Important Algorithms for IOI

Nebhrajani A.V.

December 7, 2020

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);
}

```

```cpp
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>

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)
{
  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`. Now, when any vertex wants to be added to the `visited` set, it must be an unvisited neighbour of `src`. It does this by choosing the locally best path to do so, which in this case is either the direct edge or through one of the previous neighbours. Since Djikstra's algorithm only marks a node visited when it *reaches* the node, it'll check its nearness to every one of its neighbours.

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.

4 Dynamic Programming

4.1 Fibonacci

$$T_0 = 0, T_1 = 1$$

$$T_n = T_{n-1} + T_{n-2}$$

4.1.1 Memoized

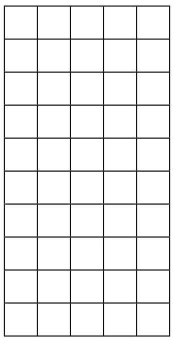
```cpp
#include <bits/stdc++.h>

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 $\uparrow\rightarrow$, 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:

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

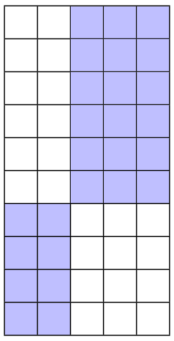
```cpp
#include <iostream>

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

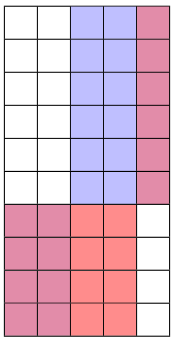
This is where the problem gets tricky: holes.



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
```