UAPC 2017 Editorial

Zac Friggstad
Dept. of Computing Science, University of Alberta

1 UAPC

We had 48 teams competing which included at least 22 students from high school. Excellent! Let's continue building momentum with another contest next year.

You can download a .zip with the solutions from all divisions (including the practice) from the UAPC page:
`cs.ualberta.ca/~UAPC/2017/solutions.zip`

I had solutions written in both `c++` and `python`. However, I had to clean them up for "public consumption" so I've only done that for `python` solutions. If you are dying to see how I did things in `c++`, send me an email.

Unfortunately when I implemented the solutions I did not use variable names that correspond to the parameters in the problem statement. Feel free to ask me for clarification if you are having a hard time seeing what I did.

The test data can also be downloaded:
`cs.ualberta.ca/~UAPC/2017/data.zip`

The `.in` files are an input file and the corresponding `.ans` files are the correct answers.

Terminology: I report running times using $O()$ notation, e.g. $O(n^2)$. If you haven't seen this before, it means the number of steps the algorithm takes grows like the function in the $O()$ brackets. e.g. If a function takes $O(n \cdot \log T)$ time then the number of steps taken is at most $c \cdot n \cdot \log_2 T$ for some constant c that does not grow with the input (and usually depends on the amount of time spent evaluating simple assignments, if conditions, etc.).

This is a coarse way to measure the running time of algorithms, but it provides a fairly clear picture if the algorithm you are considering is fast enough for the largest possible input as given in the specification.

Finally: I wrote this rather quickly. If there is a debilitating typo that you just can't figure out, let me know and I'll happily clarify.

2 Practice

The practice contest contained a fairly tough problem that is worthy of an editorial.

2.1 testing

Nothing much to say here. See the solutions for one particular implementation. There is a neat trick that avoids explicitly writing a loop you might want to look at if you are not used to the Python operator that unpacks a list/range/set/etc. into a sequence of arguments for a function.

2.2 factor

Just do trial division. Start a loop at $d = 2$ and increment by 1 each time the loop iterates. Stop when $d > n$.

If $n \bmod d == 0$ then report d as a prime divisor of n. Divide d out of n and repeat with this current d until all occurrences of d as a divisor of n are factored out.

Each iteration when some d is output, it will be a smallest divisor of the current n so it will be the next prime divisor.

2.3 coughing

There are two main steps:

- First, note the following. Suppose, for some given value x, we can answer the yes/no question "Is it possible to move each point at most x to have the required spacing between points"?

  If so, then we can solve the original problem with a *binary search*. We maintain two values `lo` and `hi` such that we know the answer lies somewhere in the range `(lo, hi]`. I chose `hi` $= 2^{30}$ based on the input values because we can always move point p_i to $i \cdot d$ which moves distance at most 2^{30} and satisfies the requirements.

  Iterate the following: try $x = \frac{\texttt{lo+hi}}{2}$ and ask the above question.

  If the answer is affirmative, set `hi` $= x$. Otherwise set `lo` $= x$. Repeat until `hi-lo` is very tiny (so tiny that rounding off should give the correct answer). The number of iterations is very small as each iteration cuts the entire range in half.

- To answer the above question, we process the points in left-to-right fashion. Each point, we try moving it to the leftmost point it can move that has the required distance from the previous point (not this may mean we move the point right).

  That is, the first point p_1 moves to $\min\{d, p_i - x\}$ to ensure it is at least d units from the triage station but as far left as possible given this constraint. If this exceeds $p_1 + x$, then we know this guess for x is too small and we return `False`. Otherwise, set p'_1 to be this position, move p_2 to $\min\{p'_1 + d, p_2 - x\}$ and iterate. If we successfully place all points, then return `True`.

Bonus Exercise (not necessary to solve the problem but still neat).
Show that if x is the minimum distance to move to solve the problem then $2 \cdot x$ is an integer.

3 Division II

3.1 helloworld

Nothing much to say. I used a tiny trick to avoid "if" statements. I put the three messages in the array and just indexed it using the input integer (minus 1 because arrays are 0-indexed).

3.2 overflow

The only hangup is for those who used `c/c++/java`. If you just used the `int` type then in fact you would see overflow in the `overflow` case when adding a and b. If you used `python`, there was no problem.

With `c/c++/java`, I would suggest using a larger integer type. Even an `unsigned int` would work as it gets you the one extra bit of precision to ensure $a + b$ never overflows and you can just check the answer against $2^{31} - 1$ directly. Even less worrisome, use a 64-bit integer type such as `long long int`.

Neat fact: if you just used the `int` type for a and b, then there will be overflow if and only if $a + b < 0$. e.g. this `c++` snippet.

```
int a, b;
cin >> a >> b;
if (a+b < 0) cout << "overflow" << endl;
else cout << a+b << endl;
```

3.3 callcentre

Simply summing all values in each query range is too slow: there are $n = 86,400$ values and $q = 50,000$ queries so the amount of work would be proportional to $n \cdot q$. That's too slow for 3 seconds of CPU time.

Rather, build an array $f[]$ where $f[i]$ is the *sum* of the first i values (so $f[0] = 0$). That is, if c_i denotes the number of calls that are placed in second i for $1 \leq i \leq n$ then

$$f[i] = c_1 + c_2 + \ldots + c_i.$$

We can compute $f[i]$ very quickly in an iterative fashion:

$$f[0] = 0 \text{ and } f[i] = f[i-1] + c_i \text{ for } i \geq 1.$$

Then when given a query s, t the answer is just $f[t] - f[s-1]$. That is, all calls up until time t but excluding those that were placed before time s. Each query can be answered immediately with no looping!

3.4 stacking

There isn't really much to say. Just do what the input says. Make sure you check if the first pair (r_1, c_1) refers to an empty stack or not (i.e. the corresponding grid value is 0 or not) before processing the move.

3.5 dicegame

This is a slightly annoying problem in that there are a number of things to check, even though each individual rule by itself is not so bad.

Tips:

- Test each possible rule before submitting to make sure they work.
- Create an array that "counts" the number of occurrences of each value so you can easily read off the maximum "of a kind" value. I sort of did this, but I used a python dictionary instead (same idea really).

3.6 lineup

The question was to determine if any three points in the given set of points lie on a common line.

The input was small enough that you could just check all triples of points (enumerated with three nested for loops) to see if they lie on a common line.

To see if three points p, q, r lie on a common line, you could calculate the line passing through p and q and see if r lies on this line. I suspect many tried to do this by calculating a linear equation like $y = m \cdot x + b$ where m is the slope of pq and b is the y-intercept of the line, and then checking if $q.y = m \cdot q.x + b$. But you have to be careful of vertical lines pq and you run into floating point issues.

A much easier solution if you know it is to check $(p - r) \times (q - r)$ where $p - r$ is the point with coordinates $(p.x - r.x, p.y - r.y)$ and $\times$ is the "cross product" $a \times b = a.x \cdot b.y - a.y \cdot b.x$. From standard linear algebra, this cross product is zero if and only if the three points lie on a line. If you haven't seen this, it is basically checking that the slope of the line pr equals the slope of the line qr by cross multiplying the fractions describing the slopes (which also gets rid of the vertical line issue).

Note: If you poke around online to learn about the cross product you will see a lot of information about cross products in 3-dimensional space. Think of points in this exercise as 3-dimensional points with z-component 0 and the cross product here is the value of the z-component of the 3D cross prouduct.

Running Time: $O(n^3)$. It can be solved in $O(n^2)$ time by hashing pairs of points into buckets with common slopes and then in each bucket seeing if the number of distinct lines actually created is equal to the bucket size or not, but this was too much for Div 2.

3.7 mountaineer

As with the `coughing` problem in the practice set, we will use a binary search. That is, we will implement a function that reports whether a given guess for s is sufficient or not to reach the top in T seconds. Given this, we start with `lo` $= 0$ and `hi` $= T$ and maintain that throughout we will not be able to reach the top with $s =$ `lo` stamina but we can with $s =$ `hi` stamina. We guess the midpoint of the range [`lo`, `hi`], ask if this midpoint is sufficient, and adjust `lo` or `hi` as appropriate.

Note we round the guess for the midpoint down to the nearest integer if needed because we are looking for the minimum integer s.

To decide if a given guess for s is sufficient, we simply have Link climb as far as he can with stamina s and rest only when necessary (i.e. cannot make the next ledge). Simulate this by iterating through the ledges and keeping a value `rem` indicating Link's remaining stamina. If `rem` is enough to make the next ledge, do it. If not, then rest, wait `r` seconds, and reset `rem` to our guess.

Each computation of whether a guess s was sufficient or not iterated through 50,000 points. The number of such guesses is roughly $\log_2 T \leq \log_2 2^{30} = 30$. In $O()$ terms, this runs in time $O(n \cdot \log T)$.

3.8 bigsums

For brevity, let $f(a, n, m)$ denote the answer for the given values a, n, m. It takes far too long to sum each term incrementally as there can be 2^{64} terms!

We will use recursion. Of course, $f(a, 0, m) = 1 \bmod m$ (don't forget the mod m in case $m = 1$, because then it should actually be 0). We can recursively say $f(a, n, m) = f(a, n-1, m) + a^n \bmod m$, but that will be too slow as well (actually, the recursion will go too deep and will crash the program).

But there is a **MUCH** faster recursive rule that only has to evaluate$f(a, n', m)$ for up to 64 different values of n'. If n is odd (so there are an even number of terms, e.g. $a^0 + a^1 + a^2 + a^3$) then note

$$f(a, n, m) = (1 + a^{(n+1)/2}) \cdot f(a, (n-1)/2, m) \text{ mod m (for odd } n).$$

For example (ignoring the mod m part):

$$f(a, 7, m) = 1 + a + a^2 + a^3 + a^4 + a^5 + a^6 + a^7 = (1 + a^4) \cdot (1 + a + a^2 + a^3).$$

For even n, there is a simpler rule.

$$f(a, n, m) = 1 + a \cdot f(a, n-1, m) \text{ mod m (for even } n).$$

Every other recursive call cuts the value of n in half and $n < 2^{64}$ to begin with, so the number of recursive calls will be at most $2 \cdot \log_2 n < 128$.

The only other thing to mention is that we still need to calculate terms like $a^n \bmod m$ quickly. This can be done with $O(\log n)$ multiplications: `python` has this algorithm as a built-in function `pow(a, n, m)` and I believe `java` does as well. In `c++`, you would need to implement it yourself. It can be calculated using a recursive rule as well.

In the following, every calculation should be performed mod m. I just don't feel like writing it at each step.

$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ a \cdot a^{n-1} & \text{if } n \text{ is odd} \\ a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \end{cases}$$

Note the last rule really just makes one recursive call. Store the value in a variable and then multiply the variable by itself so you only actually recurse once.

Running Time: $O(\log^2 n)$ because there are $O(\log n)$ recursive evaluations of $f(a, n, m)$ values for various n and each runs in $O(\log n)$ time (due to the evaluation of $a^n \bmod m$).

3.9 patternstring

Recall a, b, c, d are the given number of 00, 01, 10, 11 occurrences, respectively. There are two cases it is not possible:

- If $a, b > 0$ but $c = d = 0$. This is because the string must contain both 0s and 1s because $a, b > 0$ so there would be a consecutive 01 or 10.
- If $|b - c| > 1$. Think of it this way. If there was a string, it would alternate between sequences of 0s and sequences of 1s. Every 01 occurrence would be followed by a 10 occurrence before the next 01. Similarly, every 10 occurrence would be followed by a 01 occurrence before the next 10. So the 01 and 10 occurrences alternate.

If these two conditions are met (i.e. $|b - c| <= 1$ and $a, b > 0$ means one of c or d is nonzero) then I claim there is a solution.

- If $b = c = 0$ then we know either $a = 0$ or $b = 0$. If, say, $a = 0$ then the only string is just $b + 1$ 1s. Similarly, if $b = 0$ then the only string is $a + 1$ 0s.
- Otherwise, we know the string alternates between 0s and 1s.

  For the moment, suppose $a = d = 0$ so the string really just alternates between 0 and 1. If $c = b + 1$ then it has to start with a 1, otherwise we may start with 0 (if $c = b$ then we can start with either, but we prefer to start with 0 for the lexicographic condition).

  If $a > 0$, then we just expand the first occurrence of 0 into $a + 1$ occurrences of 0. If $d > 0$ then we just expand the last occurrence of 1 into $d + 1$ occurrences of 1.

3.10 parityhamlet

This was probably the hardest one to understand if you don't know what a graph is. Really, looking at the picture the goal is to double some edges so every node/circle is the endpoint of the correct parity (odd or even) of road segments. In fact, if there is a solution there is only one as the following discussion shows.

The road network was promised to be a tree: a connected graph with no cycles. Every tree with ≥ 2 nodes has a leaf: a node with precisely one edge touching it. So to ensure the leaf node has the proper parity we decide whether or not to double its only incident edge.

If we do not double it (i.e. its parity preference is odd) then reduce the problem conceptually by deleting the leaf node and its incident edge and flipping the parity preference of the other endpoint of that edge.

If we did double it, then conceptually reduce the problem by deleting the leaf node and its incident doubled edge but do not flip the parity of the ether endpoint of that edge.

In either case, we reduced the problem by one node. Repeat until there is only one node left. If its parity preference (in the residual problem) is odd, there is no way to satisfy it. Otherwise, we have already succeeded!

This can all be done quickly with one depth-first search. I suggest you look this up if you have not heard of it before. Look at the sample solution to see how I implemented it.

The running time is $O(n)$ as the search runs in linear time. So this is even fast enough for the Div 1 variant that had way more nodes.

4 Division I

Some problems were already discussed above in Division II.

4.1 stacking

See Division II.

4.2 patternstring

See Division II.

4.3 bigsums

See Division II.

4.4 mounaineer

See Division II.

4.5 parityville

See `parityhamlet` in Division II. The discussion there would also work in this version which only differs in the size of n.

4.6 seriesparallel

Ultimately it is a dynamic programming problem. The recurrence takes some effort to see and the implementation details are a bit annoying, so I think this was one of the hardest in the set (along with `convexholes`).

Each vertex of the tree corresponds to a subset of vertices. In particular, the root of the tree is the set $\{0, 1\}$. Every other vertex is a set of vertices $\{u, v, w\}$ that corresponds to one of the generation instructions u, v, w where w was a new vertex added with edges uw, vw where uv was an edge that was produced in an earlier step.

Add an edge connecting subsets S and T if T is a generation instruction u, v, w where uv was an edge that was first introduced in S. This completes the description of the tree.

For each vertex S of the tree, let $V(S)$ be all vertices contained in some vertex of the tree in the subtree rooted at S. For each vertex S of the tree and each $I \subseteq S$ that is itself an independent set, let $f[S, I]$ be the maximum size of an independent set $J \subseteq V(S)$.

To compute $f[S, I]$, use the following recurrence in a DP-like fashion (i.e. computing each entry only once and then storing it). If S is a leaf of the tree, then $f[S, I] = |I|$. Otherwise, let $T_1, \ldots, T_k$ be the children of S in the tree. For $J_i \subseteq T_i$, say that J_i is *consistent* with I if $J_i \cap (S \cap T_i) = I \cap (S \cap T_i)$. That is, if I and J_i agree on the vertices in common between S and T_i.

The recurrence is then this:

$$f[S, I] = |I| + \sum_{i=1}^{k} \max_{\substack{J_i \subseteq T_i \\ J_i \text{ consistent with } I \\ J_i \text{ an independent set}}} f[T_i, J_i] - |J_i \cap I|.$$

The subtraction in the inner term is to avoid double counting vertices in common between S and the child T_i.

The running time is indeed $O(n)$, but the constant suppressed by the $O()$ notation is moderately large which is why n was not set higher.

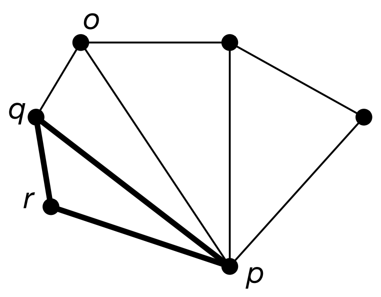
Neato Fact: One can generalize this construction technique. For a value k, start with a clique of size $k-1$. Each generation step picks some clique of size $k-1$ and adds a new vertex connected to every vertex of this clique. Then some edges may be deleted once this is done. The maximum independent set problem can be solved in time $O(f(k) \cdot n)$ where $f(k)$ is like 2^k (maybe slightly worse, perhaps $k \cdot 2^k$).

If $k = 2$ this generation process just makes a tree. If $k = 3$, this is the problem we discussed. Generally, the set of graphs generated this way are called the graphs with **treewidth** k. Many hard problems, like determining if there is a Hamiltonian cycle or the minimum dominating set or minimum vertex cover, can be solve in polynomial time in graphs with constant treewidth. We can even detect, in $O(g(k) \cdot n)$ time for some function $g()$, if an arbitrary graph has treewidth k and the algorithm produces a sequence to generate the graph.

4.7 convexholes

Another dynamic programming problem and, again, spotting the recurrence is the hardest part. Suppose we guessed the lowest point p on the best convex hole. Really, we will try all points p as the lowest point and take the best answer over all guesses.

Think of a convex hole with point p as the lowest point as being composed of triangles. See the picture below:



Say a triangle is *empty* if it contains no other points in its interior (so it is itself a convex hole). There are $O(n^2)$ triangles having p as a point, and for each triangle we can tell if it is empty in $O(n)$ time so it takes $O(n^3)$ time to generate all $O(n^2)$ empty triangles having p as a point.

Denote an empty triangle as pqr where q, r are the other points where the rotation from q to r about p is counterclockwise (see the picture). Let $f[q, r]$ be the biggest possible area of a convex hole having p as the lowest point where triangle pqr is the "leftmost" triangle in the hole (again, see the picture). Then for any empty triangle pqr say a point o *may precede* pqr if poq is also an empty triangle and the minimum angle between o and r about q is clockwise (again, see the picture).

$$f[q, r] = \text{area}(pqr) + \sum_{\substack{o\text{: a point} \\ o \text{ may precede } pqr}} f[o, q].$$

If there is no point o that may precede pqr then just say the term with the sum is 0.

The area of triangle pqr is just $\frac{(q-p)\times(r-p)}{2}$ where $\times$ is the cross product discussed in problem `lineup`.

It is convenient to sort all points about p by their angle with the ray that extends from p horizontally to the right. i.e. if p is the the origin then sort by CCW angle with the x-axis. A point q should precede a point r in this order if $(q - p) \times (r - p) > 0$, so the sorting function can be overloaded easily. Then we can iterate over the points in increasing order to easily fill the DP table in a bottom-up fashion.

Overall, each of the n points p takes $O(n^3)$ time to process so the algorithm runs in $O(n^4)$ time.

4.8 callcentres

See Division II.

4.9 babynames

My solution used *tries*:
`https://en.wikipedia.org/wiki/Trie`.

Given a dictionary of words (in this case, simple names) with total length d, one can construct a trie in $O(d)$ time for this dictionary.

For each query string s, in $O(\texttt{len}(s))$ time can then find *all* indices i such that the substring up to character i is a word in the original dictionary. Just march through the trie as you read characters of s and record any index that corresponded to a name in the dictionary.

Now do the same for the reverses of the dictionary: construct a trie for the dictionary of words obtained by reversing each name in the original set of names. This also takes $O(d)$ time. Then for each query we can also determine all indices i such the substring *after* index i is also a name in the dictionary.

A query name is simple if the march through the forward trie determines the whole name is in the trie. A query name is a compound if there is an index i such that the march through the forward trie determined the substring up to i is in the dictionary, and the march through the reverse trie determined the substring after i is also in the dictionary.

Overall, the query name s can be processed in $O(\texttt{len}(s))$ time, so the total time taken to process all query strings is linear in the total length of these strings.