Interesting 1:

<algorithm>

fill(a, a+n, 0)

<numeric>

partial\_sum(arr1, arr1+n, arr2)

iota(a, a+n, 1)

Interesting 2:

for (auto e : map){

int key = e.first;

int value = e.second;

}

Interesting 3:

Global arrays can have much larger size than local arrays. I can declare integer arrays with more than 10^8 integers on my laptop.

Interesting 4:

In some problems regarding coordinates, you can use maps to compress the coordinates. For example, if n <= 10^5 but x,y <= 10^9, you can sort all of them to obtain a relative order of the xy coordinates and magically x,y <= 10^5. A problem starting with the letter A in hkoj can be solved using this technique.

Interesting 5:

Don't write mid=(l+r)/2 because it may overflow! Always write mid=l+(r-l)/2.

Interesting 6:

If you forgot how to write binary search, you can always look up Wikipedia!

Interesting 7:

If you just flip everything when writing binary search for non-increasing sequences, remember to change floor() to ceil()!

I usually write int mid = l+(r-l)/2+((r-l)%2);

Interesting 8:

If you are dfs-ing a tree, you can do this instead of maintaining a visited array.

void dfs(int c, int p){

for (int i = 0; i < a[c].size(); i++){

if (a[c][i] != p) dfs(a[c][i], c);

}

}

Interesting 8.5:

void dfs(int c, int p){

for (auto e : a[c]) if (e != p) dfs(e, c);

}

Interesting 9:

Meet-in-the-middle technique

When the input is small, but not small enough to directly brute force, this technique may be used.

For example, let’s say brute force takes O(n!). You may split the input into halves and brute force both of them, then combine them. For combining results, commonly you may store all the results in a map or array to then combine in O(n^2) time, or you may sort both of them and combine them cleverly to get O(n log n) combining time. Using this technique, O(n!) and O(2^n) can be brought down to O((n/2)! + combine) and O(sqrt(2^n) + combine), which is very fast.

<https://www.geeksforgeeks.org/meet-in-the-middle/>

Interesting 10:

!!Advanced alert!!

Binary lifting is a technique that allows us to find the kth ancestor of a node in a tree in O(log k) time. Maintain int anc[n][log n]. anc[i][j] tells us the 2^j th ancestor of node i. (if j=0 then it’s the parent)

int anc[100000][20]

void dfs(int c, int p){

for (int i = 1; i < 20; i++){

if (anc[c][i-1] == -1) break;

anc[c][i] = anc[anc[c][i-1]][i-1]

}

for (auto e : adj[c]){

if (e == p) continue;

anc[e][0] = c;

dfs(e, c);

}

}

Interesting 11:

!!Advanced alert!!

Applications of binary lifting:

If it wasn’t obvious, you can convert k into a binary number to get the kth ancestor of a node.

Binary lifting can also help use binary search the path between a node and the root, or the path between a node and its kth ancestor, which is extremely useful.

int cur = node;

for (int i = 19; i >= 0; i--){

if (anc[cur][i] != -1 && (condition)) cur = anc[cur][i];

}

Interesting 13:

Euler tour

The euler tour of a tree is like the preorder traversal combined with the postorder traversal. You push the elements to a global vector before the for loop part in the dfs and after the for loop part. Its like drawing a dfs traversal of the tree in one stroke. The cool part is that with the Euler tour, you can answer queries of the form is node u an ancestor of node v? in O(1) time.

Let's say you have an Euler tour 1 2 3 3 4 4 2 5 6 6 5 1. If you change the first occurrences of each number to '(' and the second occurrences of each number to ')', you observe that the sequence becomes ( ( ( ) ( ) ) ( ( ) ) ), and that the 'brackets' of a node completely envelopes the 'brackets' of its children and grandchildren, etc. Therefore, if you postprocess the euler tour after the dfs such that you have the index of the first occurrences in[n], and index of the second occurrences out[n], if in[u] < in[v] and out[v] < out[u], you know that u is an ancestor of v in O(1) time.

Interesting 14:

Using Euler tour and segment tree to query LCA

In the last interesting, I talked about my version of Euler tour for finding if one node is an ancestor of another. Now, I will talk about the more formal Euler tour. It is actually more like preorder, inorder, postorder all combined. The pseudocode for finding the Euler tour looks like this:

//Push cur into e

for each adj vertex{

If not visited{

If not the first adj vertex{

//Push cur into e

}

Dfs(adj vertex)

}

}

//Push cur into e

Now, you observe, if you have 2 nodes u and v in the euler tour, let's say ......u...u...u......v...v...... The node with smallest depth between the part u......v is the lca of u and v. Therefore, if you precalculate depth[n], and also postprocess the Euler tour such that you know the index for first and last occurrences of each node, let's say first[n] and last[n], you can simply use a segment tree to query for the node with minimum depth (range minimum query), and return the answer.

1. Check if u is before v or otherwise

2. Query(last[u]+1, first[v]-1)

3. Find the corresponding vertex label

4. Done!!

Interesting 15:

Segment tree concept

Imagine you have an array of size 8 and you need to do some operation f(a[0]~a[7]). Then, you observe that divide and conquer can be used. so f(a[0]~a[7]) can be calculated using f(a[0]~a[3]) and f(a[4]~a[7]). Therefore, you can draw a diagram. On the first level there is 1 node f(a[0]~a[7]), it splits into 2 nodes on the second level, and those 2 splits into 4, and finally there are 8 nodes on the fourth level, f(a[0]~a[0]) ~ f(a[7]~a[7]).

Let's analyze the time complexity. On the first level there is 1 node. The second level has 2 nodes, and so on. Therefore, the number of nodes are 1 + 2 + 4 + ... + n. From your math skills, you should know that that is about 2n (if n is 2^k). So the time complexity is O(n), or O(n \* mergetime).

Now let's say the problem is online and have updates of the form change the ith element to value v. You observe that in the diagram, only the 'branch' containing a[i] needs to be updated. This means that you only need to update log n nodes. Once you update the node f(a[i]~a[i]), you can update the upper nodes by merging the updated node and another node which didn't require an update. The time complexity is O(log n \* mergetime).

Interesting 16:

Segment tree concept 2

First, I'll show how to query from l to r. Then, I'll prove why it's O(log n). Start with the root node f(a[0]~a[7]). Call query(node(a[0]~a[7])). If the range of the current node is completely in the queried range (l, r), just return that node. Else, return f(query(left child), query(right child)).

This may seem like O(n) because each time you call the 2 children. But if you look closely, at each level, at most 4 nodes are called. Let's start from the root. The range of the root is not inside the queried range, so you chop the range if the root into 2 and query them. Now, for the left half and right half, they may also be not completely in range, so there will at most be 4 calls at the third level. Now, observe that the right half of the left half must always lie completely inside the queries range. Same goes for the right half of the left half. Therefore, in the next level, at most, only the 2 children of the leftmost call and the rightmost call are called. So, the next level also has at most 4 nodes queried. Since there are log n levels, the time complexity is O(log n).

Interesting 17:

!!Advanced alert!!

segment tree lazy propagation

Now you can do point update and range query in O(log n) time. But what if you what to range update? (for example, add delta to all elements in range l to r) You can use this lazy propagation technique. It is actually surprisingly easy to understand. You keep a lazy value for each node, and propagate it downwards when needed. So, instead of propagating it all in a update, which results in O(n) update, you store it in lazy and propagate when you happen to traverse it in a update or a query function. If you understand the previous interesting and why query is O(log n), it is pretty clear that the update is also O(log n). It is also simple to write, and you only need to add 2 small functions.

Pseudocode (range sum query, range update add x to all elements in range)

void upd(int id, int l, int r, int x){

lazy[id] += x;

s[id] += (r-l+1) \* x;

}

void prop(int id, int l, int r){

mid = …

upd(id \* 2, l, mid, lazy[id]);

upd(id \* 2 + 1, mid + 1, r, lazy[id]);

lazy[id] = 0;

}

void update(int id, int l, int r, int L, int R, int x){

if (out of range) return;

if (completely in range){

upd(id, l, r, x);

return;

}

prop(id, l, r);

int mid = …;

update(left child);

update(right child);

s[id] = s[id \* 2] + s[id \* 2 + 1]);

}

int query(int id, int l, int r, int L, int R){

if (out of range) return 0;

if (completely in range) return s[id];

prop(id, l, r);

int mid = …;

return query(left child) + query(right child);

}

Interesting 20:

tree DP #2

The diameter of a graph is the length of the longest shortest path between 2 nodes. Since any 2 nodes in a tree only has 1 simple path that connects them, the diameter of a tree is the longest distance between 2 nodes. How do you find the diameter in a tree?

if node i is part of the diameter, there can be 2 possibilities: node i is the node with lowest depth, and the rest of the diameter goes into 2 of its subtrees OR node i is not the node with lowest depth, so one side of the diameter goes into 1 of its subtrees and the other side goes to its ancestors.

Let’s denote dp1[i] as the maximum length of a path that starts from node i and ends at one of its subtrees and dp2[i] as the maximum length of a path that starts from a subtree of node i, goes to node i, then ends at another one of its subtrees.

dp1[i] = 1 + max(dp1[c])

dp2[i] = 1 + max(dp1[c]) + second\_max(dp1[c])

where c is the child of i

The diameter of the tree is just the max of all dp1[i] and dp2[i]!

Interesting 21:

Floyd-warshall algorithm

As you all know, Floyd-warshall algorithm is used to solve the all-pairs shortest paths problem on directed or undirected graphs. I have always treated it like a black box, a magical thing that can solve this extremely difficult problem in O(n^3) time, and memorized it (because it is so easy to memorize). However, I found out, I actually don’t understand this algorithm at all. So, I watched some videos on it, and now I feel awakened. You might think, why do I need to learn this? Well, If you understand the algorithm, you can modify it whatever way you like to solve difficult problems, and you can also learn some graph DP.

It can be very hard to define the subproblems in graph DP. This time, we are going to restrict the internal nodes on an i-j path to define the subproblems. Internal nodes meaning excluding the endpoints, or excluding i and j. Let’s define dp[i][j][k] as follows: dp[i][j][k] = length of the shortest i-j path with all internal nodes in the set {1, 2, …, k}. Now, what are the subproblems? This is the ingenious part.

There are 2 possibilities: k is one of the internal nodes on the shortest i-j path with all internal nodes in the set {1, 2, …, k}, or k is not one of them.

Let’s consider the case where k is not one of the internal nodes. Then that’s easy!

dp[i][j][k] = dp[i][j][k-1]

But What if k is one of the internal nodes? We can imagine a path from i to j, and k being in the middle of it. Then, we can decompose it to 2 paths: a path from i to k, and a path from k to j! And because for both paths k is not the internal nodes (k is one of the endpoints), we can reduce the set{1, 2, …, k} to {1, 2, …, k-1}!!

dp[i][j][k] = dp[i][k][k-1] + dp[k][j][k-1]

Oh my god this is so beautiful. Therefore, the transition formula is:

dp[i][j][k] = min(dp[i][j][k-1], dp[i][k][k-1] + dp[k][j][k-1])

What are the base cases? (k=0)

if i==j, dp[i][j][0] is 0

if i->j is one of the edges in the graph, dp[i][j][0] is weight(i, j)

else, dp[i][j][0] is positive infinity

The code is as follows: (1-based)

for (int k = 1; k <= n; k++){

for (int i = 1; i <= n; i++){

for (int j = 1; j <= n; j++){

dp[i][j][k] = min(dp[i][j][k-1], dp[i][k][k-1] + dp[k][j][k-1]);

}

}

}

Interesting 22:

Bellman-Ford algorithm

Bellman-Ford algorithm is a single-source shortest path algorithm that can be used on graph with any edge weights. (Djikstra can only be used on non-negative edge weights) It can also detect negative cycles, but it cannot find the shortest cycle-free path (its NP-hard). Everyone knows Bellman-Ford, but I want to explain how and why it works in detail and also explain the graph DP aspects of it.

Last time, to define the subproblems, we restricted the internal nodes on a path to the set {1, 2, …, k}. This time, we will restrict the number of edges on the shortest path.

Let dp[i][j] = the shortest path from s (source) to i such that <= j edges are used.

There are 2 possibilities: the shortest path from s to i use all j edges, or it uses less than j edges.

Let’s consider the case where the shortest path uses less than j edges. That’s simple.

dp[i][j] = dp[i][j-1]

What about the other case? Well, we can imagine we pluck the last edge away, so it would be like this:

dp[i][j] = min(dp[k][j-1] + w(k, i)) for each edge k to i

For the base cases, dp[s][0] = 0 and dp[i][0] = positive infinity for all i not equal to s.

So, how do we code this thing? Instead of iterating through vertices and their neighbours that can reach them, we can just iterating through all the edges!

for (int j = 1; j <= n; j++){

for (int i = 1; i <= n; i++){

dp[i][j] = dp[i][j – 1];

}

for (int i = 1; i <= m; i++){

dp[e[i].to][j] = min(dp[e[i].to][j], dp[e[i].from][j – 1] + e[i].dist);

}

}

Now, we can just remove the second dimension and the for (i = 1 to n) loop entirely, and it will become our usual implementation.

Questions to think (if you don’t already know): why is n the upper limit of j? How can we detect negative cycles?

So, how do we code this thing? Normally, in an adjacency list, we store which vertices can be reached from i, but not which vertices can go to i. How can we implement this? From my experience, DP has 2 approaches: top-down and bottom-up. Top-down is usually implemented using recursive functions, and bottom-up is usually layers of for loops. Now, bottom-up approach also has 2 different types: one where you solve for the current problem with solved subproblems, and one where you optimize future problems by looking at the current problem as the subproblem. What do I mean? Let’s look at a few example:

Typically, bottom-up is coded like this:

for (int i = 0; i < n; i++){

for (int j = 0; j < n; j++){

//solve dp[i][j]

dp[i][j] = min(dp[i - 1][j] + f(i), dp[i][j - 1] + g(j));

}

}

But, you can also code it like this:

for (int i = 0; i < n; i++){

for (int j = 0; j < n; j++){

//treat dp[i][j] as a possible candidate for future problems

//dp[i][j] is already solved at this point

dp[i+1][j] = min(dp[i + 1][j], dp[i][j] + f(i + 1));

dp[i][j+1] = min(dp[i][j + 1], dp[i][j] + g(j + 1));

}

}

Interesting 23:

AtCoder ABC 127 F Absolute Minima

There is a function f(x) and q queries. Initially, f(x) = 0 and there are 2 types of query. The first type, you update f(x) to f(x) + |x – a| + b and in the second type, you need to output the minimum x the minimizes f(x), and the minimum value of f(x).

At first glance, this seems incredibly complicated and mathematical. Let’s break the problem down. First, you quickly realize the b in the update query is useless. Let’s say after some queries f(x) = |x – a1| + |x – a2| + … + B. Then, after another update query, you just add b to B, and it becomes the new B. Therefore, you just need to maintain a sum, and add the sum to the computed minimum. Let’s ignore b from now on.

What is the function f(x) = |x|? Well, basically it’s a V-shaped graph with the tip touching the origin. Now, what is the function f(x) = |x – a|? If you knew some functions math, you would know that f(x) = g(x – a) just mean f(x) is the graph of g(x) shifted rightwards a units. (Not hard to see why!) Therefore, f(x) = |x – a| is just the V-shaped graph shifted a units rightwards, so now the tip is touching the x-axis at a units to the right of the origin.

Now how do we think about the sum of the absolute terms? For example, what does f(x) = |x – 1| + |x – 4| look like? After some trial and error, or going to desmos, you can see that the graph looks like a bucket where the line slopes downwards when x < 1, is flat when 1 < x < 4, and slopes upwards when x > 4. Why is it like that?

Here is the juicy part. Instead of thinking of f(x) = |x – a| as a function where f(x) = -(x – a) when x – a < 0 and f(x) = x – a when x – a > 0, we can think of it as slopes! 2 lines, a line with slope -1 when x < a, and another line with slope 1 when x > a. So what is f(x) = |x – 1| + |x – 4|? when x < 1, the slope is -1 + -1 = -2, when 1 < x < 4, the slope is 1 + -1 = 0, and when x > 4, the slope is 1 + 1 = 2! Wow! This makes perfect sense!

After poking around desmos with functions such as f(x) = |x – a| + |x – b| + |x – c|, etc. You will find that the bucket becomes smoother and smoother, and you may start to notice the minimum always lies on the median, or lies on the line between the 2 median points (if there’s even number of points). Why is that?

Let’s consider the case where an even number of update queries were called, so the ‘bucket’ should have a flat bottom. Where does the minimum lie? Between the n / 2 and n / 2 + 1-th points!! Why? Well that is because all the x cancel out! Let’s say f(x) = |x – a1| + |x – a2| + … + |x – an|. Well, at the point between a(n / 2) and a(n / 2 + 1), All the absolutes from n / 2 + 1 to n are flipped! This is because if x – a < 0, |x – a| = -(x – a). Therefore, if x is between a(n / 2) and a(n / 2 + 1), f(x) becomes (x – a1) + (x – a2) + … + (x – a(n / 2)) + (-x + a(n / 2 + 1)) + … + (-x + an) and all the x cancel out! so the minimum is just sum of right side minus sum of left side! If you are still wondering, why is this the minimum? Think back to the bucket graphs and the adding of the slopes.

What about odd number? Well that’s just exactly at the point a(ceil(n / 2))! All the left x and the right x cancels out, and the middle term |x - a(ceil(n / 2))| is zero because x = a(ceil(n / 2))!

Now that we have figured out the answer, how do we find the median, and also how do we maintain the sum of left side and the sum of right side? You can directly apply the solution of 30107 – What is the median! Maintain 2 priority queues, a max priority queue storing the smaller elements, and a min priority queue storing the larger elements. Here’s my code if you want to see the implementation details:

https://atcoder.jp/contests/abc127/submissions/9702395

Interesting 23:

KMP

Let’s say s = “ABCDABCDABDE” and t = “ABCDABD” and initialize i (s) and j (t) as 0.

For each step of the algorithm, we check if s[i] == t[j].

s[0] == t[0], so i++ and j++

s[1] == t[1], so i++ and j++

…

s[6] != t[6] Oh no! what should we do?

In a naïve approach, we would set i to 1 and j to 0 and start over

However, we observe something!!

ABCDABCDABDE

ABCDABD

now notice, AB is both the suffix of substring ABCDAB (from s) and the prefix of t (ABCDABD). Therefore, instead of starting over from s[1], we can continue to s[6] and set j to 2!! In particular, AB is also the longest substring of ABCDAB such that it is both the prefix and the suffix, so this is the most optimal way (sort of like greedy).

s[6] == t[2]

s[7] == t[3]

…

s[10] = t[6]

ABCDABCDABDE

ABCDABD

wow! match!

Now, we need to compute the maximum length of strings that are suffix of s[0 ~ i] and prefix of t. Notice that suffix of some s[0 ~ i] must be also suffix of some t[0 ~ j] because they match (we know this from previous iterations). Therefore, we only need to care about the target string! Let’s denote this as lps[m]. (also the suffix cannot be the whole substring, because in that case we already matched)

some examples of lps:

lps(ABCDCAB) = AB

lps(ABCDDCBA) = A

To compute lps, we maintain 2 variables: pos and i. pos stores the index after matched prefix, and i stores the current character.

If t[pos] == t[i], we can just increase pos and i by 1, then assign lps[i] to pos.

What if t[pos] != t[i]? since the prefix and suffix of t[0~pos – 1] is the same, the prefix of the prefix is also the suffix of the suffix! Therefore, we reduce pos = lps[pos – 1] until t[pos] == t[i], or until pos == 0.

Now why is this O(m)? This seems like O(m^2)! Observe that if pos is decreased, it is at least decreased by 1. Moreover, for each iteration, pos is at most increased by 1. Therefore, it can only decrease at most m times!!

Wow I’m happy I finally understand KMP.

(In the pic I messed up m should be pos)

Interesting 25:

Linearity of expected values

expected value = probability i \* result i

E(X + Y) = E(X) + E(Y)

!!Doesn’t matter if the variables are dependant on each other or not!!

https://brilliant.org/wiki/linearity-of-expectation/

Interesting 26:

Knapsack DP

Given n, the total number of items and c, the capacity of the knapsack, and v[i] and w[i] for value and weight of each items, maximize the sum of v[i] such that the sum of w[i] <= c.

Unbounded knapsack:

You can use each items infinite amount of times

dp array of size c such that dp[i] = max sum of values with sum of weights == i.

dp[i] = max(dp[i – w[j]] + v[j]) (for all 0 <= j < n)

time: O(nc)

(There is another approach involving sorting, but that would only solve the case where c = c while the dp solution solve all the cases where c is smaller than the input c.)

* 1. knapsack:

You can only choose to include or not include the items

dp[n][c], dp[i][j] = max sum of values with sum of weights == j, only considering the items 0 ~ i.

if (w[i] > j) dp[i][j] = dp[i – 1][j]

else dp[i][j] = max(dp[i – 1][j], dp[i – 1][j – w[i]] + v[i])

time: O(nc)

bounded knapsack:

there are limits of how many of the ith item can be put in the knapsack, l[i] for each item

Stars and bars:

Given n stars. How many ways are there to divide them into k groups? (a group can have 0 stars)

(btw u guys know how to do if each group at least 1 stars right??? right??? plz say yes)

big brain 1: instead of k groups, think of k – 1 bars in between the stars!!

for example, (\*\*) () (\*) () -> \*\*||\*|

big brain 2: think of them as n + k – 1 objects!!!

Therefore, it becomes: you have n + k – 1 stars, change (k – 1) of them to bars (vice versa)

The answer is just (n + k – 1) choose (k – 1) or (n + k – 1) choose (n)!!!

Now, the problem:

Find the number of ways to distribute at most r things to n people.

big brain 3: add a ‘trash can’! The problem becomes: distribute precisely r things to n + 1 people (and a trash can)

That’s just r stars and n + 1 groups!

(n + k – 1) choose (n) -> (r + n + 1 – 1) choose (r) = (n + r) choose (r)

I like combinatorics :)



bruh