Important Algorithms for IOI

Nebhrajani A.V.

January 18, 2021

Contents

1: Introduction

Most problems can be reduced to either one of or a combination of the algorithms presented in this notebook. While not meant to be as comprehensive as a set of notes on CLRS, it includes the more challenging and important programs. Sorters and searchers (with the exception of binary search, which I struggled with early on) are omitted since they're basic and C++'s STL implements them well enough. This notebook focuses majorly on graph algorithms, dynamic programming, greedy programs, and trees. It assumes familiarity with C/C++ and basic programming techniques such as arrays/vectors, loops, lists, iteration, and recursion.

Note that it is bad practice to use the header `bits/stdc++.h`: it is **not** a standard header, and increases compile time by including *every* C++ library, not only the required ones. However, it's okay in competitive programming since the programs are small and runtime matters more than compile time. Obviously, if compilation takes too long, the bottleneck is probably this header.

2: Searchers

2.1 Binary Search

```cpp
// To search a sorted array for a value.
#include <bits/stdc++.h>

int binary_search(std::vector<int>& a, int k, int l, int r)
{
  if (r >= l) {
    int mid = (l+r)/2;
    if (k == a[mid]) {
      return mid;
    }
    if (a[mid] < k) {
      return binary_search(a, k, mid+1, r);
    }
    if (a[mid] > k) {
      return binary_search(a, k, l, mid-1);
    }
  }
  return -1;
}

int main()
{
  std::vector<int> arr = {34,54,23,35,13,35,46,678,34,576,46,34,646};
  std::sort(std::begin(arr), std::end(arr));
  int find = binary_search(arr, 46, 0, arr.size());
  std::cout << arr[find] << "\n";
  return 0;
}
```

Complexity: $O(\log_2 n)$ A basic algorithm, with a lot of similarity to QuickSort. r and l are the indexes of the subarray to search recursively. If $r < l$ directly return -1, the element wasn't found. If not, see if the current *mid* is the element we're looking for. If so, return *mid*. Otherwise, recurse on either the LHS or RHS subarray.

3: Graphs

Graphs can be represented as adjacency matrices and adjacency lists. Most algorithms use adjacency lists. Always write a function returning `void` called `add_edge` that adds an edge. It can be modified depending on whether the graph is directed and weighted.

3.1 Graph Searchers

3.1.1 Breath First Search (BFS)

```cpp
// BFS to check if two nodes are connected, and can calculate shortest path
// where edge weights are 1, using the 'parent' vector and backtracking how we
// got there.
#include <bits/stdc++.h>

bool bfs(std::vector<std::vector<int>>& g, int src, int dest)
{
  std::vector<int> visited (g.size());
  // std::vector<int> parent (g.size());
  std::deque<int> q;

  visited[src] = 1;
  q.push_back(src);

  while (q.size() != 0) {
    int j = q[0];
    for (int i = 0; i < g[j].size(); ++i) {
      if (visited[g[j][i]] == 0) {
        visited[g[j][i]] = 1;
        // parent[g[j][i]] = j;
        q.push_back(g[j][i]);
      }
    }
    q.pop_front();
  }
  // for (auto x: parent) {
  //       std::cout << x << "\n";
  // }


  if (visited[dest] == 1) {
    return true;
  }
  else {
    return false;
  }
}

void add_edge(std::vector<std::vector<int>>& g, int a, int b)
{
  g[a].push_back(b);
  g[b].push_back(a);
}

int main()
{
  int V = 10;
  std::vector<std::vector<int>> g (V);
  add_edge(g,0,1);
  add_edge(g,0,2);
  add_edge(g,1,2);
  add_edge(g,0,3);
  add_edge(g,3,4);
  add_edge(g,3,7);
  add_edge(g,4,5);
  add_edge(g,5,7);
  add_edge(g,5,6);
  add_edge(g,5,8);
  add_edge(g,7,8);
  add_edge(g,8,9);

  if (bfs(g,0,9)) {
    std::cout << "True" << "\n";
  }
  else {
    std::cout << "False" << "\n";
  }

  return 0;
}
```

Complexity: $O(m + n)$ Adjacency list representation. Undirected and unweighted in this case. Usual declaration using `std::vector<std::vector<int>>` over `std::vector<std::list<int>>` since vectors are faster with end appends. Add the edges.

The procedure `bfs` maintains a `deque q`. Start at the `src` node. Visit all of its neighbours who are unvisited. (For `src`, this is by definition *all* its neighbours.) Push them into `q` to visit later. Pop the `q` and push the unvisited neighbours of the node we popped. Keep doing this. At some point, all possible nodes visitable from `src` will have been visited, and `q` will be empty. We keep track of visited nodes using a standard vector `visited`.

Note that if we maintain a `parent` vector, that keeps track of *from whom* we visited the current node, we can backtrace the vector to get the path from `src` to `dest`. This is by definition the shortest path, since BFS doesn't go out of its way to explore longer paths: it follows the most direct (least edges) route from `src` to `dest`. This means BFS is a shortest path finder for unweighted graphs (all edge weights = 1).

3.1.2 Depth First Search (DFS)

```cpp
// DFS: the 'stack' is maintained as recursive function calls implicitly.

#include <bits/stdc++.h>

```

```cpp
void dfs(std::vector<std::vector<int>>& g, std::vector<int>& visited,
  ↪  std::vector<int>& parent, int src)
{
  visited[src] = 1;
  for (int i = 0; i < g[src].size(); ++i) {
    if (visited[g[src][i]] == 0) {
      parent[g[src][i]] = src;
      dfs(g, visited, parent, g[src][i]);
    }
  }
}

void add_edge(std::vector<std::vector<int>>& g, int a, int b)
{
  g[a].push_back(b);
  g[b].push_back(a);
}

int main()
{
  int V = 10;
  std::vector<std::vector<int>> g (V);
  std::vector<int> visited (V);
  std::vector<int> parent (V);

  add_edge(g,0,1);
  add_edge(g,0,2);
  add_edge(g,1,2);
  add_edge(g,0,3);
  add_edge(g,3,4);
  add_edge(g,3,7);
  add_edge(g,4,5);
  add_edge(g,5,7);
  add_edge(g,5,6);
  add_edge(g,5,8);
  add_edge(g,7,8);
  add_edge(g,8,9);

  dfs(g,visited,parent,0);
  std::cout << visited[9] << "\n";
  return 0;
}
```

Complexity: $O(m+n)$ DFS maintains a stack. It goes as far along a random path as possible, pushing the node it left from into a stack. As soon as it reaches the end of a path (i.e., it cannot visit an unvisited node), it pops out and tries the same from the previous node it left. This stack can be maintained implicitly by recursive calls to DFS, as shown in the program.

Note that DFS **does not** cover shortest paths, but it can be used for node numbering: by recording the 'time' DFS entered a node (started recursively calling its neighbours) and left it (exited out of its last remaining neighbour and by extension, the node in question).

3.1.3 Applications of Graph Searchers

1. Connectivity: Some parts of a graph may not be connected to other parts of the graph, creating sets of connected nodes. To figure out what exactly these sets *are*, run either search from a `src` (say 0). Observe the resultant `visited` array. The nodes that have been visited are from the first set. Note them down if you need to. Now run a search from an unvisited node. The additional nodes visited are the second set. Repeat until all nodes are visited, and you'll know the connectivity sets of the graph.
2. (TODO) Catching cycles: Using DFS Numbering

   More advanced concept, catches cycles in graphs using the numbers DFS assigns as it visits nodes.

3.2 Shortest Paths

3.2.1 Single Source: Djikstra's Algorithm

```cpp
#include <bits/stdc++.h>

struct node
{
  int dest;
  int weight;
};

void add_edge(std::vector<std::vector<node>>& g, int src, int dest, int
↪  weight)
{
  g[src].push_back({dest, weight});
  g[dest].push_back({src, weight});
}

void djikstra(std::vector<std::vector<node>>& g, std::vector<int>& distance,
↪  int src)
{
  std::vector<bool> visited (g.size());
  distance[src] = 0;

  for (int i = 0; i < g.size(); ++i) {
    int min = std::numeric_limits<int>::max();
    int u;
    for (int j = 0; j < visited.size(); ++j) {
      if (!visited[j]) {
        if (distance[j] < min) {
          min = distance[j];
          u = j;
        }
      }
    }
    visited[u] = true;
    for (int k = 0; k < g[u].size(); ++k) {
      if (!visited[g[u][k].dest] && distance[g[u][k].dest] > distance[u] +
      ↪  g[u][k].weight) {
        distance[g[u][k].dest] = distance[u] + g[u][k].weight;
      }
    }
  }
}

int main()
{
  int V = 7;
  std::vector<std::vector<node>> g (V);
  add_edge(g,0,2,80);
  add_edge(g,0,1,10);
  add_edge(g,1,2,6);
  add_edge(g,2,3,70);
  add_edge(g,1,4,20);
  add_edge(g,4,6,10);
  add_edge(g,4,5,50);
  add_edge(g,5,6,5);
  std::vector<int> distance (V, std::numeric_limits<int>::max());

  djikstra(g, distance, 0);
  for (int x: distance) {
    std::cout << x << "\n";
  }

  return 0;
}
```

Djikstra's algorithm finds the shortest path from `src` to every other vertex in the graph. It does so by making the locally optimum choice ('greedy'). This results in the correct result by induction, since any node that has been visited must have been visited by a shortest path, and any extension chooses the shortest route to that node.

Now, our implementation of Djikstra's works just fine, but its complexity is $O(n^2)$, because of the nested loop that searches for the minimum element. This is quite simply *horrible*, since the loop that's actually doing the work using the adjacency list for the edges requires only $O(m)$ time. What is done is practice is that a heap or red-black-tree is used so that the operations `decrease-key` and `get-min-element` require constant time rather than linear. This gives us a speedup to complexity $O((n+m)\log n)$.

In C++, the STL provides `std::set`, which used a balanced binary search tree to manage data, meaning it's perfect in this application: to access the minimum element, we only have to call `set.begin()`. This is implemented in `GREATESC`, given below. Note that we use C++'s inbuilt `pair` instead of defining our own `struct node`, since it by default compares the first value (weight). This prevents us from having to write a comparator function and passing it to `set`.

```
#include<bits/stdc++.h>

# define INF 0x3f3f3f3f

void add_edge(std::vector<std::list<std::pair<int,int>>>& graph, int u, int v)
```

```cpp
{
  graph[u].push_back(std::make_pair(v, 1));
  graph[v].push_back(std::make_pair(u, 1));
}

int shortest_path(std::vector<std::list<std::pair<int,int>>>& graph, int src,
↪  int dest, int V)
{
  std::set<std::pair<int,int>> setds;
  std::vector<int> dist(V, INF);
  setds.insert(std::make_pair(0, src));
  dist[src] = 0;

  while (!setds.empty()) {
    std::pair<int,int> tmp = *(setds.begin());
    setds.erase(setds.begin());
    int u = tmp.second;
    std::list<std::pair<int,int>>::iterator i;
    for (i = graph[u].begin(); i != graph[u].end(); ++i) {
      int v = (*i).first;
      int weight = (*i).second;
      if (dist[v] > dist[u] + weight) {
        if (dist[v] != INF) {
          setds.erase(setds.find(std::make_pair(dist[v], v)));
        }
        dist[v] = dist[u] + weight;
        setds.insert(std::make_pair(dist[v], v));
      }
    }
  }
  return dist[dest];
}

int main()
{
  int V;
  int N;
  int a;
  int b;
  std::cin >> V >> N;
  std::vector<std::list<std::pair<int,int>>> g (V);

  for (int i = 0; i < N; ++i) {
    std::cin >> a >> b;

    add_edge(g, a-1, b-1);
  }

  std::cin >> a >> b;
  int x = shortest_path(g, (a-1), (b-1), V);
  if (x == INF) {
    x = 0;
  }
  std::cout << x << "\n";

  return 0;
}
```

A good and useful visualisation of Djikstra's algorithm is that of a fire starting at $t = 0$ at the source, and running along every path connected to it at a speed of 1 unit per second. Every time it reaches a vertex, note down t. That's the shortest path distance from `src` to that vertex. This is also useful for Djikstra's proof of correctness. Any vertex that is in the set of 'burnt' vertices has been reached (by induction) via a shortest path. The base case is obvious: `dist[src] = 0`. We now visit every `unvisited` vertex that's a neighbour of `src`. We choose the minimum out of these, meaning we make a locally optimum choice, and mark it visited. The two vertices in the `visited` set are a shortest path. Why? Assume the vertex added was `A`. It was the vertex at minimum distance from `src`. For any other path to be a shorter path from `src` to `A`, it will use some other neighbour of `src`, plus its own path distance to `A`. But, `A` is the closest neighbour of `src`, and is therefore the shortest path. Now, follow a similar chain of logic taking `A` as the new source. It won't 'back-visit' `src` since both are in the visited set.

Note that we simultaneously maintain an array called `expected-burn-times` that can be thought of as the update array. In the zeroth iteration, all values in this array are $+\infty$. In the first iteration, all neighbours of `src` are marked with their distance from `src`. If there's a shorter path that comes up later, from `A`, or `A`'s nearest neighbour, or so on, it's updated. This also means $\mathtt{A}_\mathtt{i}$'s neighbour selected may not be the one with the least absolute distance from it, but instead the one with the least value in `unvisited` – it's greedy. All selected vertices are marked visited, and all have been reached by a shortest path.

Visualise as a spreading fire, with times being marked per vertex.

3.2.2 Single Source, Negative Edge Weights: Bellman-Ford Algorithm

```cpp
#include <bits/stdc++.h>

struct node
{
  int dest;
  int weight;
};

void add_edge(std::vector<std::vector<node>>& g, int src, int dest, int
↪  weight)
{
  g[src].push_back({dest, weight});
}

void bellman(std::vector<std::vector<node>>& g, std::vector<int>& distance,
↪  int src)
{
  distance[src] = 0;

  for (int k = 0; k < g.size() - 1; ++k) {
    for (int i = 0; i < g.size(); ++i) {
      for (int j = 0; j < g[i].size(); ++j) {
        if (distance[i] != std::numeric_limits<int>::max()) {
          distance[g[i][j].dest] = std::min(distance[g[i][j].dest],
          ↪  distance[i] + g[i][j].weight);
        }
      }
    }
  }
}

int main()
{
  int V = 8;
  std::vector<std::vector<node>> g (V);
  add_edge(g,0,7,8);
  add_edge(g,0,1,10);
  add_edge(g,7,6,1);
  add_edge(g,6,5,-1);
  add_edge(g,6,1,-4);
  add_edge(g,1,5,2);
  add_edge(g,5,2,-2);
  add_edge(g,4,5,-1);
  add_edge(g,3,4,3);
  add_edge(g,2,3,1);
  add_edge(g,2,1,1);
  std::vector<int> distance (V, std::numeric_limits<int>::max());


  bellman(g, distance, 0);
  for (int x: distance) {
    std::cout << x << "\n";
  }

  return 0;
}
```

Bellman-Ford finds shortest paths when negative edge weights are included. Djikstra's algorithm cannot be used for this application since in making a locally optimum choice, it may miss out on some better path due to a negative weight somewhere. To fix this issue, Bellman-Ford 'runs' Djikstra $V - 1$ times, on the set of vertices whose distance is **not** infinity. What this does is it updates the direction vector (array) *every* time a better path is found. If a better path isn't found, it can 'update' it, but that doesn't change the value from its previously optimal state. This takes care of Djikstra's central idea – checking each vertex only once – which also contributes to Djikstra's higher efficiency. At best, Bellman-Ford has time complexity $O(mn)$ due to the $V - 1$ loop.

3.3 Minimum Spanning Trees

3.3.1 What is an MST?

A **tree** is a connected, acyclic graph. Connected means that there is no component (set of vertices) of the graph that **cannot** be reached from any other component. An acyclic graph is one where there is only one (unique) path from any two vertices i to j.

A **spanning tree** is one which includes all vertices of another graph G, that is, it *spans* all vertices of this graph.

A **minimum spanning tree** is a spanning tree which spans a weighted acyclic graph while costing the *minimum* in terms of edge weights.

3.3.2 Motivation

MSTs are interesting to, say, a network where every node must be connected to every other node with minimum possible cost. Other graph optimization problems where a spanning tree with low weight is required (generally in many networking problems), such as approximation of NP-complete problems and cluster analysis. It's also used in handwriting recognition.

3.3.3 Prim's Algorithm

Outline

Select the edge with lowest weight from G. Draw the tree. Look at the edge weights of neighbours of tree vertices. Select the lowest such that the tree will not become acyclic. Run $V - 1$ times.

Prim's is a greedy algorithm, so we require a proof of correctness.

Proof and Minimum Seperator Lemma

Let set V be partitioned into two non-empty sets of vertices U and W, and let e be the edge of minimum weight joining these two sets. We claim that every MST must include e.

To prove this by contradiction is easy: assume an edge e joining u (a member of U) and w (a member of W). Let this be the smallest edge joining U and W.

Consider some other edge e' joining U and W via path (u', w'), which is included in some ST.

Now, the weight of this ST could be reduced by using e instead of e' (by definition, e is the edge of minimum weight). Therefore, the minimum spanning tree must contain e.

Note that e' **must** be an edge that connects to u and w as well (part of the same component): we cannot replace any arbitrary edge with e.

The correctness of Prim's algorithm is now obvious: we always pick the minimum edge when adding an edge to the current tree, which must be the MST. Recall that we no longer need to start with the edge with globally the lowest weight: we can start with any vertex's lowest edge.

Implementation

```
#include <bits/stdc++.h>

struct tree_edge
{
  int src;
  int dest;
};

struct node
```

```
{
  int dest;
  int weight;
};

void add_edge(std::vector<std::vector<node>>& g, int src, int dest, int
↪  weight)
{
  g[src].push_back({dest, weight});
  g[dest].push_back({src, weight});
}

void prim(std::vector<std::vector<node>>& g, std::vector<tree_edge>&
↪  tree_edges)
{
  std::vector<bool> visited (g.size());
  std::vector<int> nbr (g.size());
  std::vector<int> distance (g.size(), std::numeric_limits<int>::max());

  visited[0] = true;
  distance[0] = 0;

  for (int i = 0; i < g[0].size(); ++i) {
    nbr[g[0][i].dest] = 0;
    distance[g[0][i].dest] = g[0][i].weight;
  }

  for (int i = 1; i < g.size(); ++i) {
    int min = std::numeric_limits<int>::max();
    int u;
    for (int j = 0; j < g.size(); ++j) {
      if (!visited[j]) {
        if (distance[j] < min) {
          min = distance[j];
          u = j;
        }
      }
    }
    visited[u] = true;
    tree_edges.push_back({u, nbr[u]});
    for (int j = 0; j < g[u].size(); ++j) {
      if (!visited[g[u][j].dest]) {
        if (distance[g[u][j].dest] > g[u][j].weight) {
          distance[g[u][j].dest] = g[u][j].weight;
          nbr[g[u][j].dest] = u;
        }
      }
    }
  }
}

int main()
{
  int V = 7;
  std::vector<std::vector<node>> g (V);
  add_edge(g,0,1,10);
  add_edge(g,0,2,18);
  add_edge(g,1,2,6);
  add_edge(g,2,3,70);
  add_edge(g,1,4,20);
  add_edge(g,4,6,10);
  add_edge(g,4,5,10);
  add_edge(g,5,6,5);

  std::vector<tree_edge> tree_edges;
  prim(g, tree_edges);

  for (auto x: tree_edges) {
    std::cout << x.src << " " << x.dest << "\n";
  }
  return 0;
}
```

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

Note that this is exactly like Djikstra's algorithm, albeit with a different update function and with recording the tree in a vector called `tree_edges`. Again, this implementation doesn't use a heap for the `min_find` operation, which would reduce the complexity from $O(n^2)$ to $O((m+n)\log n)$.

3.3.4 Kruskal's Algorithm

Outline

Pick edges in ascending order, and add them to the tree. The tree will initially not necessarily be connected. Keep adding edges in ascending order, then connect the components.

Proof

Kruskal's algorithm involves merging disjoint components of a graph. During this `merge` operation, Kruskal's algorithm selects the globally minimum edge that **does not form a cycle.** This means that adding this edge is the equivalent of the edge chosen by the minimum seperator lemma: it's joining two disjoint components using the shortest possible edge.

Implementation

4: Dynamic Programming

4.1 Fibonacci

$$T_0 = 0, T_1 = 1$$
$$T_n = T_{n-1} + T_{n-2}$$

4.1.1 Memoized

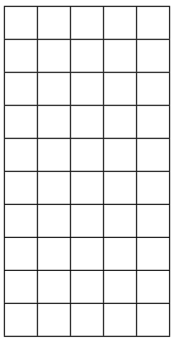
```cpp
#include <bits/stdc++.h>

int fib(std::vector<int>& fibtable, int n)
{
  int value;
  if (fibtable[n] != 0) {
    return fibtable[n];
  }
  if (n == 0 || n == 1) {
    value = n;
  }
  else {
    value = fib(fibtable, n-1) + fib(fibtable, n-2);
  }
  fibtable[n] = value;

  return value;
}

int main()
{
  int n = 7;
  std::vector<int> fibtable (n+1);
  std::cout << fib(fibtable, n) << "\n";

  return 0;
}
```

4.1.2 DP

```cpp
#include <bits/stdc++.h>

int fib(int x)
{
  std::vector<int> fibtable (x+1);
  fibtable[0] = 0;
  fibtable[1] = 1;
  for (int i = 2; i <= x; ++i) {
    fibtable[i] = fibtable[i-1] + fibtable[i-2];
  }
  return fibtable[x];
}

int main()
{
  std::cout << fib(7) << "\n";
  return 0;
}
```

4.2 Grid Paths

Interesting problem: given a grid:



We want to go from $(0, 0)$ to $(5, 10)$. How many ways are there of doing this? (Only up and right moves allowed.)

4.2.1 Combinatorics

15 moves must be made in *all* possible paths. 5 of these will be horizontal, and the rest vertical. Or, if we were to write moves using the notation ↑→, selecting only rightward moves:

$$\ldots\_ \rightarrow \_ \rightarrow \_ \rightarrow\rightarrow\rightarrow \_\ldots$$

Clearly, we must select any 5 spots out of 15, or, equivalently, any 10 spots out of 15. Our solution is:

$$\binom{15}{5} = \binom{15}{10} = 3003$$

4.2.2 Induction

Consider an intersection (i, j). How did we get here? Either from $(i - 1, j)$ or $(i, j - 1)$. Therefore,

$$p(i, j) = p(i - 1, j) + p(i, j - 1)$$

where $p()$ is a function that returns number of paths. It's always important to check boundary cases, because things often change there.

$$p(i, 0) = p(i - 1, 0)$$

$$p(0, j) = p(0, j-1)$$

And they do, there's no -1 position. In fact, this'll probably mess things up in a language like Python where -1 refers to the *last* element of an array. Anyway, the base case becomes:

$$p(0, 0) = 1$$

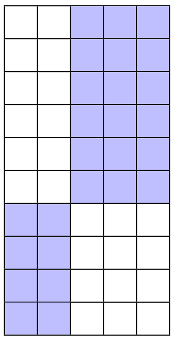
```cpp
#include <iostream>

int naive_recursion_path(int i, int j)
{
  if (i == 0 && j == 0) {
    return 1;
  }
  if (i == 0) {
    return naive_recursion_path(0,j-1);
  }
  if (j == 0) {
    return naive_recursion_path(i-1,0);
  }
  return naive_recursion_path(i-1,j) + naive_recursion_path(i,j-1);
}

int main()
{
  std::cout << naive_recursion_path(5,10) << "\n";
  return 0;
}
```

```
3003
```

We get the correct answer. The clever will recognize this grid as a Pascal's triangle, with $(0, 0)$ as the root vertex. This further substantiates our claim of $\binom{15}{5}$: it's a binomial coefficient of the 15th level.

4.2.3 Holes

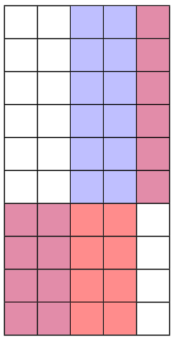
This is where the problem gets tricky: holes.



Assume you cannot use the intersection at $(2, 4)$. Resolving it using combinatorics, we remove paths from $(0, 0)$ to $(2, 4)$ and from $(2, 4)$ to $(5, 10)$. We then multiply these to get all possible combinations, and subtract these from 3003.

$$\binom{15}{5} - \binom{6}{2} \times \binom{9}{3} = 3003 - 1260 = 1743$$

It is obvious that this problem becomes difficult for two holes onward, since we may have to use the inclusion-exclusion principle to remove overlaps, extra counts, or undercounts. Consider an extra hole at $(4, 4)$.



The overlaps make this an annoying problem to solve. This is, however, incredibly simple with the inductive/recursive approach: just set the paths function to return 0 at the intersections where holes exist.

```cpp
#include <iostream>

int naive_recursion_path(int i, int j)
{
  if (i == 2 && j == 4) {
    return 0;
  }
  if (i == 0 && j == 0) {
    return 1;
  }
  if (i == 0) {
    return naive_recursion_path(0,j-1);
  }
  if (j == 0) {
    return naive_recursion_path(i-1,0);
  }
  return naive_recursion_path(i-1,j) + naive_recursion_path(i,j-1);
}

int main()
{
  std::cout << naive_recursion_path(5,10) << "\n";
  return 0;
}
```

```
1743
```

Naturally, we can define an arbitrary number of holes this way.

4.2.4 DP

It is obvious that the naive recursion approach recalculates many values in the table. To reduce the complexity, we can use both memoization or DP. DP is generally faster, although there are some cases where a top-down memoized approach *may* be better. (An example is given later.)

As we do in DP, we must first reduce the problem to a DAG, then solve it in topological order **iteratively**. This is obvious, it's the same grid, with arrows pointing up and right in each cell. Therefore, the bottom left intersection, $(0, 0)$ is where we start, working column by column, row by row, or diagonal by diagonal. For the sake of this program, let's use row by row.

```cpp
#include <bits/stdc++.h>

int dp_path(int p, int q)
{
  int i = p+1;
  int j = q+1;
  int dp[j][i];

  for (int l = 0; l < i; ++l) {
    dp[0][l] = 1;
  }

  for (int l = 0; l < j; ++l) {
    dp[l][0] = 1;
  }

  for (int l = 1; l < j; ++l) {
    for (int k = 1; k < i; ++k) {
      dp[l][k] = dp[l-1][k] + dp[l][k-1];
    }
  }
  return dp[q][p];
}

int main()
{
  std::cout << dp_path(5,10) << "\n";
  return 0;
}
```

```
3003
```

We can now add holes, thus.

```cpp
#include <bits/stdc++.h>

int dp_path(int p, int q)
{
  int i = p+1;
  int j = q+1;
  int dp[j][i];

  for (int l = 0; l < i; ++l) {
    dp[0][l] = 1;
  }

  for (int l = 0; l < j; ++l) {
    dp[l][0] = 1;
  }

  for (int l = 1; l < j; ++l) {
    for (int k = 1; k < i; ++k) {
      dp[l][k] = dp[l-1][k] + dp[l][k-1];
      if (l == 4 && k == 2) {
        dp[l][k] = 0;
      }
    }
  }
  return dp[q][p];
}

int main()
{
  std::cout << dp_path(5,10) << "\n";
  return 0;
}
```

```
1743
```

With the additional hole at $(4, 4)$:

```
#include <bits/stdc++.h>

int dp_path(int p, int q)
{
  int i = p+1;
  int j = q+1;
  int dp[j][i];

  for (int l = 0; l < i; ++l) {
    dp[0][l] = 1;
  }

  for (int l = 0; l < j; ++l) {
    dp[l][0] = 1;
  }

  for (int l = 1; l < j; ++l) {
    for (int k = 1; k < i; ++k) {
      dp[l][k] = dp[l-1][k] + dp[l][k-1];
      if ((l == 4 && k == 2) || (l == 4 && k == 4)) {
        dp[l][k] = 0;
      }
    }
  }
  return dp[q][p];
}

int main()
{
  std::cout << dp_path(5,10) << "\n";
  return 0;
}
```

```
1358
```

4.3 Longest Common

4.3.1 Subword

Consider two strings:

$$u = a_1 a_2 \dots a_n$$

$$v = b_1 b_2 \dots b_m$$

Our task is to find the longest common (contiguous) substring of these two strings, that is, k for some

$$a_i \dots a_{i+k-1} = b_j \dots b_{j+k-1}$$

This is inherently an extremely recursive/inductive problem. Consider a function $l()$ returning the length of the longest common substring. Then,

$$l(i,j) = \begin{cases} 0 & a_i \neq b_j \\ 1 + l(i+1, j+1) & a_i = b_j \end{cases}$$

where i and j are the indexes of the strings we're checking. We must, however, also define boundary cases, for when we reach the end of either string.

$$l(n+1, j) = 0$$

$$l(i, m+1) = 0$$

Now, the structure of this problem means that when we try to solve it using DP, we will need the result of $l(i+1, j+1)$ before we can find $l(i,j)$. Therefore, we create a DP matrix, which looks like this:

		0	1	2	3	4	5	6
		s	e	c	r	e	t	∅
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	∅							

We can fill it up starting from the bottom right corner. This time, let's fill it up column by column, after initialising the boundary cases. The completed matrix is:

		0	**1**	**2**	**3**	**4**	**5**	**6**
		s	e	c	r	e	t	∅
0	**b**	0	0	0	0	0	0	0
1	**i**	0	0	0	0	0	0	0
2	**s**	**3**	0	0	0	0	0	0
3	**e**	0	2	0	0	1	0	0
4	**c**	0	0	1	0	0	0	0
5	**t**	0	0	0	0	0	1	0
6	∅	0	0	0	0	0	0	0

Putting this into C++:

```cpp
#include <iostream>
#include <cstring>

int lcw(char *u, char *v)
{
  int p = strlen(v);
  int q = strlen(u);
  int dp[p + 1][q + 1];
  for (int i = 0; i < p+1; ++i) {
    dp[i][q] = 0;
  }
  for (int i = 0; i < q+1; ++i) {
    dp[p][i] = 0;
  }
  int greatest = 0;
  for (int i = q-1; i >= 0; --i) {
    for (int j = p-1; j >= 0; --j) {
      if (u[i] == v[j]) {
        dp[j][i] = 1 + dp[j+1][i+1];
        if (dp[j][i] > greatest) {
          greatest = dp[j][i];
        }
      }
      else {
        dp[j][i] = 0;
      }
    }
  }
  for (int i = 0; i < p+1; ++i) {
    for (int j = 0; j < q+1; ++j) {
      std::cout << dp[i][j] << " ";
    }
    std::cout << "\n";
  }
  return greatest;
}

int main()
{
  std::cout << lcw("secret", "bisect") << "\n";
```

```
    return 0;
}
```

```
0  0  0  0  0  0  0
0  0  0  0  0  0  0
3  0  0  0  0  0  0
0  2  0  0  1  0  0
0  0  1  0  0  0  0
0  0  0  0  0  1  0
0  0  0  0  0  0  0
3
```

The complexity of this solution is $O(mn)$.

4.3.2 Subsequence

Our task is to find the longest non-contiguous common letter sequence. For example, in `bisect` and `secret`, the longest subsequence is `sect`.

Now, this problem is a bit gnarly. The trick is to realise one crucial point, illustrated below.

$$\begin{array}{cccccc} \mathrm{a}_0 & \mathrm{a}_1 & \mathrm{a}_2 & \mathrm{a}_3 & \mathrm{a}_4 & \mathrm{a}_5 \\ \mathrm{b}_0 & \mathrm{b}_1 & \mathrm{b}_2 & \mathrm{b}_3 & \mathrm{b}_4 & \mathrm{b}_5 \end{array}$$

Cases:

1. If a_0 and b_0 match, it's all good, and we can continue to check for a_1 and b_1, and so on. This is similar to our first inductive conclusion in longest subword.

$$\begin{array}{cccccc} \mathrm{a}_0 & \mathrm{a}_1 & \mathrm{a}_2 & \mathrm{a}_3 & \mathrm{a}_4 & \mathrm{a}_5 \\ \mathrm{b}_0 & \mathrm{b}_1 & \mathrm{b}_2 & \mathrm{b}_3 & \mathrm{b}_4 & \mathrm{b}_5 \end{array}$$

2. If a_0 and b_0 do **not** match, we cannot discard either of them. This is because a_0 may match with some other b_j, or b_0 may match with some other a_i:

$$\begin{array}{cccccc} \mathrm{a}_0 & \mathrm{a}_1 & \mathrm{a}_2 & \mathrm{a}_3 & \mathrm{a}_4 & \mathrm{a}_5 \\ \mathrm{b}_0 & \mathrm{b}_1 & \mathrm{b}_2 & \mathrm{b}_3 & \mathrm{b}_4 & \mathrm{b}_5 \end{array}$$

$$\begin{array}{cccccc} \mathrm{a}_0 & \mathrm{a}_1 & \mathrm{a}_2 & \mathrm{a}_3 & \mathrm{a}_4 & \mathrm{a}_5 \\ \mathrm{b}_0 & \mathrm{b}_1 & \mathrm{b}_2 & \mathrm{b}_3 & \mathrm{b}_4 & \mathrm{b}_5 \end{array}$$

   The two cases are either ignoring a_0 or b_0, not both, and checking forward. This can be done inductively, always checking for the next index onward with one of the values preserved. This can also be thought of as maintaining (say) a_0 'shifting' string b leftwards one unit, then rechecking, treating this new shifted string as a new string. This must be done for both, that is, we must check shifts for both strings.

   We must therefore check *both*, and choose the maximum out of them (recall that we are choosing the maximum length subsequence, so it's okay to forget about ones with shorter length).

3. Naturally, the boundary/edge cases remain the same.

Our function, $L()$, therefore, has the following cases:

$$L(i,j) = \begin{cases} 1 + L(i+1, j+1) & a_i = b_j \\ \max(L(i+1,j), L(i,j+1)) & a_i \neq b_j \end{cases}$$

The edge cases remain the same:

$$L(n+1, j) = 0$$
$$L(i, m+1) = 0$$

Again, we can build a matrix that looks a lot like the previous one, with dependencies in three directions: a call to $L(i, j)$ needs to check the maximum on the right and below as well as know what's in $(i+1, j+1)$.

		0	1	2	3	4	5	6
		s	e	c	r	e	t	∅
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	∅							

Filling it up as a DP:

		0	**1**	**2**	**3**	**4**	**5**	**6**
		s	**e**	**c**	**r**	**e**	**t**	∅
0	**b**	4	3	2	2	2	1	0
1	**i**	4	3	2	2	2	1	0
2	**s**	4	3	2	2	2	1	0
3	**e**	3	3	2	2	2	1	0
4	**c**	2	2	2	1	1	1	0
5	**t**	1	1	1	1	1	1	0
6	∅	0	0	0	0	0	0	0

Note that we can just look at the value of $(0, 0)$ to get the answer, since the maximum propagates to the top and left. We don't have to explicitly maintain a `max` variable.

Expressing this in C++:

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>

int lcs(char *u, char *v)
{
  int p = strlen(v);
  int q = strlen(u);
  int dp[p + 1][q + 1];
  for (int i = 0; i < p+1; ++i) {
    dp[i][q] = 0;
  }
  for (int i = 0; i < q+1; ++i) {
    dp[p][i] = 0;
  }
  for (int i = q-1; i >= 0; --i) {
    for (int j = p-1; j >= 0; --j) {
      if (u[i] == v[j]) {
        dp[j][i] = 1 + dp[j+1][i+1];
      }
      else {
        dp[j][i] = std::max(dp[j+1][i], dp[j][i+1]);
      }
    }
  }
  for (int i = 0; i < p+1; ++i) {
    for (int j = 0; j < q+1; ++j) {
      std::cout << dp[i][j] << " ";
    }
    std::cout << "\n";
  }
  return dp[0][0];
}
int main()
{
  std::cout << lcs("secret", "bisect") << "\n";
  return 0;
}
```

```
4 3 2 2 2 1 0
4 3 2 2 2 1 0
4 3 2 2 2 1 0
3 3 2 2 2 1 0
2 2 2 1 1 1 0
1 1 1 1 1 1 0
0 0 0 0 0 0 0
4
```

This program also runs in $O(mn)$ time.

4.4 Edit Distance

The edit distance between two strings is merely how many operations apart they are. This naturally varies according to which operations are allowed. For our study, we use Levenshtein distance, that is, the allowed operations are:

1. Insert a letter.
2. Delete a letter.
3. Substitute a single letter with another one.

Converting this to an inductive definition, for a function $E()$ returning the edit distance, is the following.

$$\begin{array}{cccccc} \mathrm{a}_0 & \mathrm{a}_1 & \mathrm{a}_2 & \mathrm{a}_3 & \mathrm{a}_4 & \mathrm{a}_5 \\ \mathrm{b}_0 & \mathrm{b}_1 & \mathrm{b}_2 & \mathrm{b}_3 & \mathrm{b}_4 & \mathrm{b}_5 \end{array}$$

1. If $\mathrm{a}_0 = \mathrm{b}_0$, there's nothing to do (no edits to be made), move on to $E(i+1, j+1)$.
2. If $\mathrm{a}_0 \neq \mathrm{b}_0$,
   (a) We can substitute one for the other, and move on. This should return $1 + E(i+1, j+1)$.
   (b) We can delete a_0. The function should then return $1 + E(i+1, j)$.
   (c) We can insert b_0 at $i-1$ in a, meaning we now must find $1 + E(i, j+1)$.

This is pretty similar to the definition for $L()$, with the exception that we will seek the minimum value for (2) rather than the maximum, since we want the *minimum* edit distance.

$$E(i,j) = \begin{cases} E(i+1, j+1) & a_i = b_j \\ 1 + \min(E(i+1, j+1), E(i+1, j), E(i, j+1)) & a_i \neq b_j \end{cases}$$

The edge cases, however, have a major change. This is because of the fact that the edit distance between a $\varnothing$ string and a string of non-negative length is the length of the non-null string. Therefore,

$$E(n+1, j) = m - j + 1$$
$$E(i, m+1) = n - i + 1$$

We skip to writing the code for this example, since the matrices are very similar to the previous problems, with the exception that on a mismatch, we seek to add the minimum of the surrounding three values with one. In C++,

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>

int ed(char *u, char *v)
{
  int p = strlen(v);
  int q = strlen(u);
  int dp[p + 1][q + 1];
  for (int i = 0; i < p+1; ++i) {
    dp[i][q] = p-i;
  }
  for (int i = 0; i < q+1; ++i) {
    dp[p][i] = q-i;
  }
  for (int i = q-1; i >= 0; --i) {
    for (int j = p-1; j >= 0; --j) {
      if (u[i] == v[j]) {
        dp[j][i] = dp[j+1][i+1];
      }
      else {
        dp[j][i] = 1 + std::min(dp[j+1][i+1], std::min(dp[j+1][i],
        ↪  dp[j][i+1]));
      }
    }
  }
  for (int i = 0; i < p+1; ++i) {
    for (int j = 0; j < q+1; ++j) {
      std::cout << dp[i][j] << " ";
    }
    std::cout << "\n";
  }
  return dp[0][0];
}
int main()
{
  std::cout << ed("secret", "bisect") << "\n";
  return 0;
}
```

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

4.4.1 A Note on Backtracking and Space Complexity

To find the actual *edits* or the actual subsequence, we must backtrace in the matrix. This is done simply by taking our answer, and figuring out how we got there. This is easy for subwords, since we can also return the position where the last match was, and simply read the subword from either of the strings. However, for subsequences and edit distances, this can be harder. Consider our last edit matrix,

	s	e	c	r	e	t	∅
b	**4**	4	4	4	4	5	6
i	3	4	3	3	3	4	5
s	2	3	3	2	2	3	4
e	3	2	3	2	1	2	3
c	4	3	2	2	1	1	2
t	5	4	3	2	1	0	1
∅	6	5	4	3	2	1	0

We start from position $(0, 0)$. How could we have gotten there? Since s $\neq$ b, it must have been by taking the minimum element around it, and adding 1. Clearly, we got there from the 3 below. From there, again, we came from 2. From 2, since s = s, we came from the 2 diagonal to it. Again, e = e, and c = c. We then move two spaces left, reaching t = t. Finally, we reach the bottom left corner.

u / v	s	e	c	r	e	t	∅
b	**4**	4	4	4	4	5	6
i	**3**	4	3	3	3	4	5
s	**2**	3	3	2	2	3	4
e	3	**2**	3	2	1	2	3
c	4	3	**2**	2	1	1	2
t	5	4	3	**2**	**1**	**0**	1
∅	6	5	4	3	2	1	**0**

From this, the sequence of moves is easy to compute, a vertical downward move is deleting a letter from v. A horizontal rightward move is deleting a letter from u, or inserting a letter into v. A diagonal move can be one of two cases: if the two letters were equal, it's the null move, it does nothing. If the two letters were unequal, it was a substitution. Our complete traceback is therefore:

1. Delete 'b' from `bisect`.
2. Delete 'i' `isect`.
3. Insert 'r' into `sec_t`.
4. Insert 'e' into `secr_t`.

We therefore have edited `bisect` to `secret` in 4 moves.

On space complexity: the current space complexity is $O(mn)$, since we're storing the whole matrix. However, as in many DP problems, this isn't necessary. We only need two rows or columns at a time. This means that if we really want to optimize for space, we can basically forget about all but the column (or row) we're currently operating on. This will prevent a post-write traceback, we must record the moves we make as we write the two arrays. This method has space complexity $O(m)$ or $O(n)$. Whichever is smaller among m and n can be used, minimising space complexity.