin(3), std::basic_string::crbegin(3) [C++11] - returns a reverse iterator to the beginning (public member function) std::basic_string::rend(3), std::basic_string::crend(3) [C++11] - returns a reverse iterator to the end (public member function)

Capacity std::basic_string::empty(3) - checks whether the string is empty (public member function) std::basic_string::size(3), std::basic_string::length(3) - returns the number of characters (public member function) std::basic_string::max_size(3) - returns the maximum number of characters (public member function) std::basic_string::reserve(3) - reserves storage (public member function) std::basic_string::capacity(3) - returns the number of characters that can be held in currently allocated storage (public member function) std::basic_string::shrink_to_fit(3) [C++11] - reduces memory usage by freeing unused memory (public member function)

Operations std::basic_string::clear(3) - clears the contents (public member function) std::basic_string::insert(3) - inserts characters (public member function) std::basic_string::erase(3) - removes characters (public member function) std::basic_string::push_back(3) - appends a character to the end (public member function) std::basic_string::pop_back(3) [C++11] - removes the last character (public member function) std::basic_string::append(3) - appends characters to the end (public member function) std::basic_string::operator+=(3) - appends characters to the end (public member function) std::basic_string::compare(3) - compares two strings (public member function) std::basic_string::replace(3) - replaces specified portion of a string (public member function) std::basic_string::substr(3) - returns a substring (public member function) std::basic_string::copy(3) - copies characters (public member function) std::basic_string::resize(3) - changes the number of characters stored (public member function) std::basic_string::swap(3) - swaps the contents (public member function)

Search std::basic_string::find(3) - find characters in the string (public member function) std::basic_string::rfind(3) - find the last occurrence of a substring (public member function) std::basic_string::find_first_of(3) - find first occurrence of characters (public member function) std::basic_string::find_first_not_of(3) - find first absence of characters (public member function) std::basic_string::find_last_of(3) - find last occurrence of characters (public member function) std::basic_string::find_last_not_of(3) - find last absence of characters (public member function)

NON-MEMBER FUNCTIONS

operator+(3) - concatenates two strings or a string and a char (function template) operator==(3), operator!=(3), operator<(3), operator>(3), operator<=(3), operator>=(3) - lexicographically compares two strings (function template) std::swap(std::basic_string)(3) - specializes the std::swap algorithm (function template)

Input/output operator«(3), operator»(3) - performs stream input and output on strings (function template) getline(3) - read data from an I/O stream into a string (function)

Numeric conversions stoi(3), stol(3), stoll [C++11] [C++11](3) [C++11] - converts a string to a signed integer (function) stoul(3), stoull [C++11](3) [C++11] - converts a string to an unsigned integer (function) stof(3), stod(3), stold [C++11] [C++11](3) [C++11] - converts a string to a floating point value (function) to_string(3) [C++11] - converts an integral or floating point value to string (function) to_wstring(3) [C++11] - converts an integral or floating point value to wstring (function)

5.3. Containers library

5.3.1 std::array

NAME

std::array - std::array is a container that encapsulates fixed size arrays.

SYNOPSIS

#include <array>

```
 template<
    class T,
    std::size\_t N

> struct array; [since C++11]
```

MEMBER FUNCTIONS

Implicitly-defined member functions std::array::array(implicitly declared)(3) - default-initializes or copy-initializes every element of the array (public member function) std::array:: array(implicitly declared)(3) - destroys every element of the array (public member function) std::array::operator=(implicitly declared)(3) - overwrites every element of the array with the corresponding element of another array (public member function)

Element access std::array::at(3) - access specified element with bounds checking (public member function) std::array::operator[](3) - access specified element (public member function) std::array::front(3) - access the first element (public member function) std::array::back(3) - access the last element (public member function) std::array::data(3) - direct access to the underlying array (public member function)

Iterators std::array::begin(3), std::array::cbegin(3) - returns an iterator to the beginning (public member function) std::array::end(3), std::array::cend(3) - returns an iterator to the end (public member function) std::array::rbegin(3), std::array::crbegin(3) - returns a reverse iterator to the beginning (public member function) std::array::rend(3), std::array::crend(3) - returns a reverse iterator to the end (public member function)

Capacity std::array::empty(3) - checks whether the container is empty (public member function) std::array::size(3) - returns the number of elements (public member function) std::array::max_size(3) - returns the maximum possible number of elements (public member function)

Operations std::array::fill(3) - fill the container with specified value (public member function) std::array::swap(3) - swaps the contents (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3), operator<(3), operator<=(3), operator>(3), operator>=(3) - lexicographically compares the values in the array (function template) std::get(std::array)(3) - accesses an element of an array (function template) std::swap(std::array)(3) [C++11] - specializes the std::swap algorithm (function template)

5.3.2 std::vector

NAME

std::vector - std::vector is a sequence container that encapsulates dynamic size arrays.

SYNOPSIS

#include <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```

MEMBER FUNCTIONS

std::vector::vector(3) - constructs the vector (public member function) std::vector:: vector(3) - destructs the vector (public member function) std::vector::operator=(3) - assigns values to the container (public member function) std::vector::assign(3) - assigns values to the container (public member function) std::vector::get_allocator(3) - returns the associated allocator (public member function)

Element access std::vector::at(3) - access specified element with bounds checking (public member function) std::vector::operator[](3) - access specified element (public member function) std::vector::front(3) - access the first element (public member function) std::vector::back(3) - access the last element (public member function) std::vector::data(3) [C++11] - direct access to the underlying array (public member function)

Iterators std::vector::begin(3), std::vector::cbegin(3) - returns an iterator to the beginning (public member function) std::vector::end(3), std::vector::cend(3) - returns an iterator to the end (public member function) std::vector::rbegin(3), std::vector::crbegin(3) - returns a reverse iterator to the beginning (public member function) std::vector::rend(3), std::vector::crend(3) - returns a reverse iterator to the end (public member function)

Capacity std::vector::empty(3) - checks whether the container is empty (public member function) std::vector::size(3) - returns the number of elements (public member function) std::vector::max_size(3) - returns the maximum possible number of elements (public member function) std::vector::reserve(3) - reserves storage (public member function) std::vector::capacity(3) - returns the number of elements that can be held in currently allocated storage (public member function) std::vector::shrink_to_fit(3) [C++11] - reduces memory usage by freeing unused memory (public member function)

Modifiers std::vector::clear(3) - clears the contents (public member function) std::vector::insert(3) - inserts elements (public member function) std::vector::emplace(3) [C++11] - constructs element in-place (public member function) std::vector::erase(3) - erases elements (public member function) std::vector::push_back(3) - adds elements to the end (public member function) std::vector::emplace_back(3) [C++11] - constructs elements in-place at the end (public member function) std::vector::pop_back(3) - removes the last element (public member function) std::vector::resize(3) - changes the number of elements stored (public member function) std::vector::swap(3) - swaps the contents (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3), operator<(3), operator<=(3), operator>(3), operator>=(3) - lexicographically compares the values in the vector (function template) std::swap(std::vector)(3) - specializes the std::swap algorithm (function template)

5.3.3 std::deque

NAME

std::deque - std::deque (double-ended queue) is an indexed sequence container that allows fast insertion and deletion at both its beginning and its end. In addition, insertion and deletion at either end of a deque never invalidates pointers or references to the rest of the elements.

SYNOPSIS

#include <deque>

```
 template<
    class T,
    class Allocator = std::allocator<T>
> class deque;
```

MEMBER FUNCTIONS

std::deque::deque(3) - constructs the deque (public member function) std::deque:: deque(3) - destructs the deque (public member function) std::deque::operator=(3) - assigns values to the container (public member function) std::deque::assign(3) - assigns values to the container (public member function) std::deque::get_allocator(3) - returns the associated allocator (public member function)

Element access std::deque::at(3) - access specified element with bounds checking (public member function) std::deque::operator[](3) - access specified element (public member function) std::deque::front(3) - access the first element (public member function) std::deque::back(3) - access the last element (public member function)

Iterators std::deque::begin(3), std::deque::cbegin(3) - returns an iterator to the beginning (public member function) std::deque::end(3), std::deque::cend(3) - returns an iterator to the end (public member function) std::deque::rbegin(3), std::deque::crbegin(3) - returns a reverse iterator to the beginning (public member function) std::deque::rend(3), std::deque::crend(3) - returns a reverse iterator to the end (public member function)

Capacity std::deque::empty(3) - checks whether the container is empty (public member function) std::deque::size(3) - returns the number of elements (public member function) std::deque::max_size(3) - returns the maximum possible number of elements (public member function) std::deque::shrink_to_fit(3) [C++11] - reduces memory usage by freeing unused memory (public member function)

Modifiers std::deque::clear(3) - clears the contents (public member function) std::deque::insert(3) - inserts elements (public member function) std::deque::emplace(3) [C++11] - constructs element in-place (public member function) std::deque::erase(3) - erases elements (public member function) std::deque::push_back(3) - adds elements to the end (public member function) std::deque::emplace_back(3) [C++11] - constructs elements in-place at the end (public member function) std::deque::pop_back(3)

- removes the last element (public member function) std::deque::push_front(3) - inserts elements to the beginning (public member function) std::deque::emplace_front(3) [C++11] - constructs elements in-place at the beginning (public member function) std::deque::pop_front(3) - removes the first element (public member function) std::deque::resize(3) - changes the number of elements stored (public member function) std::deque::swap(3) - swaps the contents (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3), operator<(3), operator<=(3), operator>(3), operator>=(3) - lexicographically compares the values in the deque (function template) std::swap(std::deque)(3) - specializes the std::swap algorithm (function template)

5.3.4 std::forward_list

NAME

std::forward_list - std::forward_list is a container that supports fast insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is implemented as a singly-linked list and essentially does not have any overhead compared to its implementation in C. Compared to std::list this container provides more space efficient storage when bidirectional iteration is not needed.

SYNOPSIS

#include <forward_list>

```
 template<
    class T,
    class Allocator = std::allocator<T>

> class forward\_list; [since C++11]
```

MEMBER FUNCTIONS

std::forward_list::forward_list(3) - constructs the forward_list (public member function) std::forward_list:: forward_list(3) - destructs the forward_list (public member function) std::forward_list::operator=(3) - assigns values to the container (public member function) std::forward_list::assign(3) - assigns values to the container (public member function) std::forward_list::get_allocator(3) - returns the associated allocator (public member function)

Element access std::forward_list::front(3) - access the first element (public member function)

Iterators std::forward_list::before_begin(3), std::forward_list::cbefore_begin(3) - returns an iterator to the element before beginning (public member function) std::forward_list::begin(3), std::forward_list::cbegin(3) - returns an iterator to the beginning (public member function) std::forward_list::end(3), std::forward_list::cend(3) - returns an iterator to the end (public member function)

Capacity std::forward_list::empty(3) - checks whether the container is empty (public member function) std::forward_list::max_size(3) - returns the maximum possible number of elements (public member function)

Modifiers std::forward_list::clear(3) - clears the contents (public member function) std::forward_list::insert_after(3) - inserts elements after an element (public member function) std::forward_list::emplace_after(3) - constructs elements in-place after an element (public member function) std::forward_list::erase_after(3) - erases an element after an element (public member function) std::forward_list::push_front(3) - inserts elements to the beginning (public member function) std::forward_list::emplace_front(3) - constructs elements in-place at the beginning (public member function) std::forward_list::pop_front(3) - removes the first element (public member function) std::forward_list::resize(3) - changes the number of elements stored (public member function) std::forward_list::swap(3) - swaps the contents (public member function)

Operations std::forward_list::merge(3) - merges two sorted lists (public member function) std::forward_list::splice_aft - moves elements from another forward_list (public member function) std::forward_list::remove(3), std::forward_list::remove_if(3) - removes elements satisfying specific criteria (public member function) std::forward_list::reverse(3) - reverses the order of the elements (public member function) std::forward_list::unique(3) - removes consecutive duplicate elements (public member function) std::forward_list::sort(3) - sorts the elements (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3), operator<(3), operator<=(3), operator>(3), operator>=(3) - lexicographically compares the values in the forward_list (function template) std::swap(std::forward_list)(3) [C++11] - specializes the std::swap algorithm (function template)

5.3.5 std::list

NAME

std::list - std::list is a container that supports constant time insertion and removal of elements from anywhere in the container. Fast random access is not supported. It is usually implemented as double-linked list. Compared to std::forward_list this container provides bidirectional iteration capability while being less space efficient.

SYNOPSIS

#include <list>

```
 template<
    class T,
    class Allocator = std::allocator<T>
> class list;
```

MEMBER FUNCTIONS

std::list::list(3) - constructs the list (public member function) std::list:: list(3) - destructs the list (public member function) std::list::operator=(3) - assigns values to the container (public member function) std::list::assign(3) - assigns values to the container (public member function) std::list::get_allocator(3) - returns the associated allocator (public member function)

Element access std::list::front(3) - access the first element (public member function) std::list::back(3) - access the last element (public member function)

Iterators std::list::begin(3), std::list::cbegin(3) - returns an iterator to the beginning (public member function) std::list::end(3), std::list::cend(3) - returns an iterator to the end (public member function) std::list::rbegin(3), std::list::crbegin(3) - returns a reverse iterator to the beginning (public member function) std::list::rend(3), std::list::crend(3) - returns a reverse iterator to the end (public member function)

Capacity std::list::empty(3) - checks whether the container is empty (public member function) std::list::size(3) - returns the number of elements (public member function) std::list::max_size(3) - returns the maximum possible number of elements (public member function)

Modifiers std::list::clear(3) - clears the contents (public member function) std::list::insert(3) - inserts elements (public member function) std::list::emplace(3) [C++11] - constructs element in-place (public member function) std::list::erase(3) - erases elements (public member function) std::list::push_back(3) - adds elements to the end (public member function) std::list::emplace_back(3) [C++11] - constructs elements in-place at the end (public member function) std::list::pop_back(3) - removes the last element (public member function) std::list::push_front(3) - inserts elements to the beginning (public member function) std::list::emplace_front(3) [C++11] - constructs elements in-place at the beginning (public member function) std::list::pop_front(3) - removes the first element (public member function) std::list::resize(3) - changes the number of elements stored (public member function) std::list::swap(3) - swaps the contents (public member function)

Operations std::list::merge(3) - merges two sorted lists (public member function) std::list::splice(3) - moves elements from another list (public member function) std::list::remove(3), std::list::remove_if(3) - removes elements satisfying specific criteria (public member function) std::list::reverse(3) - reverses the order of the elements (public member function) std::list::unique(3) - removes consecutive duplicate elements (public member function) std::list::sort(3) - sorts the elements (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3), operator<(3), operator<=(3), operator>(3), operator>=(3) - lexicographically compares the values in the list (function template) std::swap(std::list)(3) - specializes the std::swap algorithm (function template)

5.3.6 std::set

NAME

std::set - std::set is an associative container that contains a sorted set of unique objects of type Key. Sorting is done using the key comparison function Compare. Search, removal, and insertion operations have logarithmic complexity. Sets are usually implemented as red-black trees.

SYNOPSIS

#include <set>

```
 template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

MEMBER FUNCTIONS

std::set::set(3) - constructs the set (public member function) std::set:: set(3) - destructs the set (public member function) std::set::operator=(3) - assigns values to the container (public member function) std::set::get_allocator(3) - returns the associated allocator (public member function)

Iterators std::set::begin(3), std::set::cbegin(3) - returns an iterator to the beginning (public member function) std::set::end(3), std::set::cend(3) - returns an iterator to the end (public member function) std::set::rbegin(3), std::set::crbegin(3) - returns a reverse iterator to the beginning (public member function) std::set::rend(3), std::set::crend(3) - returns a reverse iterator to the end (public member function)

Capacity std::set::empty(3) - checks whether the container is empty (public member function) std::set::size(3) - returns the number of elements (public member function) std::set::max_size(3) - returns the maximum possible number of elements (public member function)

Modifiers std::set::clear(3) - clears the contents (public member function) std::set::insert(3) - inserts elements (public member function) std::set::emplace(3) [C++11] - constructs element in-place (public member function) std::set::emplace_hint(3) [C++11] - constructs elements in-place using a hint (public member function) std::set::erase(3) - erases elements (public member function) std::set::swap(3) - swaps the contents (public member function)

Lookup std::set::count(3) - returns the number of elements matching specific key (public member function) std::set::find(3) - finds element with specific key (public member function) std::set::equal_range(3) - returns range of elements matching a specific key (public member function) std::set::lower_bound(3) - returns an iterator to the first element not less than the given key (public member function) std::set::upper_bound(3) - returns an iterator to the first element greater than the given key (public member function)

Observers std::set::key_comp(3) - returns the function that compares keys (public member function) std::set::value_comp(3) - returns the function that compares keys in objects of type value_type (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3), operator<(3), operator<=(3), operator>(3), operator>=(3) - lexicographically compares the values in the set (function template) std::swap(std::set)(3) - specializes the std::swap algorithm (function template)

5.3.7 std::map

NAME

std::map - std::map is a sorted associative container that contains key-value pairs with unique keys. Keys are sorted by using the comparison function Compare. Search, removal, and insertion operations have logarithmic complexity. Maps are usually implemented as red-black trees.

SYNOPSIS

#include <map>

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

MEMBER FUNCTIONS

std::map::map(3) - constructs the map (public member function) std::map:: map(3) - destructs the map (public member function) std::map::operator=(3) - assigns values to the container (public member function) std::map::get_allocator(3) - returns the associated allocator (public member function)

Element access std::map::at(3) [C++11] - access specified element with bounds checking (public member function) std::map::operator[](3) - access specified element (public member function)

Iterators std::map::begin(3), std::map::cbegin(3) - returns an iterator to the beginning (public member function) std::map::end(3), std::map::cend(3) - returns an iterator to the end (public member function) std::map::rbegin(3), std::map::crbegin(3) - returns a reverse iterator to the beginning (public member function) std::map::rend(3), std::map::crend(3) - returns a reverse iterator to the end (public member function)

Capacity std::map::empty(3) - checks whether the container is empty (public member function) std::map::size(3) - returns the number of elements (public member function) std::map::max_size(3) - returns the maximum possible number of elements (public member function)

Modifiers std::map::clear(3) - clears the contents (public member function) std::map::insert(3) - inserts elements (public member function) std::map::insert_or_assign(3) [C++17] - inserts an element or assigns to the current element if the key already exists (public member function) std::map::emplace(3) [C++11] - constructs element in-place (public member function) std::map::emplace_hint(3) [C++11] - constructs elements in-place using a hint (public member function) std::map::try_emplace(3) [C++17] - inserts in-place if the key does not exist, does nothing if the key exists (public member function) std::map::erase(3) - erases elements (public member function) std::map::swap(3) - swaps the contents (public member function)

Lookup std::map::count(3) - returns the number of elements matching specific key (public member function) std::map::find(3) - finds element with specific key (public member function) std::map::equal_range(3) - returns range of elements matching a specific key (public member function) std::map::lower_bound(3) - returns an iterator to the first element not less than the given key (public member function) std::map::upper_bound(3) - returns an iterator to the first element greater than the given key (public member function)

Observers std::map::key_comp(3) - returns the function that compares keys (public member function) std::map::value_comp(3) - returns the function that compares keys in objects of type value_type (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3), operator<(3), operator<=(3), operator>(3), operator>=(3) - lexicographically compares the values in the map (function template) std::swap(std::map)(3) - specializes the std::swap algorithm (function template)

5.3.8 std::multiset

NAME

std::multiset - Multiset is an associative container that contains a sorted set of objects of type Key. Unlike set, multiple keys with equal values are allowed. Sorting is done using the key comparison function Compare. Search, insertion, and removal operations have logarithmic complexity.

SYNOPSIS

#include <set>

```
 template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class multiset;
```

MEMBER FUNCTIONS

std::multiset::multiset(3) - constructs the multiset (public member function) std::multiset:: multiset(3) - destructs the multiset (public member function) std::multiset::operator=(3) - assigns values to the container (public member function) std::multiset::get_allocator(3) - returns the associated allocator (public member function)

Iterators std::multiset::begin(3), std::multiset::cbegin(3) - returns an iterator to the beginning (public member function) std::multiset::end(3), std::multiset::cend(3) - returns an iterator to the end (public member function) std::multiset::rbegin(3), std::multiset::crbegin(3) - returns a reverse iterator to the beginning (public member function) std::multiset::rend(3), std::multiset::crend(3) - returns a reverse iterator to the end (public member function)

Capacity std::multiset::empty(3) - checks whether the container is empty (public member function) std::multiset::size(3) - returns the number of elements (public member function) std::multiset::max_size(3) - returns the maximum possible number of elements (public member function)

Modifiers std::multiset::clear(3) - clears the contents (public member function) std::multiset::insert(3) - inserts elements (public member function) std::multiset::emplace(3) [C++11] - constructs element in-place (public member function) std::multiset::emplace_hint(3) [C++11] - constructs elements in-place using a hint (public member function) std::multiset::erase(3) - erases elements (public member function) std::multiset::swap(3) - swaps the contents (public member function)

Lookup std::multiset::count(3) - returns the number of elements matching specific key (public member function) std::multiset::find(3) - finds element with specific key (public member function) std::multiset::equal_range(- returns range of elements matching a specific key (public member function) std::multiset::lower_bound(3) - returns an iterator to the first element not less than the given key (public member function) std::multiset::upper_bound(- returns an iterator to the first element greater than the given key (public member function)

Observers std::multiset::key_comp(3) - returns the function that compares keys (public member function) std::multiset::value_comp(3) - returns the function that compares keys in objects of type value_type (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3), operator<(3), operator<=(3), operator>(3), operator>=(3) - lexicographically compares the values in the multiset (function template) std::swap(std::multiset)(3) - specializes the std::swap algorithm (function template)

5.3.9 std::multimap

NAME

std::multimap - Multimap is an associative container that contains a sorted list of key-value pairs. Sorting is done according to the comparison function Compare, applied to the keys. Search, insertion, and removal operations have logarithmic complexity.

SYNOPSIS

#include <map>

```
 template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class multimap;
```

MEMBER FUNCTIONS

std::multimap::multimap(3) - constructs the multimap (public member function) std::multimap:: multimap(3) - destructs the multimap (public member function) std::multimap::operator=(3) - assigns values to the container (public member function) std::multimap::get_allocator(3) - returns the associated allocator (public member function)

Iterators std::multimap::begin(3), std::multimap::cbegin(3) - returns an iterator to the beginning (public member function) std::multimap::end(3), std::multimap::cend(3) - returns an iterator to the end (public member function) std::multimap::rbegin(3), std::multimap::crbegin(3) - returns a reverse iterator to the beginning (public member function) std::multimap::rend(3), std::multimap::crend(3) - returns a reverse iterator to the end (public member function)

Capacity std::multimap::empty(3) - checks whether the container is empty (public member function) std::multimap::size(3) - returns the number of elements (public member function) std::multimap::max_size(3) - returns the maximum possible number of elements (public member function)

Modifiers std::multimap::clear(3) - clears the contents (public member function) std::multimap::insert(3) - inserts elements (public member function) std::multimap::emplace(3) [C++11] - constructs element in-place (public member function) std::multimap::emplace_hint(3) [C++11] - constructs elements in-place using a hint (public member function) std::multimap::erase(3) - erases elements (public member function) std::multimap::swap(3) - swaps the contents (public member function)

Lookup std::multimap::count(3) - returns the number of elements matching specific key (public member function) std::multimap::find(3) - finds element with specific key (public member function) std::multimap::equal_rang
- returns range of elements matching a specific key (public member function) std::multimap::lower_bound(3)
- returns an iterator to the first element not less than the given key (public member function) std::multimap::upper_boun
- returns an iterator to the first element greater than the given key (public member function)

Observers std::multimap::key_comp(3) - returns the function that compares keys (public member function) std::multimap::value_comp(3) - returns the function that compares keys in objects of type value_type (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3), operator<(3), operator<=(3), operator>(3), operator>=(3) - lexicographically compares the values in the multimap (function template) std::swap(std::multimap)(3) - specializes the std::swap algorithm (function template)

5.3.10 std::unordered_set

NAME

std::unordered_set - Unordered set is an associative container that contains set of unique objects of type Key. Search, insertion, and removal have average constant-time complexity.

SYNOPSIS

#include <unordered_set>

```
 template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal\_to<Key>,
    class Allocator = std::allocator<Key>

> class unordered\_set; [since C++11]
```

MEMBER FUNCTIONS

std::unordered_set::unordered_set(3) - constructs the unordered_set (public member function) std::unordered_set:: unordered_set(3) - destructs the unordered_set (public member function) std::unordered_set::operator=(3)
- assigns values to the container (public member function) std::unordered_set::get_allocator(3) - returns the associated allocator (public member function)

Iterators std::unordered_set::begin(3), std::unordered_set::cbegin(3) - returns an iterator to the beginning (public member function) std::unordered_set::end(3), std::unordered_set::cend(3) - returns an iterator to the end (public member function)

Capacity std::unordered_set::empty(3) - checks whether the container is empty (public member function) std::unordered_set::size(3) - returns the number of elements (public member function) std::unordered_set::max_siz
- returns the maximum possible number of elements (public member function)

Modifiers std::unordered_set::clear(3) - clears the contents (public member function) std::unordered_set::insert(3) - inserts elements (public member function) std::unordered_set::emplace(3) - constructs element in-place (public member function) std::unordered_set::emplace_hint(3) - constructs elements in-place using a hint (public member function) std::unordered_set::erase(3) - erases elements (public member function) std::unordered_set::swap(3) - swaps the contents (public member function)

Lookup std::unordered_set::count(3) - returns the number of elements matching specific key (public member function) std::unordered_set::find(3) - finds element with specific key (public member function) std::unordered_set::equal_range(3) - returns range of elements matching a specific key (public member function)

Bucket interface std::unordered_set::begin(int) cbegin(int)(3) - returns an iterator to the beginning of the specified bucket (public member function) std::unordered_set::end(int) cend(int)(3) - returns an iterator to the end of the specified bucket (public member function) std::unordered_set::bucket_count(3) - returns the number of buckets (public member function) std::unordered_set::max_bucket_count(3) - returns the maximum number of buckets (public member function) std::unordered_set::bucket_size(3) - returns the number of elements in specific bucket (public member function) std::unordered_set::bucket(3) - returns the bucket for specific key (public member function)

Hash policy std::unordered_set::load_factor(3) - returns average number of elements per bucket (public member function) std::unordered_set::max_load_factor(3) - manages maximum average number of elements per bucket (public member function) std::unordered_set::rehash(3) - reserves at least the specified number of buckets.This regenerates the hash table. (public member function) std::unordered_set::reserve(3) - reserves space for at least the specified number of elements.This regenerates the hash table. (public member function)

Observers std::unordered_set::hash_function(3) - returns function used to hash the keys (public member function) std::unordered_set::key_eq(3) - returns the function used to compare keys for equality (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3) - compares the values in the unordered_set (function template) std::swap(std::unordered_
[C++11] - specializes the std::swap algorithm (function template)

5.3.11 std::unordered_map

NAME

std::unordered_map - Unordered map is an associative container that contains key-value pairs with unique keys. Search, insertion, and removal of elements have average constant-time complexity.

SYNOPSIS

#include <unordered_map>

```
template<
   class Key,
   class T,
   class Hash = std::hash<Key>,
   class KeyEqual = std::equal\_to<Key>,
```

```
    class Allocator = std::allocator< std::pair<const Key, T> >

> class unordered\_map; [since C++11]
```

MEMBER FUNCTIONS

std::unordered_map::unordered_map(3) - constructs the unordered_map (public member function) std::unordered_map: ordered_map(3) - destructs the unordered_map (public member function) std::unordered_map::operator=(3) - assigns values to the container (public member function) std::unordered_map::get_allocator(3) - returns the associated allocator (public member function)

Iterators std::unordered_map::begin(3), std::unordered_map::cbegin(3) - returns an iterator to the beginning (public member function) std::unordered_map::end(3), std::unordered_map::cend(3) - returns an iterator to the end (public member function)

Capacity std::unordered_map::empty(3) - checks whether the container is empty (public member function) std::unordered_map::size(3) - returns the number of elements (public member function) std::unordered_map::m - returns the maximum possible number of elements (public member function)

Modifiers std::unordered_map::clear(3) - clears the contents (public member function) std::unordered_map::insert(3) - inserts elements (public member function) std::unordered_map::insert_or_assign(3) [C++17] - inserts an element or assigns to the current element if the key already exists (public member function) std::unordered_map::emplace(3) - constructs element in-place (public member function) std::unordered_map::emplace_ - constructs elements in-place using a hint (public member function) std::unordered_map::try_emplace(3) [C++17] - inserts in-place if the key does not exist, does nothing if the key exists (public member function) std::unordered_map::erase(3) - erases elements (public member function) std::unordered_map::swap(3) - swaps the contents (public member function)

Lookup std::unordered_map::at(3) - access specified element with bounds checking (public member function) std::unordered_map::operator[](3) - access specified element (public member function) std::unordered_map::count(3) - returns the number of elements matching specific key (public member function) std::unordered_map::find(3) - finds element with specific key (public member function) std::unordered_map::equal_range(3) - returns range of elements matching a specific key (public member function)

Bucket interface std::unordered_map::begin(int) cbegin(int)(3) - returns an iterator to the beginning of the specified bucket (public member function) std::unordered_map::end(int) cend(int)(3) - returns an iterator to the end of the specified bucket (public member function) std::unordered_map::bucket_count(3) - returns the number of buckets (public member function) std::unordered_map::max_bucket_count(3) - returns the maximum number of buckets (public member function) std::unordered_map::bucket_size(3) - returns the number of elements in specific bucket (public member function) std::unordered_map::bucket(3) - returns the bucket for specific key (public member function)

Hash policy std::unordered_map::load_factor(3) - returns average number of elements per bucket (public member function) std::unordered_map::max_load_factor(3) - manages maximum average number of elements per bucket (public member function) std::unordered_map::rehash(3) - reserves at least the specified number of buckets.This regenerates the hash table. (public member function) std::unordered_map::reserve(3) - reserves space for at least the specified number of elements.This regenerates the hash table. (public member function)

Observers std::unordered_map::hash_function(3) - returns function used to hash the keys (public member function) std::unordered_map::key_eq(3) - returns the function used to compare keys for equality (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3) - compares the values in the unordered_map (function template) std::swap(std::unordered
[C++11] - specializes the std::swap algorithm (function template)

5.3.12 std::unordered_multiset

NAME

std::unordered_multiset - Unordered multiset is an associative container that contains set of possibly non-unique objects of type Key. Search, insertion, and removal have average constant-time complexity.

SYNOPSIS

#include <unordered_set>

```
 template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal\_to<Key>,
    class Allocator = std::allocator<Key>

> class unordered\_multiset; [since C++11]
```

MEMBER FUNCTIONS

std::unordered_multiset::unordered_multiset(3) - constructs the unordered_multiset (public member function) std::unordered_multiset:: unordered_multiset(3) - destructs the unordered_multiset (public member function) std::unordered_multiset::operator=(3) - assigns values to the container (public member function) std::unordered_multiset::get_allocator(3) - returns the associated allocator (public member function)

Iterators std::unordered_multiset::begin(3), std::unordered_multiset::cbegin(3) - returns an iterator to the beginning (public member function) std::unordered_multiset::end(3), std::unordered_multiset::cend(3) - returns an iterator to the end (public member function)

Capacity std::unordered_multiset::empty(3) - checks whether the container is empty (public member function) std::unordered_multiset::size(3) - returns the number of elements (public member function) std::unordered_multiset::max_size(3) - returns the maximum possible number of elements (public member function)

Modifiers std::unordered_multiset::clear(3) - clears the contents (public member function) std::unordered_multiset::in
- inserts elements (public member function) std::unordered_multiset::emplace(3) - constructs element in-place (public member function) std::unordered_multiset::emplace_hint(3) - constructs elements in-place using a hint (public member function) std::unordered_multiset::erase(3) - erases elements (public member function) std::unordered_multiset::swap(3) - swaps the contents (public member function)

Lookup std::unordered_multiset::count(3) - returns the number of elements matching specific key (public member function) std::unordered_multiset::find(3) - finds element with specific key (public member function) std::unordered_multiset::equal_range(3) - returns range of elements matching a specific key (public member function)

Bucket interface std::unordered_multiset::begin(int) cbegin(int)(3) - returns an iterator to the beginning of the specified bucket (public member function) std::unordered_multiset::end(int) cend(int)(3) - returns an iterator to the end of the specified bucket (public member function) std::unordered_multiset::bucket_count(3) - returns the number of buckets (public member function) std::unordered_multiset::max_bucket_count(3) - returns the maximum number of buckets (public member function) std::unordered_multiset::bucket_size(3) - returns the number of elements in specific bucket (public member function) std::unordered_multiset::bucket(3) - returns the bucket for specific key (public member function)

Hash policy std::unordered_multiset::load_factor(3) - returns average number of elements per bucket (public member function) std::unordered_multiset::max_load_factor(3) - manages maximum average number of elements per bucket (public member function) std::unordered_multiset::rehash(3) - reserves at least the specified number of buckets.This regenerates the hash table. (public member function) std::unordered_multiset::reserve(3) - reserves space for at least the specified number of elements.This regenerates the hash table. (public member function)

Observers std::unordered_multiset::hash_function(3) - returns function used to hash the keys (public member function) std::unordered_multiset::key_eq(3) - returns the function used to compare keys for equality (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3) - compares the values in the unordered_multiset (function template) std::swap(std::unordered_multiset)(3) [C++11] - specializes the std::swap algorithm (function template)

5.3.13 std::unordered_multimap

NAME

std::unordered_multimap - Unordered multimap is an unordered associative container that supports equivalent keys (an unordered_multimap may contain multiple copies of each key value) and that associates values of another type with the keys. The unordered_multimap class supports forward iterators. Search, insertion, and removal have average constant-time complexity.

SYNOPSIS

#include <unordered_map>

```
 template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal\_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >

> class unordered\_multimap; [since C++11]
```

MEMBER FUNCTIONS

std::unordered_multimap::unordered_multimap(3) - constructs the unordered_multimap (public member function) std::unordered_multimap:: unordered_multimap(3) - destructs the unordered_multimap (public member function) std::unordered_multimap::operator=(3) - assigns values to the container (public member function) std::unordered_multimap::get_allocator(3) - returns the associated allocator (public member function)

Iterators std::unordered_multimap::begin(3), std::unordered_multimap::cbegin(3) - returns an iterator to the beginning (public member function) std::unordered_multimap::end(3), std::unordered_multimap::cend(3) - returns an iterator to the end (public member function)

Capacity std::unordered_multimap::empty(3) - checks whether the container is empty (public member function) std::unordered_multimap::size(3) - returns the number of elements (public member function) std::unordered_multimap::max_size(3) - returns the maximum possible number of elements (public member function)

Modifiers std::unordered_multimap::clear(3) - clears the contents (public member function) std::unordered_multimap
- inserts elements (public member function) std::unordered_multimap::emplace(3) - constructs element in-place (public member function) std::unordered_multimap::emplace_hint(3) - constructs elements in-place using a hint (public member function) std::unordered_multimap::erase(3) - erases elements (public member function) std::unordered_multimap::swap(3) - swaps the contents (public member function)

Lookup std::unordered_multimap::count(3) - returns the number of elements matching specific key (public member function) std::unordered_multimap::find(3) - finds element with specific key (public member function) std::unordered_multimap::equal_range(3) - returns range of elements matching a specific key (public member function)

Bucket interface std::unordered_multimap::begin(int) cbegin(int)(3) - returns an iterator to the beginning of the specified bucket (public member function) std::unordered_multimap::end(int) cend(int)(3)
- returns an iterator to the end of the specified bucket (public member function) std::unordered_multimap::bucket_count
- returns the number of buckets (public member function) std::unordered_multimap::max_bucket_count(3)
- returns the maximum number of buckets (public member function) std::unordered_multimap::bucket_size(3)
- returns the number of elements in specific bucket (public member function) std::unordered_multimap::bucket(3)
- returns the bucket for specific key (public member function)

Hash policy std::unordered_multimap::load_factor(3) - returns average number of elements per bucket (public member function) std::unordered_multimap::max_load_factor(3) - manages maximum average number of elements per bucket (public member function) std::unordered_multimap::rehash(3) - reserves at least the specified number of buckets.This regenerates the hash table. (public member function) std::unordered_multimap::reserve(3) - reserves space for at least the specified number of elements.This regenerates the hash table. (public member function)

Observers std::unordered_multimap::hash_function(3) - returns function used to hash the keys (public member function) std::unordered_multimap::key_eq(3) - returns the function used to compare keys for equality (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3) - compares the values in the unordered_multimap (function template) std::swap(std::unordered_multimap)(3) [C++11] - specializes the std::swap algorithm (function template)

5.3.14 std::stack

NAME

std::stack - The std::stack class is a container adapter that gives the programmer the functionality of a stack - specifically, a FILO (first-in, last-out) data structure.

SYNOPSIS

#include <stack>

```
 template<
    class T,
    class Container = std::deque<T>
> class stack;
```

MEMBER FUNCTIONS

std::stack::stack(3) - constructs the stack (public member function) std::stack:: stack(3) - destructs the stack (public member function) std::stack::operator=(3) - assigns values to the container adaptor (public member function)

Element access std::stack::top(3) - accesses the top element (public member function)

Capacity std::stack::empty(3) - checks whether the underlying container is empty (public member function) std::stack::size(3) - returns the number of elements (public member function)

Modifiers std::stack::push(3) - inserts element at the top (public member function) std::stack::emplace(3) [C++11] - constructs element in-place at the top (public member function) std::stack::pop(3) - removes the top element (public member function) std::stack::swap(3) - swaps the contents (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3), operator<(3), operator<=(3), operator>(3), operator>=(3) - lexicographically compares the values in the stack (function template) std::swap(std::stack)(3) - specializes the std::swap algorithm (function template)

5.3.15 std::queue

NAME

std::queue - The std::queue class is a container adapter that gives the programmer the functionality of a queue - specifically, a FIFO (first-in, first-out) data structure.

SYNOPSIS

#include <queue>

```
template<
   class T,
   class Container = std::deque<T>
> class queue;
```

MEMBER FUNCTIONS

std::queue::queue(3) - constructs the queue (public member function) std::queue:: queue(3) - destructs the queue (public member function) std::queue::operator=(3) - assigns values to the container adaptor (public member function)

Element access std::queue::front(3) - access the first element (public member function) std::queue::back(3) - access the last element (public member function)

Capacity std::queue::empty(3) - checks whether the underlying container is empty (public member function) std::queue::size(3) - returns the number of elements (public member function)

Modifiers std::queue::push(3) - inserts element at the end (public member function) std::queue::emplace(3) [C++11] - constructs element in-place at the end (public member function) std::queue::pop(3) - removes the first element (public member function) std::queue::swap(3) - swaps the contents (public member function)

NON-MEMBER FUNCTIONS

operator==(3), operator!=(3), operator<(3), operator<=(3), operator>(3), operator>=(3) - lexicographically compares the values in the queue (function template) std::swap(std::queue)(3) - specializes the std::swap algorithm (function template)

5.3.16 std::priority_queue

NAME

std::priority_queue - A priority queue is a container adaptor that provides constant time extraction of the largest (by default) element, at the expense of logarithmic insertion.

SYNOPSIS

#include <queue>

```
template<
   class T,
   class Container = std::vector<T>,
   class Compare = std::less<typename Container::value\_type>
> class priority\_queue;
```

MEMBER FUNCTIONS

std::priority_queue::priority_queue(3) - constructs the priority_queue (public member function) std::priority_queue:: priority_queue(3) - destructs the priority_queue (public member function) std::priority_queue::operator=(3) - assigns values to the container adaptor (public member function)

Element access std::priority_queue::top(3) - accesses the top element (public member function)

Capacity std::priority_queue::empty(3) - checks whether the underlying container is empty (public member function) std::priority_queue::size(3) - returns the number of elements (public member function)

Modifiers std::priority_queue::push(3) - inserts element and sorts the underlying container (public member function) std::priority_queue::emplace(3) [C++11] - constructs element in-place and sorts the underlying container (public member function) std::priority_queue::pop(3) - removes the top element (public member function) std::priority_queue::swap(3) - swaps the contents (public member function)

NON-MEMBER FUNCTIONS

std::swap(std::priority_queue)(3) - specializes the std::swap algorithm (function template)

5.4. Algorithms library

5.4.1 std::all_of

NAME

std::all_of - 1) Checks if unary predicate p returns true for all elements in the range [first, last).

SYNOPSIS

#include <algorithm>

```
 template< class InputIt, class UnaryPredicate >
bool all\_of( InputIt first, InputIt last, UnaryPredicate p ); [since C++11]
 template< class InputIt, class UnaryPredicate >
bool any\_of( InputIt first, InputIt last, UnaryPredicate p ); [since C++11]
 template< class InputIt, class UnaryPredicate >
bool none\_of( InputIt first, InputIt last, UnaryPredicate p ); [since C++11]
```

PARAMETERS

first, last - the range of elements to examine p - unary predicate . The signature of the predicate function should be equivalent to the following:

bool pred(const Type &a);

The signature does not need to have const &, but the function must not modify the objects passed to it. The type Type must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type.

Type requirements -InputIt must meet the requirements of InputIterator. -UnaryPredicate must meet the requirements of Predicate.

RETURN VALUE

1) true if unary predicate returns true for all elements in the range, false otherwise. Returns true if the range is empty.

2) true if unary predicate returns true for at least one element in the range, false otherwise. Returns false if the range is empty.

3) true if unary predicate returns true for no elements in the range, false otherwise. Returns true if the range is empty.

5.4.2 std::for_each

NAME

std::for_each - Applies the given function object f to the result of dereferencing every iterator in the range [first, last), in order.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt, class UnaryFunction >
UnaryFunction for\_each( InputIt first, InputIt last, UnaryFunction f );
```

PARAMETERS

first, last - the range to apply the function to f - function object, to be applied to the result of dereferencing every iterator in the range [first, last) The signature of the function should be equivalent to the following:

void fun(const Type &a);

The signature does not need to have const &. The type Type must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type.

Type requirements -InputIt must meet the requirements of InputIterator. -UnaryFunction must meet the requirements of MoveConstructible. Does not have to be CopyConstructible

RETURN VALUE

f [until C++11] std::move(f) [since C++11]

5.4.3 std::count

NAME

std::count - Returns the number of elements in the range [first, last) satisfying specific criteria. The first version counts the elements that are equal to value, the second version counts elements for which predicate p returns true.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt, class T >
typename iterator\_traits<InputIt>::difference\_type
```

```
    count( InputIt first, InputIt last, const T \&value );
 template< class InputIt, class UnaryPredicate >
typename iterator\_traits<InputIt>::difference\_type
    count\_if( InputIt first, InputIt last, UnaryPredicate p );
```

PARAMETERS

first, last - the range of elements to examine value - the value to search for p - unary predicate which returns true for the required elements. The signature of the predicate function should be equivalent to the following:

bool pred(const Type &a);

The signature does not need to have const &, but the function must not modify the objects passed to it. The type Type must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type.

Type requirements -InputIt must meet the requirements of InputIterator.

RETURN VALUE

number of elements satisfying the condition.

5.4.4 std::mismatch

NAME

std::mismatch - Returns the first mismatching pair of elements from two ranges: one defined by [first1, last1) and another defined by [first2,last2). If last2 is not provided (overloads (1) and (2)), it denotes first2 + (last1 - first1).

SYNOPSIS

#include <algorithm>

```
 template< class InputIt1, class InputIt2 >
std::pair<InputIt1,InputIt2>
    mismatch( InputIt1 first1, InputIt1 last1,
              InputIt2 first2 );
 template< class InputIt1, class InputIt2, class BinaryPredicate >
std::pair<InputIt1,InputIt2>
    mismatch( InputIt1 first1, InputIt1 last1,
              InputIt2 first2,
              BinaryPredicate p );
 template< class InputIt1, class InputIt2 >
std::pair<InputIt1,InputIt2>
    mismatch( InputIt1 first1, InputIt1 last1,

              InputIt2 first2, InputIt2 last2 ); [since C++14]
 template< class InputIt1, class InputIt2, class BinaryPredicate >
std::pair<InputIt1,InputIt2>
    mismatch( InputIt1 first1, InputIt1 last1,
              InputIt2 first2, InputIt2 last2,

              BinaryPredicate p ); [since C++14]
```

PARAMETERS

first1, last1 - the first range of the elements first2, last2 - the second range of the elements p - binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

bool pred(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function must not modify the objects passed to it. The types Type1 and Type2 must be such that objects of types InputIt1 and InputIt2 can be dereferenced and then implicitly converted to Type1 and Type2 respectively.

Type requirements -InputIt1 must meet the requirements of InputIterator. -InputIt2 must meet the requirements of InputIterator. -BinaryPredicate must meet the requirements of BinaryPredicate.

RETURN VALUE

std::pair with iterators to the first two non-equivalent elements.

If no mismatches are found when the comparison reaches last1, the pair holds last1 and the corresponding iterator from the second range. The behavior is undefined if the second range is shorter than the first range. [until C++14] If no mismatches are found when the comparison reaches last1 or last2, whichever happens first, the pair holds the end iterator and the corresponding iterator from the other range. [since C++14]

5.4.5 std::equal

NAME

std::equal - 1,2) Returns true if the range [first1, last1) is equal to the range [first2, first2 + (last1 - first1)), and false otherwise

SYNOPSIS

#include <algorithm>

```
 template< class InputIt1, class InputIt2 >
bool equal( InputIt1 first1, InputIt1 last1,
            InputIt2 first2 );
 template< class InputIt1, class InputIt2, class BinaryPredicate >
bool equal( InputIt1 first1, InputIt1 last1,
            InputIt2 first2, BinaryPredicate p );
 template< class InputIt1, class InputIt2 >
bool equal( InputIt1 first1, InputIt1 last1,

            InputIt2 first2, InputIt2 last2 ); [since C++14]
 template< class InputIt1, class InputIt2, class BinaryPredicate >
bool equal( InputIt1 first1, InputIt1 last1,
            InputIt2 first2, InputIt2 last2,

            BinaryPredicate p ); [since C++14]
```

PARAMETERS

first1, last1 - the first range of the elements to compare first2, last2 - the second range of the elements to compare p - binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

bool pred(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function must not modify the objects passed to it. The types Type1 and Type2 must be such that objects of types InputIt1 and InputIt2 can be dereferenced and then implicitly converted to Type1 and Type2 respectively.

Type requirements -InputIt1, InputIt2 must meet the requirements of InputIterator.

RETURN VALUE

3,4) If the length of the range [first1, last1) does not equal the length of the range [first2, last2), returns false

If the elements in the two ranges are equal, returns true.

Otherwise returns false.

5.4.6 std::find

NAME

std::find - Returns the first element in the range [first, last) that satisfies specific criteria:

SYNOPSIS

#include <algorithm>

```
 template< class InputIt, class T >
InputIt find( InputIt first, InputIt last, const T\& value );
 template< class InputIt, class UnaryPredicate >
InputIt find\_if( InputIt first, InputIt last,
                 UnaryPredicate p );
 template< class InputIt, class UnaryPredicate >
InputIt find\_if\_not( InputIt first, InputIt last,

                     UnaryPredicate q ); [since C++11]
```

PARAMETERS

first, last - the range of elements to examine value - value to compare the elements to p - unary predicate which returns true for the required element. The signature of the predicate function should be equivalent to the following:

bool pred(const Type &a);

The signature does not need to have const &, but the function must not modify the objects passed to it. The type Type must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type.

q - unary predicate which returns false for the required element. The signature of the predicate function should be equivalent to the following:

bool pred(const Type &a);

The signature does not need to have const &, but the function must not modify the objects passed to it. The type Type must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type.

Type requirements -InputIt must meet the requirements of InputIterator. -UnaryPredicate must meet the requirements of Predicate.

RETURN VALUE

Iterator to the first element satisfying the condition or last if no such element is found.

5.4.7 std::find_end

NAME

std::find_end - Searches for the last subsequence of elements [s_first, s_last) in the range [first, last). The first version uses operator== to compare the elements, the second version uses the given binary predicate p.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt1, class ForwardIt2 >
ForwardIt1 find\_end( ForwardIt1 first, ForwardIt1 last,
                     ForwardIt2 s\_first, ForwardIt2 s\_last );
 template< class ForwardIt1, class ForwardIt2, class BinaryPredicate >
ForwardIt1 find\_end( ForwardIt1 first, ForwardIt1 last,
                     ForwardIt2 s\_first, ForwardIt2 s\_last, BinaryPredicate p );
```

PARAMETERS

first, last - the range of elements to examine s_first, s_last - the range of elements to search for p - binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

bool pred(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function must not modify the objects passed to it. The types Type1 and Type2 must be such that objects of types ForwardIt1 and ForwardIt2 can be dereferenced and then implicitly converted to Type1 and Type2 respectively.

Type requirements -ForwardIt1 must meet the requirements of ForwardIterator. -ForwardIt2 must meet the requirements of ForwardIterator.

RETURN VALUE

Iterator to the beginning of last subsequence [s_first, s_last) in range [first, last).

If no such subsequence is found, last is returned. [until C++11] If [s_first, s_last) is empty or if no such subsequence is found, last is returned. [since C++11]

5.4.8 std::find_first_of

NAME

std::find_first_of - Searches the range [first, last) for any of the elements in the range [s_first, s_last). The first version uses operator== to compare the elements, the second version uses the given binary predicate p.

SYNOPSIS

#include <algorithm>

```
(1)
 template< class ForwardIt1, class ForwardIt2 >
ForwardIt1 find\_first\_of( ForwardIt1 first, ForwardIt1 last,

                          ForwardIt2 s\_first, ForwardIt2 s\_last ); [until C++11]
 template< class InputIt, class ForwardIt >
InputIt find\_first\_of( InputIt first, InputIt last,

                       ForwardIt s\_first, ForwardIt s\_last ); [since C++11]
(2)
 template< class ForwardIt1, class ForwardIt2, class BinaryPredicate >
ForwardIt1 find\_first\_of( ForwardIt1 first, ForwardIt1 last,

                          ForwardIt2 s\_first, ForwardIt2 s\_last, BinaryPredicate p ); [until C++11]
 template< class InputIt, class ForwardIt, class BinaryPredicate >
InputIt find\_first\_of( InputIt first, InputIt last,

                       ForwardIt s\_first, ForwardIt s\_last, BinaryPredicate p ); [since C++11]
```

PARAMETERS

first, last - the range of elements to examine s_first, s_last - the range of elements to search for p - binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

bool pred(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function must not modify the objects passed to it. The types Type1 and Type2 must be such that objects of types ForwardIt1 and ForwardIt2 can be dereferenced and then implicitly converted to Type1 and Type2 respectively.

Type requirements -InputIt must meet the requirements of InputIterator. -ForwardIt1 must meet the requirements of ForwardIterator. -ForwardIt2 must meet the requirements of ForwardIterator.

RETURN VALUE

Iterator to the first element in the range [first, last) that is equal to an element from the range [s_first; s_last). If no such element is found, last is returned.

5.4.9 std::adjacent_find

NAME

std::adjacent_find - Searches the range [first, last) for two consecutive identical elements. The first version uses operator== to compare the elements, the second version uses the given binary predicate p.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt >
ForwardIt adjacent\_find( ForwardIt first, ForwardIt last );
 template< class ForwardIt, class BinaryPredicate>
ForwardIt adjacent\_find( ForwardIt first, ForwardIt last, BinaryPredicate p );
```

PARAMETERS

first, last - the range of elements to examine p - binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

bool pred(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type ForwardIt can be dereferenced and then implicitly converted to both of them.

Type requirements -ForwardIt must meet the requirements of ForwardIterator.

RETURN VALUE

an iterator to the first of the identical elements, that is, the first iterator it such that *it == *(it+1) for the first version or p(*it, *(it + 1)) != false for the second version.

If no such elements are found, last is returned

5.4.10 std::search

NAME

std::search - Searches for the first occurrence of the subsequence of elements [s_first, s_last) in the range [first, last - (s_last - s_first)). The first version uses operator== to compare the elements, the second version uses the given binary predicate p.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt1, class ForwardIt2 >
ForwardIt1 search( ForwardIt1 first, ForwardIt1 last,
                   ForwardIt2 s\_first, ForwardIt2 s\_last );
 template< class ForwardIt1, class ForwardIt2, class BinaryPredicate >
ForwardIt1 search( ForwardIt1 first, ForwardIt1 last,
                   ForwardIt2 s\_first, ForwardIt2 s\_last, BinaryPredicate p );
```

PARAMETERS

first, last - the range of elements to examine s_first, s_last - the range of elements to search for p - binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

bool pred(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function must not modify the objects passed to it. The types Type1 and Type2 must be such that objects of types ForwardIt1 and ForwardIt2 can be dereferenced and then implicitly converted to Type1 and Type2 respectively.

Type requirements -ForwardIt1, ForwardIt2 must meet the requirements of ForwardIterator.

RETURN VALUE

Iterator to the beginning of first subsequence [s_first, s_last) in the range [first, last - (s_last - s_first)). If no such subsequence is found, last is returned. If [s_first, s_last) is empty, first is returned. [since C++11]

5.4.11 std::search_n

NAME

std::search_n - Searches the range [first, last) for the first sequence of count identical elements, each equal to the given value value. The first version uses operator== to compare the elements, the second version uses the given binary predicate p.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt, class Size, class T >
ForwardIt search\_n( ForwardIt first, ForwardIt last, Size count, const T\& value );
 template< class ForwardIt, class Size, class T, class BinaryPredicate >
ForwardIt search\_n( ForwardIt first, ForwardIt last, Size count, const T\& value,
                     BinaryPredicate p );
```

PARAMETERS

first, last - the range of elements to examine count - the length of the sequence to search for value - the value of the elements to search for p - binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

bool pred(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function must not modify the objects passed to it. The type Type1 must be such that an object of type ForwardIt can be dereferenced and then implicitly converted to Type1. The type Type2 must be such that an object of type T can be implicitly converted to Type2.

Type requirements -ForwardIt must meet the requirements of ForwardIterator.

RETURN VALUE

Iterator to the beginning of the found sequence in the range [first, last). If no such sequence is found, last is returned.

5.4.12 std::copy

NAME

std::copy - Copies the elements in the range, defined by [first, last), to another range beginning at d_first. The second function only copies the elements for which the predicate pred returns true. The order of the elements that are not removed is preserved.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt, class OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt d\_first );
 template< class InputIt, class OutputIt, class UnaryPredicate >
OutputIt copy\_if( InputIt first, InputIt last,
                  OutputIt d\_first,

                  UnaryPredicate pred ); [since C++11]
```

PARAMETERS

first, last - the range of elements to copy d_first - the beginning of the destination range. pred - unary predicate which returns true for the required elements. The signature of the predicate function should be equivalent to the following:

bool pred(const Type &a);

The signature does not need to have const &, but the function must not modify the objects passed to it. The type Type must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type.

Type requirements -InputIt must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator. -UnaryPredicate must meet the requirements of Predicate.

RETURN VALUE

Output iterator to the element in the destination range, one past the last element copied.

5.4.13 std::copy_n

NAME

std::copy_n - Copies exactly count values from the range beginning at first to the range beginning at result, if count>0. Does nothing otherwise.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt, class Size, class OutputIt >
OutputIt copy\_n( InputIt first, Size count, OutputIt result ); [since C++11]
```

PARAMETERS

first - the beginning of the range of elements to copy from count - number of the elements to copy result - the beginning of the destination range Type requirements -InputIt must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

Iterator in the destination range, pointing past the last element copied if count>0 or result otherwise.

5.4.14 std::copy_backward

NAME

std::copy_backward - Copies the elements from the range, defined by [first, last), to another range ending at d_last. The elements are copied in reverse order (the last element is copied first), but their relative order is preserved.

SYNOPSIS

#include <algorithm>

```
 template< class BidirIt1, class BidirIt2 >
BidirIt2 copy\_backward( BidirIt1 first, BidirIt1 last, BidirIt2 d\_last );
```

PARAMETERS

first, last - the range of the elements to copy d_last - end of the destination range.. Type requirements -BidirIt must meet the requirements of BidirectionalIterator.

RETURN VALUE

iterator to the last element copied.

5.4.15 std::move

NAME

std::move - Moves the elements in the range [first, last), to another range beginning at d_first. After this operation the elements in the moved-from range will still contain valid values of the appropriate type, but not necessarily the same values as before the move.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt, class OutputIt >
OutputIt move( InputIt first, InputIt last, OutputIt d\_first ); [since C++11]
```

PARAMETERS

first, last - the range of elements to move d_first - the beginning of the destination range. If d_first is within [first, last), std::move_backward must be used instead of std::move. Type requirements -InputIt must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

Output iterator to the element past the last element moved (d_first + (last - first))

5.4.16 std::move_backward

NAME

std::move_backward - Moves the elements from the range [first, last), to another range ending at d_last. The elements are moved in reverse order (the last element is moved first), but their relative order is preserved.

SYNOPSIS

#include <algorithm>

```
 template< class BidirIt1, class BidirIt2 >
BidirIt2 move\_backward( BidirIt1 first, BidirIt1 last, BidirIt2 d\_last ); [since C++11]
```

PARAMETERS

first, last - the range of the elements to move d_last - end of the destination range Type requirements -BidirIt1, BidirIt2 must meet the requirements of BidirectionalIterator.

RETURN VALUE

Iterator in the destination range, pointing at the last element moved.

5.4.17 std::fill

NAME

std::fill - Assigns the given value to the elements in the range [first, last).

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt, class T >
void fill( ForwardIt first, ForwardIt last, const T\& value );
```

PARAMETERS

first, last - the range of elements to modify value - the value to be assigned Type requirements - ForwardIt must meet the requirements of ForwardIterator.

RETURN VALUE

(none)

5.4.18 std::fill_n

NAME

std::fill_n - Assigns the given value value to the first count elements in the range beginning at first if count > 0. Does nothing otherwise.

SYNOPSIS

#include <algorithm>

```

 template< class OutputIt, class Size, class T >
void fill\_n( OutputIt first, Size count, const T\& value ); [until C++11]
 template< class OutputIt, class Size, class T >
OutputIt fill\_n( OutputIt first, Size count, const T\& value ); [since C++11]
```

PARAMETERS

first - the beginning of the range of elements to modify count - number of elements to modify value - the value to be assigned Type requirements -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

(none) [until C++11] Iterator one past the last element assigned if count > 0, first otherwise. [since C++11]

5.4.19 std::transform

NAME

std::transform - std::transform applies the given function to a range and stores the result in another range, beginning at d_first.

SYNOPSIS

#include <algorithm>

```
template< class InputIt, class OutputIt, class UnaryOperation >
OutputIt transform( InputIt first1, InputIt last1, OutputIt d\_first,
                    UnaryOperation unary\_op );
template< class InputIt1, class InputIt2, class OutputIt, class BinaryOperation >
OutputIt transform( InputIt1 first1, InputIt1 last1, InputIt2 first2,
                    OutputIt d\_first, BinaryOperation binary\_op );
```

PARAMETERS

first1, last1 - the first range of elements to transform first2 - the beginning of the second range of elements to transform d_first - the beginning of the destination range, may be equal to first1 or first2 unary_op - unary operation function object that will be applied. The signature of the function should be equivalent to the following:

Ret fun(const Type &a);

The signature does not need to have const &. The type Type must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type. The type Ret must be such that an object of type OutputIt can be dereferenced and assigned a value of type Ret.

binary_op - binary operation function object that will be applied. The signature of the function should be equivalent to the following:

Ret fun(const Type1 &a, const Type2 &b);

The signature does not need to have const &. The types Type1 and Type2 must be such that objects of types InputIt1 and InputIt2 can be dereferenced and then implicitly converted to Type1 and Type2 respectively. The type Ret must be such that an object of type OutputIt can be dereferenced and assigned a value of type Ret.

Type requirements -InputIt must meet the requirements of InputIterator. -InputIt1 must meet the requirements of InputIterator. -InputIt2 must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

Output iterator to the element past the last element transformed.

5.4.20 std::generate

NAME

std::generate - Assigns each element in range [first, last) a value generated by the given function object g.

SYNOPSIS

#include <algorithm>

```
template< class ForwardIt, class Generator >
void generate( ForwardIt first, ForwardIt last, Generator g );
```

PARAMETERS

first, last - the range of elements to generate g - generator function object that will be called. The signature of the function should be equivalent to the following:

Ret fun(); The type Ret must be such that an object of type ForwardIt can be dereferenced and assigned a value of type Ret.

Type requirements -ForwardIt must meet the requirements of ForwardIterator.

RETURN VALUE

(none)

5.4.21 std::generate_n

NAME

std::generate_n - Assigns values, generated by given function object g, to the first count elements in the range beginning at first, if count>0. Does nothing otherwise.

SYNOPSIS

#include <algorithm>

```

 template< class OutputIt, class Size, class Generator >
void generate\_n( OutputIt first, Size count, Generator g ); [until C++11]
 template< class OutputIt, class Size, class Generator >
OutputIt generate\_n( OutputIt first, Size count, Generator g ); [since C++11]
```

PARAMETERS

first - the beginning of the range of elements to generate count - number of the elements to generate g - generator function object that will be called. The signature of the function should be equivalent to the following:

Ret fun(); The type Ret must be such that an object of type OutputIt can be dereferenced and assigned a value of type Ret.

Type requirements -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

(none) [until C++11] Iterator one past the last element assigned if count>0, first otherwise. [since C++11]

5.4.22 std::remove

NAME

std::remove - Deletes the file identified by character string pointed to by fname.

SYNOPSIS

#include <cstdio>

```
int remove( const char *fname );
```

PARAMETERS

fname - pointer to a null-terminated string containing the path identifying the file to delete

RETURN VALUE

0 upon success or non-zero value on error.

5.4.23 std::remove_copy

NAME

std::remove_copy - Copies elements from the range [first, last), to another range beginning at d_first, omitting the elements which satisfy specific criteria. The first version ignores the elements that are equal to value, the second version ignores the elements for which predicate p returns true. Source and destination ranges cannot overlap.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt, class OutputIt, class T >
OutputIt remove\_copy( InputIt first, InputIt last, OutputIt d\_first,
                      const T\& value );
 template< class InputIt, class OutputIt, class UnaryPredicate >
OutputIt remove\_copy\_if( InputIt first, InputIt last, OutputIt d\_first,
                         UnaryPredicate p );
```

PARAMETERS

first, last - the range of elements to copy d_first - the beginning of the destination range. value - the value of the elements not to copy Type requirements -InputIt must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator. -UnaryPredicate must meet the requirements of Predicate.

RETURN VALUE

Iterator to the element past the last element copied.

5.4.24 std::replace

NAME

std::replace - Replaces all elements satisfying specific criteria with new_value in the range [first, last). The first version replaces the elements that are equal to old_value, the second version replaces elements for which predicate p returns true.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt, class T >
void replace( ForwardIt first, ForwardIt last,
              const T\& old\_value, const T\& new\_value );
 template< class ForwardIt, class UnaryPredicate, class T >
void replace\_if( ForwardIt first, ForwardIt last,
                 UnaryPredicate p, const T\& new\_value );
```

PARAMETERS

first, last - the range of elements to process old_value - the value of elements to replace p - unary predicate which returns true if the element value should be replaced. The signature of the predicate function should be equivalent to the following:

bool pred(const Type &a);

The signature does not need to have const &, but the function must not modify the objects passed to it. The type Type must be such that an object of type ForwardIt can be dereferenced and then implicitly converted to Type.

new_value - the value to use as replacement Type requirements -ForwardIt must meet the requirements of ForwardIterator. -UnaryPredicate must meet the requirements of Predicate.

RETURN VALUE

(none)

5.4.25 std::replace_copy

NAME

std::replace_copy - Copies the all elements from the range [first, last) to another range beginning at d_first replacing all elements satisfying specific criteria with new_value. The first version replaces the elements that are equal to old_value, the second version replaces elements for which predicate p returns true. The source and destination ranges cannot overlap.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt, class OutputIt, class T >
OutputIt replace\_copy( InputIt first, InputIt last, OutputIt d\_first,
                        const T\& old\_value, const T\& new\_value );
 template< class InputIt, class OutputIt, class UnaryPredicate, class T >
OutputIt replace\_copy\_if( InputIt first, InputIt last, OutputIt d\_first,
                           UnaryPredicate p, const T\& new\_value );
```

PARAMETERS

first, last - the range of elements to copy d_first - the beginning of the destination range old_value - the value of elements to replace p - unary predicate which returns true if the element value should be replaced. The signature of the predicate function should be equivalent to the following:

bool pred(const Type &a);

The signature does not need to have const &, but the function must not modify the objects passed to it. The type Type must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type.

new_value - the value to use as replacement Type requirements -InputIt must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

Iterator to the element past the last element copied.

5.4.26 std::swap

NAME

std::swap - Exchanges the given values.

SYNOPSIS

#include <algorithm>

```
Defined in header <utility> [until C++11] [since C++11]
 template< class T >
void swap( T\& a, T\& b );
 template< class T2, size\_t N >
void swap( T2 (\&a)[N], T2 (\&b)[N]); [since C++11]
```

PARAMETERS

a, b - the values to be swapped Type requirements -T must meet the requirements of MoveAssignable and MoveConstructible. -T2 must meet the requirements of Swappable.

RETURN VALUE

(none)

5.4.27 std::swap_ranges

NAME

std::swap_ranges - Exchanges elements between range [first1, last1) and another range starting at first2.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt1, class ForwardIt2 >
ForwardIt2 swap\_ranges( ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2 )
```

PARAMETERS

first1, last1 - the first range of elements to swap first2 - beginning of the second range of elements to swap Type requirements -ForwardIt1, ForwardIt2 must meet the requirements of ForwardIterator. -The types of dereferenced ForwardIt1 and ForwardIt2 must meet the requirements of Swappable

RETURN VALUE

Iterator to the element past the last element exchanged in the range beginning with first2.

5.4.28 std::iter_swap

NAME

std::iter_swap - Swaps the values of the elements the given iterators are pointing to.

SYNOPSIS

#include <algorithm>

```
template< class ForwardIt1, class ForwardIt2 >
void iter\_swap( ForwardIt1 a, ForwardIt2 b );
```

PARAMETERS

a, b - iterators to the elements to swap Type requirements -ForwardIt1, ForwardIt2 must meet the requirements of ForwardIterator. -*a, *b must meet the requirements of Swappable.

RETURN VALUE

(none)

5.4.29 std::reverse

NAME

std::reverse - Reverses the order of the elements in the range [first, last)

SYNOPSIS

#include <algorithm>

```
template< class BidirIt >
void reverse( BidirIt first, BidirIt last );
```

PARAMETERS

first, last - the range of elements to reverse Type requirements -BidirIt must meet the requirements of BidirectionalIterator. -The type of dereferenced BidirIt must meet the requirements of Swappable.

RETURN VALUE

(none)

5.4.30 std::reverse_copy

NAME

std::reverse_copy - Copies the elements from the range [first, last) to another range beginning at d_first in such a way that the elements in the new range are in reverse order.

SYNOPSIS

#include <algorithm>

```
template< class BidirIt, class OutputIt >
OutputIt reverse\_copy( BidirIt first, BidirIt last, OutputIt d\_first );
```

PARAMETERS

first, last - the range of elements to copy d_first - the beginning of the destination range Type requirements -BidirIt must meet the requirements of BidirectionalIterator. -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

Output iterator to the element past the last element copied.

5.4.31 std::rotate

NAME

std::rotate - Performs a left rotation on a range of elements.

SYNOPSIS

#include <algorithm>

```

 template< class ForwardIt >
void rotate( ForwardIt first, ForwardIt n\_first, ForwardIt last ); [until C++11]
 template< class ForwardIt >
ForwardIt rotate( ForwardIt first, ForwardIt n\_first, ForwardIt last ); [since C++11]
```

PARAMETERS

first - the beginning of the original range n_first - the element that should appear at the beginning of the rotated range last - the end of the original range Type requirements -ForwardIt must meet the requirements of ValueSwappable and ForwardIterator. -The type of dereferenced ForwardIt must meet the requirements of MoveAssignable and MoveConstructible.

RETURN VALUE

(none) [until C++11] The iterator equal to first + (last - n_first) [since C++11]

5.4.32 std::rotate_copy

NAME

std::rotate_copy - Copies the elements from the range [first, last), to another range beginning at d_first in such a way, that the element n_first becomes the first element of the new range and n_first - 1 becomes the last element.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt, class OutputIt >
OutputIt rotate\_copy( ForwardIt first, ForwardIt n\_first,
                      ForwardIt last, OutputIt d\_first );
```

PARAMETERS

first, last - the range of elements to copy n_first - an iterator to an element in [first, last) that should appear at the beginning of the new range d_first - beginning of the destination range Type requirements -ForwardIt must meet the requirements of ForwardIterator. -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

Output iterator to the element past the last element copied.

5.4.33 std::random_shuffle

NAME

std::random_shuffle - Reorders the elements in the given range [first, last) such that each possible permutation of those elements has equal probability of appearance.

SYNOPSIS

#include <algorithm>

```
 template< class RandomIt >
void random\_shuffle( RandomIt first, RandomIt last ); [until C++17](deprecated in C++14)
(2)
 template< class RandomIt, class RandomFunc >
void random\_shuffle( RandomIt first, RandomIt last, RandomFunc\& r ); [until C++11]
 template< class RandomIt, class RandomFunc >
void random\_shuffle( RandomIt first, RandomIt last, RandomFunc\&\& r ); [since C++11]  [until C++17](
    deprecated in C++14)
 template< class RandomIt, class URNG >
void shuffle( RandomIt first, RandomIt last, URNG\&\& g ); [since C++11]
```

PARAMETERS

first, last - the range of elements to shuffle randomly r - function object returning a randomly chosen value of type convertible to std::iterator_traits<RandomIt>::difference_type in the interval [0,n) if invoked as r(n) g - a UniformRandomNumberGenerator whose result type is convertible to std::iterator_traits<RandomIt> Type requirements -RandomIt must meet the requirements of ValueSwappable and RandomAccessIterator. -URNG must meet the requirements of UniformRandomNumberGenerator.

RETURN VALUE

(none)

5.4.34 std::unique

NAME

std::unique - Removes all consecutive duplicate elements from the range [first, last) and returns a past-the-end iterator for the new logical end of the range. The first version uses operator== to compare the elements, the second version uses the given binary predicate p.

SYNOPSIS

#include <algorithm>

```
template< class ForwardIt >
ForwardIt unique( ForwardIt first, ForwardIt last );
template< class ForwardIt, class BinaryPredicate >
ForwardIt unique( ForwardIt first, ForwardIt last, BinaryPredicate p );
```

PARAMETERS

first, last - the range of elements to process p - binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

bool pred(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type ForwardIt can be dereferenced and then implicitly converted to both of them.

Type requirements -ForwardIt must meet the requirements of ForwardIterator. -The type of dereferenced ForwardIt must meet the requirements of MoveAssignable.

RETURN VALUE

Forward iterator to the new end of the range

5.4.35 std::unique_copy

NAME

std::unique_copy - Copies the elements from the range [first, last), to another range beginning at d_first in such a way that there are no consecutive equal elements. Only the first element of each group of equal elements is copied. The first version uses operator== to compare the elements, the second version uses the given binary predicate p.

SYNOPSIS

#include <algorithm>

```
template< class InputIt, class OutputIt >
OutputIt unique\_copy( InputIt first, InputIt last,
                       OutputIt d\_first );
template< class InputIt, class OutputIt, class BinaryPredicate >
OutputIt unique\_copy( InputIt first, InputIt last,
                       OutputIt d\_first, BinaryPredicate p );
```

PARAMETERS

first, last - the range of elements to process d_first - the beginning of the destination range p - binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

bool pred(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type ForwardIt can be dereferenced and then implicitly converted to both of them.

Type requirements -InputIt must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator. -The type of dereferenced InputIt must meet the requirements of CopyAssignable. -The type of dereferenced InputIt must meet the requirements of CopyConstructible. if neither InputIt nor OutputIt satisfies ForwardIterator

RETURN VALUE

Output iterator to the element past the last written element

5.4.36 std::partition

NAME

std::partition - Reorders the elements in the range [first, last) in such a way that all elements for which the predicate p returns true precede the elements for which predicate p returns false. Relative order of the elements is not preserved.

SYNOPSIS

#include <algorithm>

```

 template< class BidirIt, class UnaryPredicate >
BidirectionalIterator partition( BidirIt first, BidirIt last,

                                 UnaryPredicate p ); [until C++11]
 template< class ForwardIt, class UnaryPredicate >
ForwardIt partition( ForwardIt first, ForwardIt last,

                     UnaryPredicate p ); [since C++11]
```

PARAMETERS

first, last - the range of elements to reorder p - unary predicate which returns true if the element should be ordered before other elements. The signature of the predicate function should be equivalent to the following:

bool pred(const Type &a);

The signature does not need to have const &, but the function must not modify the objects passed to it. The type Type must be such that an object of type ForwardIt can be dereferenced and then implicitly converted to Type.

Type requirements -BidirIt must meet the requirements of BidirectionalIterator. -ForwardIt must meet the requirements of ValueSwappable and ForwardIterator. However, the operation is more efficient if ForwardIt also satisfies the requirements of BidirectionalIterator -UnaryPredicate must meet the requirements of Predicate.

RETURN VALUE

Iterator to the first element of the second group.

5.4.37 std::partition_copy

NAME

std::partition_copy - Copies the elements from the range [first, last) to two different ranges depending on the value returned by the predicate p. The elements, that satisfy the predicate p, are copied to the range beginning at d_first_true. The rest of the elements are copied to the range beginning at d_first_false.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt, class OutputIt1,
          class OutputIt2, class UnaryPredicate >
std::pair<OutputIt1, OutputIt2>
     partition\_copy( InputIt first, InputIt last,
                     OutputIt1 d\_first\_true, OutputIt2 d\_first\_false,

                     UnaryPredicate p ); [since C++11]
```

PARAMETERS

first, last - the range of elements to sort d_first_true - the beginning of the output range for the elements that satisfy p d_first_false - the beginning of the output range for the elements that do not satisfy p p - unary predicate which returns true if the element should be placed in d_first_true. The signature of the predicate function should be equivalent to the following:

bool pred(const Type &a);

The signature does not need to have const &, but the function must not modify the objects passed to it. The type Type must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type.

Type requirements -InputIt must meet the requirements of InputIterator. -The type of dereferenced InputIt must meet the requirements of CopyAssignable. -OutputIt1 must meet the requirements of OutputIterator. -OutputIt2 must meet the requirements of OutputIterator. -UnaryPredicate must meet the requirements of Predicate.

RETURN VALUE

A pair constructed from the iterator to the end of the d_first_true range and the iterator to the end of the d_first_false range.

5.4.38 std::stable_partition

NAME

std::stable_partition - Reorders the elements in the range [first, last) in such a way that all elements for which the predicate p returns true precede the elements for which predicate p returns false. Relative order of the elements is preserved.

SYNOPSIS

#include <algorithm>

```
 template< class BidirIt, class UnaryPredicate >
BidirIt stable\_partition( BidirIt first, BidirIt last, UnaryPredicate p );
```

PARAMETERS

first, last - the range of elements to reorder p - unary predicate which returns true if the element should be ordered before other elements. The signature of the predicate function should be equivalent to the following:

bool pred(const Type &a);

The signature does not need to have const &, but the function must not modify the objects passed to it. The type Type must be such that an object of type BidirIt can be dereferenced and then implicitly converted to Type.

Type requirements -BidirIt must meet the requirements of ValueSwappable and BidirectionalIterator. -The type of dereferenced BidirIt must meet the requirements of MoveAssignable and MoveConstructible. -UnaryPredicate must meet the requirements of Predicate.

RETURN VALUE

Iterator to the first element of the second group

5.4.39 std::is_sorted

NAME

std::is_sorted - Checks if the elements in range [first, last) are sorted in ascending order. The first version of the function uses operator< to compare the elements, the second uses the given comparison function comp.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt >
bool is\_sorted( ForwardIt first, ForwardIt last ); [since C++11]
 template< class ForwardIt, class Compare >
bool is\_sorted( ForwardIt first, ForwardIt last, Compare comp ); [since C++11]
```

PARAMETERS

first, last - the range of elements to examine comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type ForwardIt can be dereferenced and then implicitly converted to both of them.

Type requirements -ForwardIt must meet the requirements of ForwardIterator.

RETURN VALUE

true if the elements in the range are sorted in ascending order

5.4.40 std::sort

NAME

std::sort - Sorts the elements in the range [first, last) in ascending order. The order of equal elements is not guaranteed to be preserved. The first version uses operator< to compare the elements, the second version uses the given comparison function object comp.

SYNOPSIS

#include <algorithm>

```
 template< class RandomIt >
void sort( RandomIt first, RandomIt last );
 template< class RandomIt, class Compare >
void sort( RandomIt first, RandomIt last, Compare comp );
```

PARAMETERS

first, last - the range of elements to sort comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type RandomIt can be dereferenced and then implicitly converted to both of them.

Type requirements -RandomIt must meet the requirements of ValueSwappable and RandomAccessIterator. -The type of dereferenced RandomIt must meet the requirements of MoveAssignable and MoveConstructible. -Compare must meet the requirements of Compare.

RETURN VALUE

(none)

5.4.41 std::partial_sort

NAME

std::partial_sort - Rearranges elements such that the range [first, middle) contains the sorted middle - first smallest elements in the range [first, last).

SYNOPSIS

#include <algorithm>

```
 template< class RandomIt >
void partial\_sort( RandomIt first, RandomIt middle, RandomIt last );
 template< class RandomIt, class Compare >
void partial\_sort( RandomIt first, RandomIt middle, RandomIt last, Compare comp );
```

PARAMETERS

first, last - the range of elements to sort comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type RandomIt can be dereferenced and then implicitly converted to both of them.

Type requirements -RandomIt must meet the requirements of ValueSwappable and RandomAccessIterator. -The type of dereferenced RandomIt must meet the requirements of MoveAssignable and MoveConstructible.

RETURN VALUE

(none)

5.4.42 std::partial_sort_copy

NAME

std::partial_sort_copy - Sorts some of the elements in the range [first, last) in ascending order, storing the result in the range [d_first, d_last).

SYNOPSIS

#include <algorithm>

```
 template< class InputIt, class RandomIt >
RandomIt partial\_sort\_copy( InputIt first, InputIt last,
                            RandomIt d\_first, RandomIt d\_last );
 template< class InputIt, class RandomIt, class Compare >
RandomIt partial\_sort\_copy( InputIt first, InputIt last,
                            RandomIt d\_first, RandomIt d\_last,
                            Compare comp );
```

PARAMETERS

first, last - the range of elements to sort d_first, d_last - random access iterators defining the destination range comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type RandomIt can be dereferenced and then implicitly converted to both of them.

Type requirements -InputIt must meet the requirements of InputIterator. -RandomIt must meet the requirements of ValueSwappable and RandomAccessIterator. -The type of dereferenced RandomIt must meet the requirements of MoveAssignable and MoveConstructible.

RETURN VALUE

an iterator to the element defining the upper boundary of the sorted range, i.e. d_first + min(last - first, d_last - d_first).

5.4.43 std::stable_sort

NAME

std::stable_sort - Sorts the elements in the range [first, last) in ascending order. The order of equal elements is guaranteed to be preserved. The first version uses operator< to compare the elements, the second version uses the given comparison function comp.

SYNOPSIS

#include <algorithm>

```
 template< class RandomIt >
void stable\_sort( RandomIt first, RandomIt last );
 template< class RandomIt, class Compare >
void stable\_sort( RandomIt first, RandomIt last, Compare comp );
```

PARAMETERS

first, last - the range of elements to sort comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type RandomIt can be dereferenced and then implicitly converted to both of them.

Type requirements -RandomIt must meet the requirements of ValueSwappable and RandomAccessIterator. -The type of dereferenced RandomIt must meet the requirements of MoveAssignable and MoveConstructible.

RETURN VALUE

(none)

5.4.44 std::nth_element

NAME

std::nth_element - nth_element is a partial sorting algorithm that rearranges elements in [first, last) such that:

SYNOPSIS

#include <algorithm>

```
 template< class RandomIt >
void nth\_element( RandomIt first, RandomIt nth, RandomIt last );
 template< class RandomIt, class Compare >
void nth\_element( RandomIt first, RandomIt nth, RandomIt last, Compare comp );
```

PARAMETERS

first, last - random access iterators defining the range sort nth - random access iterator defining the sort partition point comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type RandomIt can be dereferenced and then implicitly converted to both of them.

Type requirements -RandomIt must meet the requirements of ValueSwappable and RandomAccessIterator. -The type of dereferenced RandomIt must meet the requirements of MoveAssignable and MoveConstructible.

RETURN VALUE

(none)

5.4.45 std::lower_bound

NAME

std::lower_bound - Returns an iterator pointing to the first element in the range [first, last) that is not less than (i.e. greater or equal to) value.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt, class T >
ForwardIt lower\_bound( ForwardIt first, ForwardIt last, const T\& value );
 template< class ForwardIt, class T, class Compare >
ForwardIt lower\_bound( ForwardIt first, ForwardIt last, const T\& value, Compare comp );
```

PARAMETERS

first, last - iterators defining the partially-ordered range to examine value - value to compare the elements to comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The type Type1 must be such that an object of type ForwardIt can be dereferenced and then implicitly converted to Type1. The type Type2 must be such that an object of type T can be implicitly converted to Type2.

Type requirements -ForwardIt must meet the requirements of ForwardIterator.

RETURN VALUE

Iterator pointing to the first element that is not less than value, or last if no such element is found.

5.4.46 std::upper_bound

NAME

std::upper_bound - Returns an iterator pointing to the first element in the range [first, last) that is greater than value.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt, class T >
ForwardIt upper\_bound( ForwardIt first, ForwardIt last, const T\& value );
 template< class ForwardIt, class T, class Compare >
ForwardIt upper\_bound( ForwardIt first, ForwardIt last, const T\& value, Compare comp );
```

PARAMETERS

first, last - the range of elements to examine value - value to compare the elements to comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The type Type1 must be such that an object of type T can be implicitly converted to Type1. The type Type2 must be such that an object of type ForwardIt can be dereferenced and then implicitly converted to Type2.

Type requirements -ForwardIt must meet the requirements of ForwardIterator.

RETURN VALUE

iterator pointing to the first element that is greater than value, or last if no such element is found.

5.4.47 std::binary_search

NAME

std::binary_search - Checks if an element equivalent to value appears within the range [first, last).

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt, class T >
bool binary\_search( ForwardIt first, ForwardIt last, const T\& value );
 template< class ForwardIt, class T, class Compare >
bool binary\_search( ForwardIt first, ForwardIt last, const T\& value, Compare comp );
```

PARAMETERS

first, last - the range of elements to examine value - value to compare the elements to comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The type Type1 must be such that an object of type T can be implicitly converted to Type1. The type Type2 must be such that an object of type ForwardIt can be dereferenced and then implicitly converted to Type2.

Type requirements -ForwardIt must meet the requirements of ForwardIterator.

RETURN VALUE

true if an element equal to value is found, false otherwise.

5.4.48 std::equal_range

NAME

std::equal_range - Returns a range containing all elements equivalent to value in the range [first, last).

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt, class T >
std::pair<ForwardIt,ForwardIt>
    equal\_range( ForwardIt first, ForwardIt last,
                 const T\& value );
 template< class ForwardIt, class T, class Compare >
std::pair<ForwardIt,ForwardIt>
    equal\_range( ForwardIt first, ForwardIt last,
                 const T\& value, Compare comp );
```

PARAMETERS

first, last - the range of elements to examine value - value to compare the elements to comp - comparison function which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b); The signature does not need to have const &, but the function must not modify the objects passed to it. cmp will be called as both cmp(value, *iterator) and cmp(*iterator, value).

Type requirements -ForwardIt must meet the requirements of ForwardIterator.

RETURN VALUE

std::pair containing a pair of iterators defining the wanted range, the first pointing to the first element that is not less than value and the second pointing to the first element greater than value.

If there are no elements not less than value, last is returned as the first element. Similarly if there are no elements greater than value, last is returned as the second element

5.4.49 std::merge

NAME

std::merge - Merges two sorted ranges [first1, last1) and [first2, last2) into one sorted range beginning at d_first. The first version uses operator< to compare the elements, the second version uses the given comparison function comp. The relative order of equivalent elements is preserved.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt1, class InputIt2, class OutputIt >
OutputIt merge( InputIt1 first1, InputIt1 last1,
                InputIt2 first2, InputIt2 last2,
                OutputIt d\_first );
 template< class InputIt1, class InputIt2, class OutputIt, class Compare>
OutputIt merge( InputIt1 first1, InputIt1 last1,
                InputIt2 first2, InputIt2 last2,
                OutputIt d\_first, Compare comp );
```

PARAMETERS

first1, last1 - the first range of elements to merge first2, last2 - the second range of elements to merge d_first - the beginning of the destination range comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that objects of types InputIt1 and InputIt2 can be dereferenced and then implicitly converted to Type1 and Type2 respectively.

Type requirements -InputIt1 must meet the requirements of InputIterator. -InputIt2 must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

An output iterator to element past the last element copied.

5.4.50 std::inplace_merge

NAME

std::inplace_merge - Merges two consecutive sorted ranges [first, middle) and [middle, last) into one sorted range [first, last). The order of equal elements is guaranteed to be preserved. The first version uses operator< to compare the elements, the second version uses the given comparison function comp.

SYNOPSIS

#include <algorithm>

```
 template< class BidirIt >
void inplace\_merge( BidirIt first, BidirIt middle, BidirIt last );
 template< class BidirIt, class Compare>
void inplace\_merge( BidirIt first, BidirIt middle, BidirIt last, Compare comp );
```

PARAMETERS

first - the beginning of the first sorted range middle - the end of the first sorted range and the beginning of the second last - the end of the second sorted range comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e.

is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type BidirIt can be dereferenced and then implicitly converted to both of them.

Type requirements -BidirIt must meet the requirements of ValueSwappable and BidirectionalIterator. -The type of dereferenced BidirIt must meet the requirements of MoveAssignable and MoveConstructible.

RETURN VALUE

(none)

5.4.51 std::includes

NAME

std::includes - Returns true if every element from the sorted range [first2, last2) is found within the sorted range [first1, last1). Also returns true if [first2, last2) is empty.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt1, class InputIt2 >
bool includes( InputIt1 first1, InputIt1 last1,
               InputIt2 first2, InputIt2 last2 );
 template< class InputIt1, class InputIt2 >
bool includes( InputIt1 first1, InputIt1 last1,
               InputIt2 first2, InputIt2 last2, Compare comp );
```

PARAMETERS

first1, last1 - the sorted range of elements to examine first2, last2 - the sorted range of elements to search for comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type InputIt can be dereferenced and then implicitly converted to both of them.

Type requirements -InputIt must meet the requirements of InputIterator.

RETURN VALUE

true if every element from [first2, last2) is a member of [first, last).

5.4.52 std::set_difference

NAME

std::set_difference - Copies the elements from the sorted range [first1, last1) which are not found in the sorted range [first2, last2) to the range beginning at d_first.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt1, class InputIt2, class OutputIt >
OutputIt set\_difference( InputIt1 first1, InputIt1 last1,
                         InputIt2 first2, InputIt2 last2,
                         OutputIt d\_first );
 template< class InputIt1, class InputIt2,
          class OutputIt, class Compare >
OutputIt set\_difference( InputIt1 first1, InputIt1 last1,
                         InputIt2 first2, InputIt2 last2,
                         OutputIt d\_first, Compare comp );
```

PARAMETERS

first1, last1 - the range of elements to examine first2, last2 - the range of elements to search for comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that objects of types InputIt1 and InputIt2 can be dereferenced and then implicitly converted to Type1 and Type2 respectively.

Type requirements -InputIt1 must meet the requirements of InputIterator. -InputIt2 must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

Iterator past the end of the constructed range.

5.4.53 std::set_intersection

NAME

std::set_intersection - Constructs a sorted range beginning at d_first consisting of elements that are found in both sorted ranges [first1, last1) and [first2, last2). The first version expects both input ranges to be sorted with operator<, the second version expects them to be sorted with the given comparison function comp. If some element is found m times in [first1, last1) and n times in [first2, last2), the first std::min(m, n) elements will be copied from the first range to the destination range. The order of equivalent elements is preserved. The resulting range cannot overlap with either of the input ranges.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt1, class InputIt2, class OutputIt >
```

```
OutputIt set\_intersection( InputIt1 first1, InputIt1 last1,
                           InputIt2 first2, InputIt2 last2,
                           OutputIt d\_first );
 template< class InputIt1, class InputIt2,
          class OutputIt, class Compare >
OutputIt set\_intersection( InputIt1 first1, InputIt1 last1,
                           InputIt2 first2, InputIt2 last2,
                           OutputIt d\_first, Compare comp );
```

PARAMETERS

first1, last1 - the first range of elements to examine first2, last2 - the second range of elements to examine comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that objects of types InputIt1 and InputIt2 can be dereferenced and then implicitly converted to Type1 and Type2 respectively.

Type requirements -InputIt1 must meet the requirements of InputIterator. -InputIt2 must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

Iterator past the end of the constructed range.

5.4.54 std::set_symmetric_difference

NAME

std::set_symmetric_difference - Computes symmetric difference of two sorted ranges: the elements that are found in either of the ranges, but not in both of them are copied to the range beginning at d_first. The resulting range is also sorted.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt1, class InputIt2, class OutputIt >
OutputIt set\_symmetric\_difference( InputIt1 first1, InputIt1 last1,
                                   InputIt2 first2, InputIt2 last2,
                                   OutputIt d\_first );
 template< class InputIt1, class InputIt2,
          class OutputIt, class Compare >
OutputIt set\_symmetric\_difference( InputIt1 first1, InputIt1 last1,
                                   InputIt2 first2, InputIt2 last2,
                                   OutputIt d\_first, Compare comp );
```

PARAMETERS

first1, last1 - the first sorted range of elements first2, last2 - the second sorted range of elements comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that objects of types InputIt1 and InputIt2 can be dereferenced and then implicitly converted to Type1 and Type2 respectively.

Type requirements -InputIt1 must meet the requirements of InputIterator. -InputIt2 must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

Iterator past the end of the constructed range.

5.4.55 std::set_union

NAME

std::set_union - Constructs a sorted range beginning at d_first consisting of all elements present in one or both sorted ranges [first1, last1) and [first2, last2).

SYNOPSIS

#include <algorithm>

```
 template< class InputIt1, class InputIt2, class OutputIt >
OutputIt set\_union( InputIt1 first1, InputIt1 last1,
                    InputIt2 first2, InputIt2 last2,
                    OutputIt d\_first );
 template< class InputIt1, class InputIt2,
          class OutputIt, class Compare >
OutputIt set\_union( InputIt1 first1, InputIt1 last1,
                    InputIt2 first2, InputIt2 last2,
                    OutputIt d\_first, Compare comp );
```

PARAMETERS

first1, last1 - the first input sorted range first2, last2 - the second input sorted range comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than (i.e. is ordered before) the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that objects of types InputIt1 and InputIt2 can be dereferenced and then implicitly converted to Type1 and Type2 respectively.

Type requirements -InputIt1 must meet the requirements of InputIterator. -InputIt2 must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

Iterator past the end of the constructed range.

5.4.56 std::is_heap

NAME

std::is_heap - Checks if the elements in range [first, last) are a max heap.

SYNOPSIS

#include <algorithm>

```
 template< class RandomIt >
bool is\_heap( RandomIt first, RandomIt last ); [since C++11]
 template< class RandomIt, class Compare >
bool is\_heap( RandomIt first, RandomIt last, Compare comp ); [since C++11]
```

PARAMETERS

first, last - the range of elements to examine comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type RandomIt can be dereferenced and then implicitly converted to both of them.

Type requirements -RandomIt must meet the requirements of RandomAccessIterator.

RETURN VALUE

true if the range is max heap, false otherwise.

5.4.57 std::make_heap

NAME

std::make_heap - Constructs a max heap in the range [first, last). The first version of the function uses operator< to compare the elements, the second uses the given comparison function comp.

SYNOPSIS

#include <algorithm>

```
 template< class RandomIt >
void make\_heap( RandomIt first, RandomIt last );
 template< class RandomIt, class Compare >
void make\_heap( RandomIt first, RandomIt last,
                Compare comp );
```

PARAMETERS

first, last - the range of elements to make the heap from comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type RandomIt can be dereferenced and then implicitly converted to both of them.

Type requirements -RandomIt must meet the requirements of RandomAccessIterator. -The type of dereferenced RandomIt must meet the requirements of MoveAssignable and MoveConstructible.

RETURN VALUE

(none)

5.4.58 std::push_heap

NAME

std::push_heap - Inserts the element at the position last-1 into the max heap defined by the range [first, last-1). The first version of the function uses operator< to compare the elements, the second uses the given comparison function comp.

SYNOPSIS

#include <algorithm>

```
 template< class RandomIt >
void push\_heap( RandomIt first, RandomIt last );
 template< class RandomIt, class Compare >
void push\_heap( RandomIt first, RandomIt last,
                Compare comp );
```

PARAMETERS

first, last - the range of elements defining the heap to modify comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type RandomIt can be dereferenced and then implicitly converted to both of them.

Type requirements -RandomIt must meet the requirements of RandomAccessIterator. -The type of dereferenced RandomIt must meet the requirements of MoveAssignable and MoveConstructible.

RETURN VALUE

(none)

5.4.59 std::pop_heap

NAME

std::pop_heap - Swaps the value in the position first and the value in the position last-1 and makes the subrange [first, last-1) into a max heap. This has the effect of removing the first (largest) element from the heap defined by the range [first, last).

SYNOPSIS

#include <algorithm>

```
 template< class RandomIt >
void pop\_heap( RandomIt first, RandomIt last );
 template< class RandomIt, class Compare >
void pop\_heap( RandomIt first, RandomIt last, Compare comp );
```

PARAMETERS

first, last - the range of elements defining the valid nonempty heap to modify comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type RandomIt can be dereferenced and then implicitly converted to both of them.

Type requirements -RandomIt must meet the requirements of ValueSwappable and RandomAccessIterator. -The type of dereferenced RandomIt must meet the requirements of MoveAssignable and MoveConstructible.

RETURN VALUE

(none)

5.4.60 std::sort_heap

NAME

std::sort_heap - Converts the max heap [first, last) into a sorted range in ascending order. The resulting range no longer has the heap property.

SYNOPSIS

#include <algorithm>

```
 template< class RandomIt >
void sort\_heap( RandomIt first, RandomIt last );
 template< class RandomIt, class Compare >
void sort\_heap( RandomIt first, RandomIt last, Compare comp );
```

PARAMETERS

first, last - the range of elements to sort comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type RandomIt can be dereferenced and then implicitly converted to both of them.

Type requirements -RandomIt must meet the requirements of ValueSwappable and RandomAccessIterator. -The type of dereferenced RandomIt must meet the requirements of MoveAssignable and MoveConstructible.

RETURN VALUE

(none)

5.4.61 std::max

NAME

std::max - Returns the greater of the given values.

SYNOPSIS

#include <algorithm>

```

(1)
 template< class T >
const T\& max( const T\& a, const T\& b ); [until C++14]
 template< class T >
constexpr const T\& max( const T\& a, const T\& b ); [since C++14]
(2)
 template< class T, class Compare >
const T\& max( const T\& a, const T\& b, Compare comp ); [until C++14]
 template< class T, class Compare >
constexpr const T\& max( const T\& a, const T\& b, Compare comp ); [since C++14]
(3)
 template< class T >
T max( std::initializer\_list<T> ilist ); [since C++11]  [until C++14]
 template< class T >
constexpr T max( std::initializer\_list<T> ilist ); [since C++14]
(4)
 template< class T, class Compare >
T max( std::initializer\_list<T> ilist, Compare comp ); [since C++11]  [until C++14]
 template< class T, class Compare >
constexpr T max( std::initializer\_list<T> ilist, Compare comp ); [since C++14]
```

PARAMETERS

a, b - the values to compare ilist - initializer list with the values to compare comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if if a is less than b. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type T can be implicitly converted to both of them.

Type requirements -T must meet the requirements of LessThanComparable. for the overloads (1) and (3) -T must meet the requirements of CopyConstructible. for the overloads (3) and (4)

RETURN VALUE

1-2) The greater of a and b. If they are equivalent, returns a.

3-4) The greatest value in ilist. If several values are equivalent to the greatest, returns the leftmost one.

5.4.62 std::max_element

NAME

std::max_element - Finds the greatest element in the range [first, last). The first version uses operator< to compare the values, the second version uses the given comparison function cmp.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt >
ForwardIt max\_element(ForwardIt first, ForwardIt last);
 template< class ForwardIt, class Compare >
ForwardIt max\_element(ForwardIt first, ForwardIt last, Compare cmp);
```

PARAMETERS

first, last - forward iterators defining the range to examine cmp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type ForwardIt can be dereferenced and then implicitly converted to both of them.

Type requirements -ForwardIt must meet the requirements of ForwardIterator.

RETURN VALUE

Iterator to the greatest element in the range [first, last). If several elements in the range are equivalent to the greatest element, returns the iterator to the first such element. Returns last if the range is empty.

5.4.63 std::min

NAME

std::min - Returns the smaller of the given values.

SYNOPSIS

#include <algorithm>

```

(1)
 template< class T >
const T\& min( const T\& a, const T\& b ); [until C++14]
 template< class T >
constexpr const T\& min( const T\& a, const T\& b ); [since C++14]
(2)
 template< class T, class Compare >
const T\& min( const T\& a, const T\& b, Compare comp ); [until C++14]
 template< class T, class Compare >
constexpr const T\& min( const T\& a, const T\& b, Compare comp ); [since C++14]
(3)
 template< class T >
T min( std::initializer\_list<T> ilist ); [since C++11]  [until C++14]
 template< class T >
constexpr T min( std::initializer\_list<T> ilist ); [since C++14]
(4)
 template< class T, class Compare >
T min( std::initializer\_list<T> ilist, Compare comp ); [since C++11]  [until C++14]
 template< class T, class Compare >
constexpr T min( std::initializer\_list<T> ilist, Compare comp ); [since C++14]
```

PARAMETERS

a, b - the values to compare ilist - initializer list with the values to compare cmp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if if a is less than b. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type T can be implicitly converted to both of them.

Type requirements -T must meet the requirements of LessThanComparable. for the overloads (1) and (3) -T must meet the requirements of CopyConstructible. for the overloads (3) and (4)

RETURN VALUE

1-2) The smaller of a and b. If the values are equivalent, returns a.

3-4) The smallest value in ilist. If several values are equivalent to the smallest, returns the leftmost such value.

5.4.64 std::min_element

NAME

std::min_element - Finds the smallest element in the range [first, last). The first version uses operator< to compare the values, the second version uses the given comparison function comp.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt >
ForwardIt min\_element( ForwardIt first, ForwardIt last );
 template< class ForwardIt, class Compare >
ForwardIt min\_element( ForwardIt first, ForwardIt last, Compare comp );
```

PARAMETERS

first, last - forward iterators defining the range to examine cmp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if a is less than b. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type ForwardIt can be dereferenced and then implicitly converted to both of them.

Type requirements -ForwardIt must meet the requirements of ForwardIterator.

RETURN VALUE

Iterator to the smallest element in the range [first, last). If several elements in the range are equivalent to the smallest element, returns the iterator to the first such element. Returns last if the range is empty.

5.4.65 std::minmax

NAME

std::minmax - Returns the lowest and the greatest of the given values.

SYNOPSIS

#include <algorithm>

```
(1)
 template< class T >
std::pair<const T\&,const T\&> minmax( const T\& a, const T\& b ); [since C++11]  [until C++14]
 template< class T >
constexpr std::pair<const T\&,const T\&> minmax( const T\& a, const T\& b ); [since C++14]
(2)
 template< class T, class Compare >
std::pair<const T\&,const T\&> minmax( const T\& a, const T\& b,
                                     Compare comp ); [since C++11]  [until C++14]
 template< class T, class Compare >
constexpr std::pair<const T\&,const T\&> minmax( const T\& a, const T\& b,
                                               Compare comp ); [since C++14]
(3)
 template< class T >
std::pair<T,T> minmax( std::initializer\_list<T> ilist); [since C++11]  [until C++14]
 template< class T >
constexpr std::pair<T,T> minmax( std::initializer\_list<T> ilist); [since C++14]
(4)
 template< class T, class Compare >
std::pair<T,T> minmax( std::initializer\_list<T> ilist, Compare comp ); [since C++11]  [until C++14]
 template< class T, class Compare >
constexpr std::pair<T,T> minmax( std::initializer\_list<T> ilist, Compare comp ); [since C++14]
```

PARAMETERS

a, b - the values to compare ilist - initializer list with the values to compare comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type T can be implicitly converted to both of them.

Type requirements -T must meet the requirements of LessThanComparable. -T must meet the requirements of CopyConstructible in order to use overloads (3,4).

RETURN VALUE

1-2) Returns the result of std::pair<const T&, const T&>(a, b) if a<b or if a is equivalent to b. Returns the result of std::pair<const T&, const T&>(b, a) if b<a.

3-4) A pair with the smallest value in ilist as the first element and the greatest as the second. If several elements are equivalent to the smallest, the leftmost such element is returned. If several elements are equivalent to the largest, the rightmost such element is returned.

5.4.66 std::minmax_element

NAME

std::minmax_element - Finds the greatest and the smallest element in the range [first, last). The first version uses operator< to compare the values, the second version uses the given comparison function comp.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt >
std::pair<ForwardIt,ForwardIt>

    minmax\_element( ForwardIt first, ForwardIt last ); [since C++11]
 template< class ForwardIt, class Compare >
std::pair<ForwardIt,ForwardIt>

    minmax\_element( ForwardIt first, ForwardIt last, Compare comp ); [since C++11]
```

PARAMETERS

first, last - forward iterators defining the range to examine cmp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if if *a is less than *b. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type ForwardIt can be dereferenced and then implicitly converted to both of them.

Type requirements -ForwardIt must meet the requirements of ForwardIterator.

RETURN VALUE

a pair consisting of an iterator to the smallest element as the first element and an iterator to the greatest element as the second. Returns std::make_pair(first, first) if the range is empty. If several elements are equivalent to the smallest element, the iterator to the first such element is returned. If several elements are equivalent to the largest element, the iterator to the last such element is returned.

5.4.67 std::lexicographical_compare

NAME

std::lexicographical_compare - Checks if the first range [first1, last1) is lexicographically less than the second range [first2, last2). The first version uses operator< to compare the elements, the second version uses the given comparison function comp.

SYNOPSIS

#include <algorithm>

```
 template< class InputIt1, class InputIt2 >
bool lexicographical\_compare( InputIt1 first1, InputIt1 last1,
                              InputIt2 first2, InputIt2 last2 );
```

```
 template< class InputIt1, class InputIt2, class Compare >
bool lexicographical\_compare( InputIt1 first1, InputIt1 last1,
                              InputIt2 first2, InputIt2 last2,
                              Compare comp );
```

PARAMETERS

first1, last1 - the first range of elements to examine first2, last2 - the second range of elements to examine comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that objects of types InputIt1 and InputIt2 can be dereferenced and then implicitly converted to Type1 and Type2 respectively.

Type requirements -InputIt1, InputIt2 must meet the requirements of InputIterator.

RETURN VALUE

true if the first range is lexicographically less than the second.

5.4.68 std::is_permutation

NAME

std::is_permutation - Returns true if there exists a permutation of the elements in the range [first1, last1) that makes that range equal to the range [first2,last2), where last2 denotes first2 + (last1 - first1) if it was not given.

SYNOPSIS

#include <algorithm>

```
 template< class ForwardIt1, class ForwardIt2 >
bool is\_permutation( ForwardIt1 first1, ForwardIt1 last1,

                     ForwardIt2 first2 ); [since C++11]
 template< class ForwardIt1, class ForwardIt2, class BinaryPredicate >
bool is\_permutation( ForwardIt1 first1, ForwardIt1 last1,

                     ForwardIt2 first2, BinaryPredicate p ); [since C++11]
 template< class ForwardIt1, class ForwardIt2 >
bool is\_permutation( ForwardIt1 first1, ForwardIt1 last1,

                     ForwardIt2 first2, ForwardIt2 last2 ); [since C++14]
 template< class ForwardIt1, class ForwardIt2, class BinaryPredicate >
bool is\_permutation( ForwardIt1 first1, ForwardIt1 last1,
                     ForwardIt2 first2, ForwardIt2 last2,

                     BinaryPredicate p ); [since C++14]
```

PARAMETERS

first1, last1 - the range of elements to compare first2, last2 - the second range to compare p - binary predicate which returns true if the elements should be treated as equal. The signature of the predicate function should be equivalent to the following:

bool pred(const Type &a, const Type &b);

Type should be the value type of both ForwardIt1 and ForwardIt2. The signature does not need to have const &, but the function must not modify the objects passed to it.

Type requirements -ForwardIt1, ForwardIt2 must meet the requirements of ForwardIterator. -ForwardIt1, ForwardIt2 must have the same value type.

RETURN VALUE

true if the range [first1, last1) is a permutation of the range [first2, last2).

5.4.69 std::next_permutation

NAME

std::next_permutation - Transforms the range [first, last) into the next permutation from the set of all permutations that are lexicographically ordered with respect to operator< or comp. Returns true if such permutation exists, otherwise transforms the range into the first permutation (as if by std::sort(first, last)) and returns false.

SYNOPSIS

#include <algorithm>

```
 template< class BidirIt >
bool next\_permutation( BidirIt first, BidirIt last );
 template< class BidirIt, class Compare >
bool next\_permutation( BidirIt first, BidirIt last, Compare comp );
```

PARAMETERS

first, last - the range of elements to permute comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type BidirIt can be dereferenced and then implicitly converted to both of them.

Type requirements -BidirIt must meet the requirements of ValueSwappable and BidirectionalIterator.

RETURN VALUE

true if the new permutation is lexicographically greater than the old. false if the last permutation was reached and the range was reset to the first permutation.

5.4.70 std::prev_permutation

NAME

std::prev_permutation - Transforms the range [first, last) into the previous permutation from the set of all permutations that are lexicographically ordered with respect to operator< or comp. Returns true if such permutation exists, otherwise transforms the range into the last permutation (as if by std::sort(first, last); std::reverse(first, last);) and returns false.

SYNOPSIS

#include <algorithm>

```
 template< class BidirIt >
bool prev\_permutation( BidirIt first, BidirIt last);
 template< class BidirIt, class Compare >
bool prev\_permutation( BidirIt first, BidirIt last, Compare comp);
```

PARAMETERS

first, last - the range of elements to permute comp - comparison function object (i.e. an object that satisfies the requirements of Compare) which returns true if the first argument is less than the second. The signature of the comparison function should be equivalent to the following:

bool cmp(const Type1 &a, const Type2 &b);

The signature does not need to have const &, but the function object must not modify the objects passed to it. The types Type1 and Type2 must be such that an object of type BidirIt can be dereferenced and then implicitly converted to both of them.

Type requirements -BidirIt must meet the requirements of ValueSwappable and BidirectionalIterator.

RETURN VALUE

true if the new permutation precedes the old in lexicographical order. false if the first permutation was reached and the range was reset to the last permutation.

5.5. Numerics library

5.5.1 std::iota

NAME

std::iota - Fills the range [first, last) with sequentially increasing values, starting with value and repetitively evaluating ++value.

SYNOPSIS

#include <numeric>

```
 template< class ForwardIterator, class T >
void iota( ForwardIterator first, ForwardIterator last, T value ); [since C++11]
```

PARAMETERS

first, last - the range of elements to fill with sequentially increasing values starting with value value - initial value to store, the expression ++value must be well-formed

RETURN VALUE

(none)

5.5.2 std::accumulate

NAME

std::accumulate - Computes the sum of the given value init and the elements in the range [first, last). The first version uses operator+ to sum up the elements, the second version uses the given binary function op.

SYNOPSIS

#include <numeric>

```
 template< class InputIt, class T >
T accumulate( InputIt first, InputIt last, T init );
 template< class InputIt, class T, class BinaryOperation >
T accumulate( InputIt first, InputIt last, T init,
              BinaryOperation op );
```

PARAMETERS

first, last - the range of elements to sum init - initial value of the sum op - binary operation function object that will be applied. The signature of the function should be equivalent to the following:

Ret fun(const Type1 &a, const Type2 &b);

The signature does not need to have const &. The type Type1 must be such that an object of type T can be implicitly converted to Type1. The type Type2 must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type2. The type Ret must be such that an object of type T can be assigned a value of type Ret.

Type requirements -InputIt must meet the requirements of InputIterator. -T must meet the requirements of CopyAssignable and CopyConstructible.

RETURN VALUE

The sum of the given value and elements in the given range.

5.5.3 std::inner_product

NAME

std::inner_product - Computes inner product (i.e. sum of products) of the range [first1, last1) and another range beginning at first2. The first version uses operator* to compute product of the element pairs and operator+ to sum up the products, the second version uses op2 and op1 for these tasks respectively.

SYNOPSIS

#include <numeric>

```
 template< class InputIt1, class InputIt2, class T >
T inner\_product( InputIt1 first1, InputIt1 last1,
                 InputIt2 first2, T value );
 template<
    class InputIt1,
    class InputIt2,
    class T,
    class BinaryOperation1,
```

```
    class BinaryOperation2
> T inner\_product( InputIt1 first1, InputIt1 last1,
                   InputIt2 first2, T value,
                   BinaryOperation1 op1,
                   BinaryOperation2 op2 );
```

PARAMETERS

first1, last1 - the first range of elements first2 - the beginning of the second range of elements value - initial value of the sum of the products op1 - binary operation function object that will be applied. This "sum" function takes a value returned by op2 and the current value of the accumulator and produces a new value to be stored in the accumulator. The signature of the function should be equivalent to the following:

Ret fun(const Type1 &a, const Type2 &b);

The signature does not need to have const &. The types Type1 and Type2 must be such that objects of types T and Type3 can be implicitly converted to Type1 and Type2 respectively. The type Ret must be such that an object of type T can be assigned a value of type Ret.

op2 - binary operation function object that will be applied. This "product" function takes one value from each range and produces a new value. The signature of the function should be equivalent to the following:

Ret fun(const Type1 &a, const Type2 &b);

The signature does not need to have const &. The types Type1 and Type2 must be such that objects of types InputIt1 and InputIt2 can be dereferenced and then implicitly converted to Type1 and Type2 respectively. The type Ret must be such that an object of type Type3 can be assigned a value of type Ret.

Type requirements -InputIt1, InputIt2 must meet the requirements of InputIterator. -T must meet the requirements of CopyAssignable and CopyConstructible.

RETURN VALUE

The inner product of two ranges.

5.5.4 std::adjacent_difference

NAME

std::adjacent_difference - Computes the differences between the second and the first of each adjacent pair of elements of the range [first, last) and writes them to the range beginning at d_first + 1. Unmodified copy of first is written to d_first. The first version uses operator- to calculate the differences, the second version uses the given binary function op.

SYNOPSIS

#include <numeric>

```
 template< class InputIt, class OutputIt >
OutputIt adjacent\_difference( InputIt first, InputIt last,
                                   OutputIt d\_first );
 template< class InputIt, class OutputIt, class BinaryOperation >
OutputIt adjacent\_difference( InputIt first, InputIt last,
                                   OutputIt d\_first,
                                   BinaryOperation op );
```

PARAMETERS

first, last - the range of elements d_first - the beginning of the destination range op - binary operation function object that will be applied. The signature of the function should be equivalent to the following:

Ret fun(const Type1 &a, const Type2 &b);

The signature does not need to have const &. The types Type1 and Type2 must be such that an object of type iterator_traits<InputIt>::value_type can be implicitly converted to both of them. The type Ret must be such that an object of type OutputIt can be dereferenced and assigned a value of type Ret.

Type requirements -InputIt must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

It to the element past the last element written.

5.5.5 std::partial_sum

NAME

std::partial_sum - Computes the partial sums of the elements in the subranges of the range [first, last) and writes them to the range beginning at d_first. The first version uses operator+ to sum up the elements, the second version uses the given binary function op.

SYNOPSIS

#include <numeric>

```
 template< class InputIt, class OutputIt >
OutputIt partial\_sum( InputIt first, InputIt last, OutputIt d\_first );
 template< class InputIt, class OutputIt, class BinaryOperation >
OutputIt partial\_sum( InputIt first, InputIt last, OutputIt d\_first,
                      BinaryOperation op );
```

PARAMETERS

first, last - the range of elements to sum d_first - the beginning of the destination range op - binary operation function object that will be applied. The signature of the function should be equivalent to the following:

Ret fun(const Type1 &a, const Type2 &b);

The signature does not need to have const &. The type Type1 must be such that an object of type iterator_traits<InputIt>::value_type can be implicitly converted to Type1. The type Type2 must be such that an object of type InputIt can be dereferenced and then implicitly converted to Type2. The type Ret must be such that an object of type iterator_traits<InputIt>::value_type can be assigned a value of type Ret.

Type requirements -InputIt must meet the requirements of InputIterator. -OutputIt must meet the requirements of OutputIterator.

RETURN VALUE

Iterator to the element past the last element written.

5.6. Input/output library

5.6.1 std::dec

NAME

std::dec - Modifies the default numeric base for integer I/O

SYNOPSIS

#include <ios>

```
std::ios\_base\& dec( std::ios\_base\& str );
std::ios\_base\& hex( std::ios\_base\& str );
std::ios\_base\& oct( std::ios\_base\& str );
```

PARAMETERS

str - reference to I/O stream

RETURN VALUE

str (reference to the stream after manipulation)

5.6.2 std::fixed

NAME

std::fixed - Modifies the default formatting for floating-point input/output.

SYNOPSIS

#include <ios>

```
std::ios\_base\& fixed( std::ios\_base\& str );
std::ios\_base\& scientific( std::ios\_base\& str );
std::ios\_base\& hexfloat( std::ios\_base\& str ); [since C++11]
std::ios\_base\& defaultfloat( std::ios\_base\& str ); [since C++11]
```

PARAMETERS

str - reference to I/O stream

RETURN VALUE

str (reference to the stream after manipulation)

5.6.3 std::boolalpha

NAME

std::boolalpha - 1) Enables the boolalpha flag in the stream str as if by calling str.setf(std::ios_base::boolalpha)

SYNOPSIS

#include <ios>

```
std::ios\_base\& boolalpha( std::ios\_base\& str );
std::ios\_base\& noboolalpha( std::ios\_base\& str );
```

PARAMETERS

str - reference to I/O stream

RETURN VALUE

str (reference to the stream after manipulation)

Bibliography

[1] Dr. Brian Dean. Milk routing, 2012.

[2] Dr. Brian Dean. Dynamic programming practice problems, 2013.

[3] Dr. Brian Dean. Optimal milking, 2013.

[4] Dr. Brian Dean. Dueling gpss, 2014.

[5] The C++ Resources Network. cplusplus.com, 2013.

[6] C++ Reference. cppreference.com, 2015.

[7] ACM Southeast Regional. Acm southeast regional 2013, 2013.

[8] ACM Southeast Regional. Acm southeast regional 2014, 2014.

[9] USACO. Farm tour, 2003.

Index

GNU Free Documentation License

Version 1.3, 3 November 2008

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled “Acknowledgements” or “Dedications”, Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled “Endorsements”. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled “Endorsements” or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

   A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   (a) The work must carry prominent notices stating that you modified it, and giving a relevant date.
   (b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".
   (c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless

of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

(d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

(a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

(b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

(c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

(d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

(e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

   "Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

   When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

(a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
(b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
(c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
(d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
(e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
(f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

   You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

   You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

End of Terms and Conditions

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear>  <name of author>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program>  Copyright (C) <year>  <name of author>
```

```
This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.