COMP4128 Notes

Alperen Onur

November 2024

1 C++ features

- *cin* is slower than $scanf()$, revert to scanf if necessary
- To insist on non-scientific notation, use std::fixed from $< iostream >$. To change number of decimal places use $setprecision()$ from $< iomanip >$
- Use the string stream type to take in a string and extract integers from it. In other cases, $stoi()$ might suffice.
- $pair < T, T >$
- The $< algorithm >$ header provides useful functions like $fill()$, $search()$, $count()$, $max()$, $min()$, $swap(x, y)$, $sort()$, etc.
- $next\_permutation()$ rearranges elements in a range into the lexicographically next permutation. This helps us run an exhaustive search (brute force).

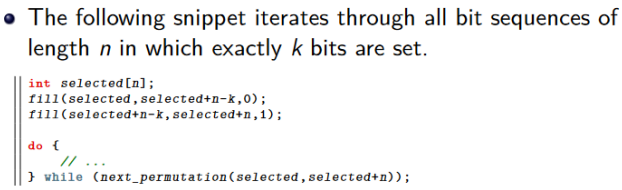

Figure 1: Next permutation

2 Debugging

Debugging can be tricky in a time environment. The key is to not panic and analyse your code carefully.

2.1 TIME-LIMIT-EXCEEDED (TLE)

Check for the following in the order they are listed.

- Check your complexity analysis. Computers can run 200 million computations per second. If your $n > 10,000$ then no amount of optimization will allow it to run quick enough. As a general rule, substitute your given max n into your complexity analysis and check to see if it runs quicker than 100 million. If I have n up to 10^5 and my complexity analysis is $O(nlogn)$, then this is $10^5 \times log(10^5) < 200,000,000$ and so our algorithm is quick enough.
- Check for infinite loops.
- If there are a lot of inputs via Iostream, you can speed this up by adding

```
int main() {
    cin.tie(nullptr);
    cin.sync_with_stdio(false);
```

- Replace vectors with arrays
- replace stacks and queues with arrays
- replace small sets with bitsets
- Look for computations that are repeated
- Sort your data to begin with.
- Remove debugging print statements.
- Change order of indices in a $2D$ array to increase caching. $grid[col][row] => grid[row][col]$.
- Consider early exiting.

2.2 RUN-ERROR/RUNTIME-ERROR

- Check for accessing out of array bounds.
- Check that your variables are initialized before using them.
- Check for accessing into or deleting from empty data structures.
- Compile with all flags using $-Wall$. $-Wextra$ and $-Wshadow$ may also help.

2.3 WRONG-ANSWER

- Write test cases. Think of possible edge cases.
- Take your time and read what you're typing. Open your code in a different editor.
- You can try run a slow brute force algorithm and compare it against yours. The below script compares your solution with a brute force solution. *soln* is an your solution executable, *brute_force* is the brute force executable and *gen* is a random data generator.

```
./gen > random.in; while diff <(./soln < random.in)
    <(./brute_force < random.in); do echo "all good"; ./gen
    > random.in; done
```

- Don't rely on sample cases, these are poor for actually testing a solution.
- Are there cases such that:
  1. a loop will never run
  2. a branch of an if-statement will not run
  3. the start is also the end? Where left = right? Usually for situations where the size of some data structure or the assignment of a variable is 0.
  4. the answer is 0 or negative?
  5. floating numbers aren't printed with enough precision?
  6. you've misinterpreted the bounds? can $N = 0$ or $N = 1$? Is it true that INT_MIN $<$ value $<$ INT_MAX?

3 Greedy Algorithms

Reduce the number of states explored to just a single best choice per stage. Never consider alternatives. Prove to yourself that your algorithm achieves the best solution at every stage. If there is a counter example to algorithm, then you may need to consider another programming technique.

3.1 Linear Sweep

Having an order is better than not having an order. Handle states one after the other rather than all at once. Sorting a data set then linear sweeping is never a bad first approach. Recall stabbing end points from activity selection, this is not uncommon in competitive programming.

3.2 Coordinate Compression

For most algorithms, the actual values of the coordinates is irrelevant. We only care about the relative order. If coordinates are up to 1 billion but there are only $n \leq 100,000$ points, then there are only $O(n)$ interesting coordinates.

A solution is to replace each coordinate by its rank among all coordinates so we can preserve the relative order of values while making the maximum coordinate $O(n)$. A simple sort or range tree can solve this issue. We can use *pair* or *tuple* to associate compressed and uncompressed coordinates.

```
#include <bits/stdc ++.h>
using namespace std;
// coordinates -> ( compressed coordinates ).
map <int , int > coordMap;
void compress(vector <int >& values) {
    for (int v : values) coordMap[v] = 0;
    int cId = 0;
    for (auto it = coordMap.begin (); it != coordMap.end (); ++it) {
        it ->second = cId ++;
    }
    for (int &v : values) v = coordMap[v];
}
```

3.3 Binary Search

A monotonic relationship for our purposes is the ability to remove $1/2$ our search space for each check of a value. Say our search space is increasing (sorted) for an array A with values $1, 2, ..., n$ and we're searching for some value x. If $A[1/2] > x$, then we can eliminate all $A[1/2]..A[n]$; the right side of our search space.

Conversely, if $A[1/2] < x$ then we can can eliminate all $1..A[1/2]$; the left side of our search space. This can be extended further to solving a key problem, given a monotone function, find the largest/smallest x such that $f(x)$ is less than/greater than/equal to y.

```
#include <bits/stdc ++.h>
using namespace std;
// Find the smallest X such that f(X) is true;
int binarysearch (function <bool(int)> f) {
    int lo = 0;
    int hi = 100000;
    int bestSoFar = -1;
    // Range [lo , hi];
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (f(mid)) {
            bestSoFar = mid;
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }
    return bestSoFar ;
}
```

3.4 Binary Search alternatives

We can avoid implementing a binary search all together by using the $< algorithm >$ header. Given a sorted array you can use these functions *binary_search*(), *lower_bound*(), *upper_bound*() and *equal_range*().

```
#include <algorithm >
#include <iostream >
using namespace std;
const int N = 100100;
int a[N];
int main () {
    int n;
    cin >> n;
    assert(n <= N);
    for (int i = 0; i < n; i++) cin >> a[i];
    assert( is_sorted (a, a+n));
    int x;
    cin >> x;
    bool found = binary_search (a, a+n, x);
    cout << (found ? "found " : "did not find ") << x;
    int y;
    cin >> y;
    int i = lower_bound (a, a+n, y) - a;
    if (i < n)
        cout << "a[" << i << "] = " << a[i] << " is the first entry
            to compare >= " << y;
    else
        cout << "all elements of a[] compare < " << y;
    int z;
    cin >> z;
    int j = upper_bound (a,a+n,z) - a;
    if (j < n)
        cout << "a[" << j << "] = " << a[j] << " is the first entry
            to compare > " << z;
    else
        cout << "all elements of a[] compare <= " << z;
}
```

3.5 Discrete Binary Search

A Discrete Binary search is particularly useful for converting problems from optimization problems to decision problems. "What is the smallest x for which you can..." => "Given the value x, can you...". Let $f(x)$ be the outcome of the decision problem for a given x, so f is an integer values function with range $\{0, 1\}$.

It is sometimes the case in such problems that increasing x does not make

it any harder for the condition to hold (if it is true for x then it is also true for $x+1$. This f is all zeros up to the first 1, after which it is all ones. This is a monotonic function we can binary search on but particularly, we're trying to find the smallest x such that $f(x) = 1$.

4 Data structures

4.0.1 Vectors

- $O(1)$ access time
- contiguous block of memory
- push_back() adds to end of vector

4.1 Stacks

- push() and pop() operations in $O(1)$
- Last in, first out
- $< stack >$ header

4.2 Queues

- push() and pop() operations in $O(1)$
- First in, first out
- $< queue >$ header

4.3 Sets

- $< set >$ header with $O(logn)$ random access
- Implemented as red-black trees
- Since sets are ordered, you can run lower_bound(x) and upper_bound(x) which returns the next element not less than x
- Multiset in $< multiset >$
- Unordered multiset in $< unordered\_multiset >$

4.4 Maps

- $< map >$ header
- Acts a dictionary with $O(logn)$ random access
- Implemented as a red/black tree of key, value pairs
- Can use lower_bound(x) and upper_bound(x) for the same reason

4.5 Implementation of sets & maps

```
include <iostream >
#include <map >
#include <set >
using namespace std;
set <int > s;
map <int , char > m;
int main () {
    s.insert (2); s.insert (4); s.insert (1);
    m = {{1,'a'}, {4,'c'}, {2,'b'}};
    // Check membership :
    cout << (s.find (2) != s.end ()) << ' ' << (s.find (3) != s.end
        ()) << '\n'; // 1 0
    // NOT binary_search (s.begin (), s.end (), 2), which takes
        linear time
    // Access map:
    cout << m[1] << '\n'; // 'a'
    // WARNING : Access to non - existent data just silently adds it
        , avoid this.
    // cout << m[3] << '\n '; // null character
    // Lower and upper bounds :
    cout << *s. lower_bound (2) << '\n'; // 2
    // NOT * lower_bound (s.begin (), s.end (), 2), which takes
        linear time
    cout << *s. upper_bound (2) << '\n'; // 4
    auto it = m. lower_bound (2);
    cout << it ->first << ' ' << it ->second << '\n'; // 2 b
    // Move around with prev/next or increment / decrement
    cout << prev(it)->first << '\n'; // 1
    cout << (++it)->first << '\n'; //
}
```

4.6 Order Statistic Trees

The mean issue with sets and maps is that they do not track index information. You can not query what the k-th number is or how many numbers are $< x$. We can support a similar operation with order statistic trees.

- find_by_order(x) finds the xth element 0-indexed
- order_of_key(x) outputs the number of elements that are $< x$.
- Both of these are still $O(logn)$.

4.7 OST Implementation

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
using namespace std;

typedef tree<int, null_type, less<int>, rb_tree_tag,
          tree_order_statistics_node_update> ordered_set;

ordered_set myset;
int main() {
    myset.insert(2);
    myset.insert(4);
    myset.insert(1);
    printf("%d\n", *(myset.find_by_order(0))); // 1
    printf("%d\n", (int)myset.order_of_key(3)); // 2
    printf("%d\n", (int)myset.order_of_key(4)); // 2
    printf("%d\n", (int)myset.size()); // 3
}
```

4.8 Heaps

- push() and pop() in $O(log)$
- top() for the top of the heap
- $< queue >$ header
- Heap proper: the value stored in every node is greater than its children
- Shape property: the tree is as close in shape to a complete binary tree as possible

4.9 Union-Find

Used to represent disjoint set of items. For some set of elements, union find supports:

- union(x,y): union the disjoint sets that contain x and y.
- find(x): return a canonical representative for the set that x is in.
- both operations take $O(logn)$ time with improved size heuristic implementation (below)

4.10 Union-Find Implementation

```
int parent[N];
int subtree_size[N];

void init(int n) {
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        subtree_size[i] = 1;
    }
}

int root(int x) {
    // only roots are their own parents
    return parent[x] == x ? x : root(parent[x]);
}

void join(int x, int y) {
    // join roots
    x = root(x); y = root(y);
    // test whether already connected
    if (x == y) return;
    // size heuristic
    // hang smaller subtree under root of larger subtree
    if (subtree_size[x] < subtree_size[y]) {
        parent[x] = y;
        subtree_size[y] += subtree_size[x];
    } else {
        parent[y] = x;
        subtree_size[x] += subtree_size[y];
    }
}
```

4.11 Range trees

Data structures that support operations on a range of values. Can support any operator that is associative (sum, min, max, gcd, lcm, merge). This works by decomposing $[0, n)$ into ranges such that:

- each item belongs to much fewer than n ranges
- any subarray can be decomposed into much fewer than n ranges.

Updates and queries over a range tree is $O(logn)$ with $O(n)$ amount of nodes in the tree

4.12 Range Tree Query

```
const int N = 100100;
// the number of additional nodes created can be as high as the next
    power of two up from N (2^17 = 131,072)
int tree[1<<18];

int n; // actual length of underlying array

// query the sum over [qL, qR) (0-based)
// i is the index in the tree, rooted at 1 so children are 2i and 2i+1
// instead of explicitly storing each node's range of responsibility
    [cL,cR), we calculate it on the way down
// the root node is responsible for [0, n)
int query(int qL, int qR, int i = 1, int cL = 0, int cR = n) {
  // the query range exactly matches this node's range of responsibility
  if (cL == qL && cR == qR)
    return tree[i];
  // we might need to query one or both of the children
  int mid = (cL + cR) / 2;
  int ans = 0;
  // query the part within the left child [cL, mid), if any
  if (qL < mid) ans += query(qL, min(qR, mid), i * 2, cL, mid);
  // query the part within the right child [mid, cR), if any
  if (qR > mid) ans += query(max(qL, mid), qR, i * 2 + 1, mid, cR);
  return ans;
}
```

4.13 Range Tree Update

```
// p is the index in the array (0-based)
// v is the value that the p-th element will be updated to
// i is the index in the tree, rooted at 1 so children are 2i and 2i+1
// instead of explicitly storing each node's range of responsibility
    [cL,cR), we calculate it on the way down
// the root node is responsible for [0, n)
void update(int p, int v, int i = 1, int cL = 0, int cR = n) {
  if (cR - cL == 1) {
    // this node is a leaf, so apply the update
    tree[i] = v;
    return;
  }
  // figure out which child is responsible for the index (p) being
      updated
  int mid = (cL + cR) / 2;
  if (p < mid)
```

```
    update(p, v, i * 2, cL, mid);
  else
    update(p, v, i * 2 + 1, mid, cR);
  // once we have updated the correct child, recalculate our stored
      value.
  tree[i] = tree[i*2] + tree[i*2+1];
}
```

4.14 Range Tree Debugg

```
// print the entire tree to stderr
// instead of explicitly storing each node's range of responsibility
    [cL,cR), we calculate it on the way down
// the root node is responsible for [0, n)
void debug(int i = 1, int cL = 0, int cR = n) {
  // print current node's range of responsibility and value
  cerr << "tree[" << cL << "," << cR << ") = " << tree[i];

  if (cR - cL > 1) { // not a leaf
    // recurse within each child
    int mid = (cL + cR) / 2;
    debug(i * 2, cL, mid);
    debug(i * 2 + 1, mid, cR);
  }
}
```

5 Dynamic Programming

For recursive solutions, time complexity bound is $O(t \times n^r)$ where t is the time spent in the recursive function, n is the number of recursive branches and r is the recursion depth.

For DP, it comes down to carefully determining the number of subproblems and the time taken to solve each subproblem. The idea of DP is to breakdown a large problem into a smaller subset of many subproblems that relate in some way. That is, I should be able to notice the transition of one subproblem to another. Rather than recurse into each subproblem like DNC, we pick and choose which subproblems to calculate and store it for future subproblems to utilise.

5.1 Top Down vs Bottom Up DP

Top down DP takes a mathematical recurrence and translates it directly to code. Subproblems are cached using a caching function; memoisation.

```
int f(int n) {
  // base cases
  if (n == 0 || n == 1) return 1;
  // return the answer from the cache if we already have one
  if (cache[n]) return cache[n];
  // calculate the answer and store it in the cache
  return cache[n] = f(n-1) + f(n-2);
}
```

Bottom-up DP starts at base cases and builds up answers one-by-one.

```
f[0] = 1, f[1] = 1;
for (int i = 2; i <= n; i++) f[i] = f[i-1] + f[i-2];
```

5.2 How to DP

1. Choose some order to do the problem in. How will you build up your solution?
2. Pick a tentative state/subproblem containing the parameters you think are necessary to determine the end result.
3. Test your state/subproblem is sufficient enough by trying to make a recurrence.

If you notice your DP is too slow consider the following:

1. Is everything in the state/subproblem necessary?
2. Is there a better order to solve states/subproblems in?
3. Can a data structure be used to speed up your recurrence?

5.3 2D DP - Integer Knapsack

```
int dp[N+2][S+1];

for (int i = N; i >= 1; --i) {
  // everything from larger i will be available here
  for (int r = 0; r <= S; ++r) {
    // we have declared the array larger, so if i == N, dp[i+1][...]
        will be zero.
```

```
    int m = dp[i+1][r];
    // bounds check so we don't go to a negative array index
    if (r - s[i] >= 0) m = max(m, dp[i+1][r-s[i]] + v[i]);
    dp[i][r] = m;
  }
}
```

5.4 Exponential DP

Sometimes our DP states are very large; 2^n. We thankfully have a work around this by representing the state as a bitset.

5.5 Bitset

A bitset is an integer which represents a set. The ith least significant bit is 1 if the ith element is in the set, and 0 otherwise. For example 01101101 represents the set $0, 1, 2, 3, 4, 6$. We can now perform extremely quick operations to manipulate the set:

- Singleton set: $1 << i$.
- Set complement: $\sim x$.
- Set intersection: $x \ \& \ y$.
- Set union: $x \mid y$.
- Symmetric difference: $x \hat{} y$.
- Membership test: $x \ \& \ (1 << y)$.
- Size of set (C++20): $< bit >$ header using popcount(x).
- Size of set (before C++20): __builtin_popcount(x). Two underscores followed by the text is intended.
- Least significant bit: $x \ \& \ (-x)$.
- Iterate over all sets and subsets:

```
// for all sets
for (int set = 0; set < (1<<n); set++) {
  // for all subsets of that set
  for (int subset = set; subset; subset = (subset-1) & set) {
    // do something with the subset
  }
}
```

5.6 Roof Tiling

You want to tile your roof with n tiles in a straight line, each of which is either black or white. Due to regulations, for every m consecutive tiles on your roof, at least k of them must be black. Given n, m and k ($1 \leq n \leq 60, 1 \leq k \leq m \leq 15, m \leq n$), how many valid tiling are there?

Subproblem/State: Let $f(i, S)$ be the number of ways to have tiled the first i tiles, such that out of the last m tiles, the ones that are black are exactly the ones in our set S.

Recurrence: $f(i, S) = f(i-1, S >> 1) + f(i-1, (S >> 1)|(1 << (m-1)))$, where $f(i-1, (S >> 1)|(1 << (m-1)))$ is taking the union of the set with an extra black tile (adding that tile) with the last $m-1$th tile (we care if this was black or white).

Time Complexity: $O(n2^m)$.

```
// base case
for (int set = 0; set < (1<<m); set++)
  dp[m%2][set] = (popcount(set) >= k);

for (int i = m+1; i <= n; i++) {
  fill(dp[i%2], dp[i%2] + (1<<M), 0);
  for (int set = 0; set < (1<<m); set++)
    if (popcount(set) >= k)
      dp[i%2][set] = dp[(i+1)%2][set>>1]
                   + dp[(i+1)%2][(set>>1)|(1<<(m-1))];
}
// answer is sum over all sets of dp[n%2][set]
```

5.7 DP with Data Structures

Sometimes you have the correct state space, bu the cost of recursion is too high. We can try to speed up our recursive step using data structures. If your states are doing some associative query taking $O(n)$ time, then we can speed this using a range tree for $O(\log n)$.

5.8 Segment Cover

Consider the $n+1$ points on the real line $0, 1, ..., n$. You are given m segments, each with a range $[s_i, e_i]$ and a cost c_i. Output the minimum cost necessary to obtain a subset of the segments which covers all $n+1$ points. First line, n, m. $1 \leq n$, $m \leq 100,000$. The following m lines describe a segment as a triple (s_i, e_i, c_i).

```
#include <bits/stdc++.h> // algorithm, iostream, utility, vector
using namespace std;
typedef long long ll;
const int N = 100100;
const ll INF = (1LL << 61);
int n, m;
vector<pair<int, ll>> segments[N]; // (start, cost)
ll dp[N];
int tree[1<<18]; // range min tree with point update
void update(int p, ll v, int i = 1, int cL = 0, int cR = n);
ll query(int qL, int qR, int i = 1, int cL = 0, int cR = n); // [qL, qR)

int main() {
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int s, e, c;
        cin >> s >> e >> c;
        segments[e].emplace_back(s, c); // preprocess: collate by end
            point
    }

    for (int i = 0; i <= n; i++)
        update(i,INF);

    for (int i = 0; i <= n; i++) {
        dp[i] = INF;
        for (auto seg : segments[i]) {
            ll prevcost = seg.first == 0 ? 0 : query(seg.first-1, i);
            dp[i] = min(dp[i], prevcost + seg.second);
        }
        update(i, dp[i]);
    }
    cout << dp[n] << '\n';
}
```

6 Graph Algorithms

Represented with an adjacency list and occasionally an adjacency matrix.

6.1 Adjacency List

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

int N = 1e6 + 5; // number of vertices in graph
vector<int> edges[N]; // each vertex has a list of connected vertices

void add(int u, int v) {
  edges[u].push_back(v);
  // Warning: If the graph has self-loops, you need to check this.
  if (v != u) {
    edges[v].push_back(u);
  }
}

// iterate over edges from u (since C++11)
for (int v : edges[u]) cout << v << '\n';

// iterate over edges from u (before C++11)
vector<int>::iterator it = edges[u].begin();
for (; it != edges[u].end(); ++it) {
  int v = *it;
  cout << v << '\n';
}

// or just a regular for loop will work too
for (unsigned int i = 0; i < edges[u].size(); i++) {
    cout << edges[u][i] << '\n';
}
```

6.2 BFS

```
vector<int> edges[N];
// dist from start. -1 if unreachable.
int dist[N];
// previous node on a shortest path to the start
// Useful for reconstructing shortest paths
int prev[N];

void bfs(int start) {
    fill(dist, dist+N, -1);
    dist[start] = 0;
    prev[start] = -1;

    queue<int> q;
    q.push(start);
    while (!q.empty()) {
        int c = q.front();
        q.pop();
        for (int nxt : edges[c]) {
            // Push if we have not seen it already.
            if (dist[nxt] == -1) {
                dist[nxt] = dist[c] + 1;
                prev[nxt] = c;
                q.push(nxt);
            }
        }
    }
}
```

6.3 DFS

```
// global arrays are initialised to zero for you
bool seen[N];

void dfs(int u) {
  if (seen[u]) return;
  seen[u] = true;
  for (int v : edges[u]) dfs(v);
}
```

6.4 DFS Trees

DFS alone is usually our traversal of choice to solve many problems. These problems can include: undirected cycle detection, connectivity, flood fill, etc. The main power of DFS is a specific invariant.

Main Invariant: By the time we return from a vertex v in our DFS, we have visited every vertex v can reach that does not require passing through an already visited vertex.

In undirected graphs, if we only consider edges that visit a vertex for the first time, these edges form a tree. All other edges are called "back-edges". **These back edges always go directly upwards to an ancestor in the DFS tree**. It often helps to bring structure to graphs by creating a DFS tree to solve a problem.

6.5 Bridge finding

Consider G an undirected, simple, connected graph. A bridge of G is an edge whose removal disconnects G. Output all bridges of G.

Input: First line, 2 integers V, E the number of vertices and edges respectively. Next E lines, each a pair, $u_i v_i$. Guaranteed $u_i \neq v_i$ and no unordered pair appears twice. $1 \leq V, E, \leq 100,000$.

We claim that **back edges can not be bridges** and **a tree edge is a bridge iff there is no back edge going "past it".** More formally, it is enough to know within each subtree of the DFS tree, what the highest node a back edge in this subtree can reach. We can compute this recursively using a DFS tree.

```
int preorder[N]; // order of pushing to the DFS stack initialise to -1
int T = 0; // counter for preorder sequence
int reach[N]; // reach[u] is the smallest preorder index of any vertex
    reachable from u
vector<pair<int,int>> bridges;

void dfs(int u, int from = -1) {
  preorder[u] = T++;
  reach[u] = preorder[u];

  for (int v : edges[u]) if (v != from) {
    if (preorder[v] == -1) { // u--v is a tree edge
      // v hasn't been seen before, so recurse
      dfs(v, u);
      // if v can't reach anything earlier than itself
      // then u--v is a bridge
      if (reach[v] == preorder[v]) bridges.emplace_back(u,v);
    // anything reachable from v is reachable from u
    reach[u] = min(reach[u], reach[v]);
  }
}
```

6.6 Directed Cycle Detection

We can use the concept of back edges to detect directed cycles. If there is a cycle C and $u \in C$ is the first vertex our DFS visits in the cycle then all vertices in the cycle will be in the subtree of u in the DFS tree. Hence this subtree must have some backedge to u.

We can rephrase this algorithm as checking if any edge visits a vertex we are still recursing from. This means we reach a vertex v that we are still trying ti build the subtree for. So v is an ancestor. We simply just mark the vertex as active in the preorder step and unmark it in the post order step.

```
// the vertices that are still marked active when this returns are the
    ones in the cycle we detected
bool has_cycle(int u) {
  if (seen[u]) return false;
  seen[u] = true;
  active[u] = true;
  for (int v : edges[u]) {
    if (active[v] || has_cycle(v)) return true;
  }
  active[u] = false;
  return false;
}
```

6.7 Tree representation

Trees are undirected, connected, have unique simple paths between any two vertices, $E = V - 1$, no cycles and where any edge removal disconnects the graph.

```
const int N = 1e6 + 5;
// We need the list of edges to construct our representation. But we
    don't use it afterwards.
vector<int> edges[N];
int par[N]; // Parent. -1 for the root.
vector<int> children[N]; // Your children in the tree.
int size[N]; // As an example: size of each subtree.

void constructTree(int c, int cPar = -1) {
    par[c] = cPar;
    size[c] = 1;
    for (int nxt : edges[c]) {
        if (nxt == par[c]) continue;
        constructTree(nxt, c); children[c].push_back(nxt);
        size[c] += size[nxt];
    }
}
```

With the tree representation of a graph, every node has a parent (except the root) and a list of children. We can do a lot by recursing through the children array. A linear sweep on the array is just a DFS, a DP on the array is to DP on the tree and we can perform range tree queries for paths over a tree.

6.8 Shortest Distance on a tree - LCA

Given a labeled rooted tree T, and Q queries of form, "What is the vertex furthest away from the root in the tree that is an ancestor of vertices labeled u and v?".

Input: A rooted tree $T(1 \leq |T| \leq 1,000,000)$, as well as $Q(1 \leq Q \leq 1,000,000)$ pairs of integers u and v.

Algorithm Instead of walking up single edges, in a tree representation we can use our precomputed parent$[u][k]$ to greedily continue moving up by the largest power of 2 possible until we're at the desired vertex. We're trying to find the **Lowest common ancestor (LCA)** in this step.

Fist, move both u and v to the same depth then binary search for the maximum amount you can jump up without reaching the same vertex. The parent of that vertex is the LCA. The implementation sets a search range that is jumping up in decreasing powers of 2 order and rejecting any jumps that results in u and v being at the same vertex.

6.9 LCA - Preprocess

```
// parent[u][k] is the 2^k-th parent of u
void preprocess() {
  for (int i = 0; i < n; i++) {
    // assume parent[i][0] (the parent of i) is already filled in
    for (int j = 1; (1<<j) < n; j++) parent[i][j] = -1;
  }

  // fill in the parent for each power of two up to n
  for (int j = 1; (1<<j) < n; j++) {
    for (int i = 0; i < n; i++) {
      if (parent[i][j-1] != -1) {
        // the 2^j-th parent is the 2^(j-1)-th parent of the 2^(j-1)-th
            parent
        parent[i][j] = parent[parent[i][j-1]][j-1];
      }
    }
  }
}
```

6.10 LCA - Querying

```
int lca (int u, int v) {
  // make sure u is deeper than v
  if (depth[u] < depth[v]) swap(u,v);

  // log2s[i] holds the largest k such that 2^k <= i
  // precompute by DP: log2s[i] = log2s[i/2] + 1
  for (int i = log2s[depth[u]]; i >= 0; i--) {
    // repeatedly raise u by the largest possible power of two until it
        is the same depth as v
    if (depth[u] - (1<<i) >= depth[v]) u = parent[u][i];
  }

  if (u == v) return u;

  for (i = log2s[depth[u]]; i >= 0; i--)
    if (parent[u][i] != -1 && parent[u][i] != parent[v][i]) {
      // raise u and v as much as possible without having them coincide
      // this is important because we're looking for the lowest common
          ancestor, not just any
      u = parent[u][i];
      v = parent[v][i];
    }

  // u and v are now distinct but have the same parent, and that parent
      is the LCA
  return parent[u][0];
}
```

We know what the paths between u and v look like, it's $u->$ lca followed by lca $->v$. We also need the lengths of certain ranges, similar to a range tree. Sum has an inverse and so we can use a cumulative sum data structure instead.

Store dist$(root, u)$ for all u then dist(u, v) =dist$(root, u)$+dist$(root, v)-2\times$dist$(root, lca)$.

6.11 LCA Algorithm

```
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 100000, LOGN = 18;
struct edge { int nd; long long d; };
int parent[MAXN+5][LOGN];
long long distToRoot[MAXN+5];
vector<edge> children[MAXN+5];
// Code to set up LCA and tree representation
void construct_tree(int c, int cPar = -1);
int lca(int a, int b);

void calc_dists_to_root(int c) {
    for (auto edg : children[c]) {
        distToRoot[edg.nd] = distToRoot[c] + edg.d;
        calc_dists_to_root(edg.nd);
    }
}

long long find_tree_dist(int a, int b) {
    int cLca = lca(a, b);
    return distToRoot[a] + distToRoot[b] - 2 * distToRoot[cLca];
}
```

The time complexity of this algorithm is $O(n \log n)$ time preprocessing and $O(\log n)$ time per query. You must do jumps to the power of 2.

6.12 Directed Acyclic Graphs (DAG)

1. Every DAG has a minimal vertex, one with no incoming edges.
2. Every DAG can be linearly ordered. There must be some ordered of vertices such that edges only go from $v_i -> v_j$ where $i < j$. This is also known as a topological sort.

6.13 Topological sort

An ordering of vertices that has the property that if some vertex u has a directed edge pointing to another vertex v, then v comes after u in the ordering. If the graph has a cycle, then there does not exist a valid topological sort.

Due to the DFS invariant, for an acyclic graph every vertex after v in the sort is returned from before v. A vertex is only added after it's children have been visited (and thus added).

```
// if the edges are in ASCENDING order of node number,
// this produces the lexicographically GREATEST ordering

void dfs(int u, vector<int>& postorder) {
  if (seen[u]) return;
  seen[u] = true;
  for (int v : edges[u]) dfs(v, postorder);
  postorder.push_back(u);
}

vector<int> topsort() {
  vector<int> res;
  for (int i = 0; i < n; i++) dfs(i, res);
  reverse(res.begin(), res.end()); // #include <algorithm>
  return res;
}
```

6.14 Strongly Connected Components (SCC)

An SCC is a maximal subset of vertices of a directed graph such that every vertex in the subset can reach every other vertex in that component. Condensing every SCC to a single vertex (condensation graph) results in a directed acyclic graph. We have two SCC algorithms.

6.15 Targan's Algorithm

```
    // we will number the vertices in the order we see them in the DFS
int dfs_index[MAX_VERTICES];
// for each vertex, store the smallest number of any vertex we see
// in its DFS subtree
int lowlink[MAX_VERTICES];

// explicit stack
stack<int> s; // #include <stack>
bool in_stack[MAX_VERTICES];

// arbitrarily number the SCCs and remember which one things are in
int scc_counter;
int which_scc[MAX_VERTICES];

void connect(int v) {
  // a static variable doesn't get reset between function calls
  static int i = 1;
  // set the number for this vertex
  // the smallest numbered thing it can see so far is itself
  lowlink[v] = dfs_index[v] = i++;
```

```
  s.push(v);
  in_stack[v] = true;

  for (auto w : edges[v]) { // for each edge v -> w
    if (!dfs_index[w]) { // w hasn't been visited yet
      connect(w);
      // if w can see something, v can too
      lowlink[v] = min(lowlink[v], lowlink[w]);
    }
    else if (in_stack[w]) {
      // w is already in the stack, but we can see it
      // this means v and w are in the same SCC
      lowlink[v] = min(lowlink[v], dfs_index[w]);
    }
  }
  // v is the root of an SCC
  if (lowlink[v] == dfs_index[v]) {
    ++scc_counter;
    int w;
    do {
      w = s.top(); s.pop();
      in_stack[w] = false;
      which_scc[w] = scc_counter;
    } while (w != v);
  }
}
```

6.16 Kosaraju's Algorithm

```
void dfs(int u) {
   if (seen[u])
      return;
   seen[u] = true;
   for (int v : edges[u])
      dfs(v);
   postorder[p++] = u;
}

void dfs_r(int u, int mark) {
   if (seen_r[u])
      return;
   seen_r[u] = true;
   scc[u] = mark;
   for (int v : edges_r[u])
      dfs_r(v, mark);
}

int compute_sccs() {
```

```
  int sccs = 0;
  for (int i = 1; i <= n; i++)
    if (!seen[i])
      dfs(i);
  for (int i = p - 1; i >= 0; i--) {
    int u = postorder[i];
    if (!seen_r[u]) // ignore visited vertices
      dfs_r(u, sccs++);
  }
  return sccs;
}
```

6.17 Minimum Spanning Trees (MST)

A spanning tree for some graph G is a subgraph of G that is a tree, and also connects all of the vertices of G. A MST is a spanning tree with minimum sum of edge weights.

To construct a minimum spanning tree of some graph G, we maintain a set of spanning forests, initially composed of just the vertices of the graph and no edges, and we keep adding edges until we have a spanning tree. If we add $|V|-1$ edges and avoid constructing cycles, we'll have a spanning tree.

We restrict ourselves to only edges that cross components that we haven't connected yet. If e has a minimum weight of edges that connects components we haven't connected yet, then there is a spanning tree containing e. We have Krukal's Algorithm that does this in $O(|E| \log |E|)$ time.

```
struct edge {
  int u, v, w;
};
bool operator< (const edge& a, const edge& b) {
  return a.w < b.w;
}

edge edges[N];
int p[N];
int root (int u); // union-find root with path compression
void join (int u, int v); // union-find join with size heuristic

int mst() {
  sort(edges, edges+m); // sort by increasing weight
  int total_weight = 0;
  for (int i = 0; i < m; i++) {
    edge& e = edges[i];
    // if the endpoints are in different trees, join them
    if (root(e.u) != root(e.v)) {
```

```
      total_weight += e.w;
      join(e.u, e.v);
    }
  }
  return total_weight;
}
```

7 Shortest Paths

Given a weighted directed graph G with two specific vertices s and t, we want to find the shortest path that goes between s and t on the graph. Generally, algorithms which solve the shortest path problem also solve the single source shortest path problem, which computes shortest paths from a single source vertex to every other vertex.

7.1 BFS

BFS can solve shortest paths in unweighted graphs or graphs with edge weight of 1.

7.2 Single Source Shortest Paths

Most single source shortest paths algorithms rely on the basic idea of building shortest paths iteratively. At any point, we keep what we think is the shortest path to each vertex and we update this by "relaxing" edges.

Relax(u, v): if the currently found shortest path from our source s to a vertex v could be improved by using the edge (u, v), update it. For non-negative edge weights, we can get away with only relaxing vertices for which we know the optimal distance. But this becomes tricker with negative edge weights.

If we keep track for each v of its most recently relaxed incoming edge, we can find the actual path from the source to v. This is because, for each v we know the vertex we would've come from if we followed the shortest path from the source. We can work backwards from v to the source to find the shortest path from the source to v. So keep relaxing edges until we can't anymore, then we'll have a shortest path.

7.3 Djikstra's Algorithm

For non-negative edge weights, visit each vertex u starting from the source s. Whenever we visit the vertex u, we relax all of the edges coming out of u. We always process the next unprocessed vertex with the smallest distance from the source. Djikstra's uses a priority queue keyed on each vertex's current known shortest distance from the source. This runs in $O(E \log V)$ using a priority queue.

7.4 Djikstra's Implementation

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

typedef pair<int, int> edge; // (distance, vertex)
const int N = 100100;
vector<edge> edges[N];
int dist[N];
bool seen[N];
priority_queue<edge, vector<edge>, greater<edge>> pq;

void dijkstra (int s) {
  fill(seen,seen+N,false);
  pq.push(edge(0, s)); // distance to s itself is zero
  while (!pq.empty()) {
    // choose (d, v) so that d is minimal
    // i.e. the closest unvisited vertex
    edge cur = pq.top();
    pq.pop();
    int v = cur.second, d = cur.first;
    if (seen[v]) continue;

    dist[v] = d;
    seen[v] = true;

    // relax all edges from v
    for (edge nxt : edges[v]) {
      int u = nxt.second, weight = nxt.first;
      if (!seen[u])
        pq.push(edge(d + weight, u));
    }
  }
}
```

7.5 Bellman-Ford Algorithm

If a graph has no negative cycles, then all shortest paths from u have length $\leq |V|-1$. Conversely, a negative cycle implies there is a shortest path of length $|V|$ better than any path of length $|V|-1$. So we should instead build up shortest paths by number of edges, not just from u outwards.

Bellman-Ford involves trying to relax every edge of the graph $|v|-1$ times and update our tentative shortest paths each time. Because every shortest path has at most $|V|-1$ edges, if after all these global relaxations, relaxations can still be made, then ere exists a negative cycle.

Subproblem: $f(v, k)$, the shortest $s->v$ path using up to k edges.

Recurrence: $f(v, k) = \min\{f(v, k-1), \min_{u:e=(u,v)\in E}(f(u, k-1) + w_e)\}$

Base cases: $f(v, 0) = 0$ for $v = s$ or ∞ otherwise.

Time complexity: $O(VE)$.

7.6 Bellman-Ford implementation

```
struct edge {
  int u, v, w; // u -> v of weight w
  edge(int _u, int _v, int _w) : u(_u), v(_v), w(_w) {}
};
vector<int> dist(n);
vector<edge> edges;

// global relaxation: try to relax every edge in the graph
// returns whether any distance was updated
bool relax() {
  bool relaxed = false;
  for (edge e : edges) {
    // we don't want to relax an edge from an unreachable vertex
    if (dist[e.u] != INF && dist[e.v] > dist[e.u] + e.w) {
      relaxed = true;
      dist[e.v] = dist[e.u] + e.w;
    }
  }
  return relaxed;
}

// Puts distances from source vertex 0 in dist
// Returns true if there is a negative cycle, false otherwise.
// NOTE: You can't trust the dist array if this function returns True.
bool bellman_ford() {
  fill(dist.begin(), dist.end(), INF);
  dist[0] = 0;
  // V-1 global relaxations
  // if no updates are made in an entire round, we can early exit
  // SPFA optimises this further
  for (int i = 0; i < n - 1; i++) if (!relax()) break;
  // there is a negative cycle iff any edge can be relaxed further
  // therefore try a Vth global relaxation
  // true if any changes made, false otherwise
  // can micro-optimise by early exiting when the first change is made
  return relax();
}
```

7.7 Floyd-Warsall Algorithm

Finds all pairs of shortest paths between all vertices in a graph using dynamic programming in $O(V^3)$ time.

Subproblem: Let $f(u, v, i)$ be the length of the shortest path between u and v using only the first i vertices as intermediate vertices. The key is to build this up for increasing values of i.

Base case: Then $f(u, u, 0) = 0$ for all vertices u , $f(u, v, 0) = w_e$ if there is an edge fom u to v and ∞ otherwise.

Recurrence: $f(u, v, i) = \min(f(u, v, i-1), f(u, i, i-1) + f(i, v, i-1))$.

7.8 Floyd-Warshall Implementation

```
// the distance between everything is infinity
for (int u = 0; u < n; ++u)
  for (int v = 0; v < n; ++v)
    dist[u][v] = INF;

// update the distances for every directed edge
for (edge e : edges)
  // each edge u -> v with weight w
  dist[e.u][e.v] = e.w;

// every vertex can reach itself
for (int u = 0; u < n; ++u)
  dist[u][u] = 0;

for (int i = 0; i < n; i++)
  for (int u = 0; u < n; u++)
    for (int v = 0; v < n; v++)
      // dist[u][v] is the length of the shortest path from u to v using
          only 0 to i as intermediate vertices
      // now that we're allowed to also use i, the only new path that
          could be shorter is u -> i -> v
      dist[u][v] = min(dist[u][v], dist[u][i] + dist[i][v]);
```

7.9 Reconstructing Paths

As well as keeping track of a distance table, any time the improved path (via i) is used, note that the next thing on the path from u to v is going to be the next thing on the path from u to i. We should already know this since we're keeping track of it. When initializing the table with the edges in the graph, don't forget to set v as next on the path from u to v for each edge $u- > v$.

8 Network Flow

A flow network, is a directed graph where each edge has a capacity that flow can be pushed through. The integrality theorem states that if all the edges in the graph have integer capacities, then there exists a maximum flow where the flow through ever edge is an integer.

8.1 Minimum cut

Given a graph with a source, sink and edge weights, find the set of edges with the smallest sum of weights that need to be removed to disconnect the source from the sink. In a directed graph, define a $s - t$ cut to be a partition of the vertices into two sets, on containing the source s, the other containing the sink t.

The capacity of such a cut is defined to be the sum of capacities of edges going from the source partition to the sink partition. The min cut is the minimum capacity of all $s - t$ cuts.

It turns out that the maximum-flow and minimum-cut problems are equivalent. The max-flow/min-cut theorem states the value of the minimum cut, if we set edge weights to be capacities in a flow graph, is the same as the value of the maximum flow.

8.2 Residual graph

The residual graph of a flow graph has all the same edges as the original graph, as well as a new edge going in the other direction for each original edge, with capacity zero. Residual edges do the opposite of flowed edges in the original graph.

Residual graph: A path from s to t as a path that travels only on edges with capacity strictly greater than current flow. If we can find an augmenting path, then we increase the flow in our graph by decreasing the capacities of the edges they pass through.

Whenever we flow along an edge, we should also do the opposite on the residual edge. So if we increase flow by 1, we should either decrease flow by 1 on the residual or, equivalently, increase capacity by 1.

If we can't find an augmenting path in the graph, then we have our maximum flow. Furthermore, this implies in our residual graph, there exists no path from s to t, so we have discovered a cut. This cut must be minimum because if there were a smaller cut, it would be impossible to push as much flow as we have already pushed.

8.3 Graph Representation of flow

Adjacency list that stores edges, adjacency matrix that stores flow amount.

```
struct FlowNetwork {
    int n;
    vector<vector<ll>> adjMat;
    vector<vector<int>> adjList;

    FlowNetwork (int _n): n(_n) {
        // adjacency matrix is zero-initialised
        adjMat.resize(n);
        for (int i = 0; i < n; i++)
            adjMat[i].resize(n);
        adjList.resize(n);
    }

    void add_edge (int u, int v, ll c) {
        // add to any existing edge without overwriting
        adjMat[u][v] += c;
        adjList[u].push_back(v);
        adjList[v].push_back(u);
    }

    void flow_edge (int u, int v, ll c) {
        adjMat[u][v] -= c;
        adjMat[v][u] += c;
    }
};
```

8.4 Ford-Fulkerson

Keep finding augmenting paths until there are none left. Each graph search takes $O(E)$ time to send a single unit of flow. Therefore $O(Ef)$ time complexity.

```
// add to FlowNetwork struct
bool augment(int u, int t, ll f, vector<bool>& seen) {
  if (u == t) return true;
  if (seen[u]) return false;
  seen[u] = true;
  for (int v : adjList[u])
    if (adjMat[u][v] >= f && augment(v,t,f,seen)) {
      flow_edge(u,v,f);
      return true;
    }
  return false;
}

ll ford_fulkerson(int s, int t) {
  vector<bool> seen(n, false);
  ll res = 0;
  while (true) {
    fill(seen.begin(),seen.end(),false);
    if (augment(s,t,1,seen))
      res++;
    else
      break;
  }
  return res;
}
```

8.4.1 Edmonds-Karp algorithm

Take the shortest augmenting path instead of any augmenting path. Find these shortest paths using a BFS. Time complexity is $O(\min(VE^2, Ef))$.

```
// add to FlowNetwork struct, replacing Ford-Fulkerson code
ll augment(int s, int t) {
    queue<int> q; // BFS for path with fewest edges
    q.push(s);
    // pred[v] = vertex from which v was found in BFS
    vector<int> pred(n,-1);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v : adjList[u]) {
            if (adjMat[u][v] <= 0) continue;
            if (pred[v] == -1) {
                pred[v] = u;
                q.push(v);
            }
        }
    }
    if (pred[t] == -1) return 0; // can't reach the sink
    ll res = INF; // must be bigger than the largest edge capacity
    // walk over the path backwards to find the minimum edge
    for (int v = t; v != s; v = pred[v])
        res = min(res, adjMat[pred[v]][v]);
    // do it again to adjust capacities
    for (int v = t; v != s; v = pred[v])
        flow_edge(pred[v],v,res);
    return res;
}

ll edmonds_karp(int s, int t) {
    ll res = 0;
    while (ll x = augment(s,t))
        res += x;
    return res;
}
```

8.5 Dinic's Algorithm

Same as Edmonds Karp, we augment along the shortest augmenting path however, we work in phases. In each phase, we augment until the shortest distance from source to sink has increased. Hence there are $O(V)$ phases.

Level graph: the graph restricted to only edges (u, v) where $d[u] + 1 = d[v]$. The level graph in Dinics only needs to be rebuilt at the start of each phase. This allows us to find each augmenting path in $O(V)$ amortized. Each phase has $O(E)$ augmenting paths and hence we get $O(V^2E)$ overall.

Claim: Once there is no flow in the level graph, the distance from the source to sink has increased. In the level graph, any path we find is a shortest augmenting path. So we can use a DFS to find these. Say we are at node u, if we DFS to child c of node u and there is no augmenting paths from c to the sink, then child c is useless and we delete child c from node u.

So our DFS will always find an augmenting path by trying the very first child of each node. Amortized out, this means we get $O(V)$ per augmenting path, with an extra overall cost of $O(E)$ for the times we have to delete a child. Hence, each pase is $O(EV)$, there are at most $O(E)$ augmenting paths and each is found in $O(V)$. Time complexity is $O(\min(V^2E, Ef))$.

8.6 Dinic's Algorithm Implementation

```
struct FlowNetwork {
    int n;
    vector<vector<ll>> adjMat, adjList;
    // level[v] stores dist from s to v
    // uptochild[v] stores first non-useless child.
    vector<int> level, uptochild;

    FlowNetwork (int _n): n(_n) {
        // adjacency matrix is zero-initialised
        adjMat.resize(n);
        for (int i = 0; i < n; i++) adjMat[i].resize(n);
        adjList.resize(n);
        level.resize(n);
        uptochild.resize(n);
    }

    void add_edge (int u, int v, ll c) {
        // add to any existing edge without overwriting
        adjMat[u][v] += c;
        adjList[u].push_back(v);
        adjList[v].push_back(u);
    }
```

```
void flow_edge (int u, int v, ll c) {
    adjMat[u][v] -= c;
    adjMat[v][u] += c;
}

// constructs the level graph and returns whether the sink is still
    reachable
bool bfs(int s, int t) {
    fill(level.begin(), level.end(), -1);
    queue<int> q;
    q.push(s);
    level[s] = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        uptochild[u] = 0; // reset uptochild
        for (int v : adjList[u])
            if (adjMat[u][v] > 0) {
                if (level[v] != -1) // already seen
                    continue;
                level[v] = level[u] + 1;
                q.push(v);
            }
    }
    return level[t] != -1;
}

// finds an augmenting path with up to f flow.
ll augment(int u, int t, ll f) {
    if (u == t) return f; // base case.
    // note the reference here! we increment uptochild[u], i.e. walk
        through u's neighbours
    // until we find a child that we can flow through
    for (int &i = uptochild[u]; i < adjList[u].size(); i++) {
        int v = adjList[u][i];
        if (adjMat[u][v] > 0) {
            // ignore edges not in the BFS tree.
            if (level[v] != level[u] + 1) continue;
            // revised flow is constrained also by this edge
            ll rf = augment(v,t,min(f, adjMat[u][v]));
            // found a child we can flow through!
            if (rf > 0) {
                flow_edge(u,v,rf);
                return rf;
            }
        }
    }
    level[u] = -1;
    return 0;
}
```

```
    ll dinic(int s, int t) {
        ll res = 0;
        while (bfs(s,t))
            for (ll x = augment(s,t,INF); x; x = augment(s,t,INF))
                res += x;
        return res;
    }
};
```

8.7 Bipartite Matching

The maximum bipartite matching problem is given a bipartite graph (one where the vertices can be partitioned into two sets with no vertices in the same set have an edge between them), is to choose the largest possible set of edges such that no one vertex is incident to more than one chosen edge.

There is a clear flow formulation for this problem, modify the bipartite graph by making a directed edge from the first set to the second set with capacity 1. Attach an edge of capacity 1 from a super source s to every vertex in the first set and attach an edge of capacity 1 from every vertex in the second set to t.

A great observation is that Dinics runs in $0(E\sqrt{V})$ on unit graphs such as a bipartite graph.

8.8 Extracting the min-cut

The extract the edges in a minimum cut, we use the fact that all of them must be completely saturated. We do a graph traversal starting from s and only traverse edges that have positive remaining capacity and record which vertices we visit. The edges which have visited a visited vertex on one end and an unvisited vertex on the other will form the minimum cut.

```
void check_reach(int u, vector<bool>& seen) {
  if (seen[u]) return;
  seen[u] = true;
  for (int v : adjList[u])
    if (adjMat[u][v] > 0)
      check_reach(v,seen);
}
```

```
vector<pair<int,int>> min_cut(int s, int t) {
  ll value = dinic(s,t);

  vector<bool> seen(n,false);
  check_reach(s,seen);

  vector<pair<int,int>> ans;
  for (int u = 0; u < n; u++) {
    if (!seen[u])
      continue;
    for (int v : adjList[u])
      if (!seen[v] && !is_virtual[u][v]) // need to record this in
          add_edge()
        ans.emplace_back(u,v);
  }
  return ans;
}
```

8.9 Vertex capacities

A graph has vertex capacities if there are also capacity restrictions on how much flow can go through a vertex. We solve this by splitting a vertex into two vertices, an "in" vertex and an "out" vertex. For some vertex u with capacity c_u, we add an edge from in_u to out_u with capacity c_U. Incoming edges go to in_u and outgoing edges leave from out_u with their original capacities. Useful for flow problems with restrictions.

8.10 Other flow techniques

- **Multiple source and sinks:** Sometimes it makes sense for a problem to be modeled with multiple sources and sinks. Just make all the sources into regular nodes and connect them with infinite edges to a supersource, the actual source of the flow graph.
- **Undirected Graphs:** Sometimes you want want to find flow or min cut in an undirected graph. Just duplicate each edge, one going forwards and another going backwards.
- **Maximum edge-disjoint paths in a graph:** To find the maximum number of edge disjoint paths (no two paths use the same edge) from s to t, make a flow graph where all edges have capacity 1.The maximum flow of this graph will give the answer.
- **Minimum vertex cover in a bipartite graph:** A vertex cover in a graph is a set of vertices which touches at least one endpoint of every edge. The size of the maximum matching is equal to the number of vertices in a minimum vertex cover. Actually this is just a min-cut max-flow. Using

our earlier construction of a flow network for bipartite graphs, the min cut corresponds to a max vertex cover.