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).

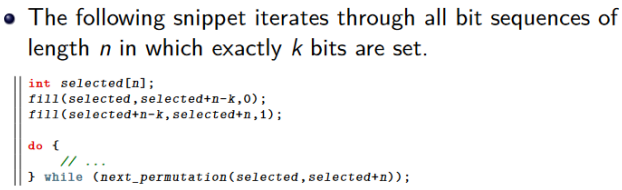

Figure 1: Next permutation

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'; //
}
```

5 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 $O(logn)$.

5.1 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
}
```

5.2 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

5.3 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)

5.4 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];
    }
}
```

6 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

6.1 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;
}
```

6.2 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];
}
```

6.3 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);
  }
}
```

7 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.

7.1 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];
```

7.2 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?

7.3 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;
  }
}
```

7.4 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.

7.5 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
  }
}
```

7.6 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]
```

7.7 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 $O(\log n)$.

7.8 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';
}
```