Miscellaneous

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

Binary Indexed Tree

- Also known as *Fenwick tree*
- Given some array of numbers $a_1, a_2, \ldots, a_n$, support the following operations:
  - Update a_i for some i
  - Evaluate $a_j + a_{j+1} + \ldots + a_k$ for some $j \leq k$

Binary Indexed Tree

- The naïve approach takes $O(1)$ per update and $O(n)$ per sum
- Using prefix sums

$$b_i = a_1 + a_2 + \ldots + a_i$$

  so that

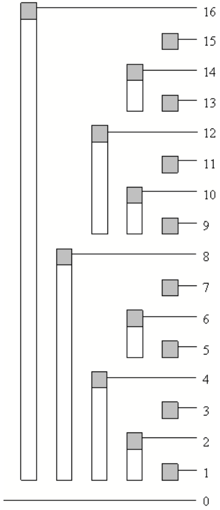
$$a_j + a_{j+1} + \ldots + a_k = b_k - b_{j-1}$$

  allows us to find sums in $O(1)$, but now updates cost $O(n)$.
- Compromise?

Responsibility

- Number the bits of i from least to most significant, 0-based. Suppose that bit number r is the lowest set bit, i.e. the binary expansion of i ends in a 1 followed by r zeroes. Then $i - 2^r$ would end in $r + 1$ zeroes.
- We say that i is *responsible* for the indices from $i - 2^r + 1$ to i inclusive.
- Denote the *tree sum*

$$t_i = a_{i-2^r+1} + a_{i-2^r+2} + \ldots + a_i.$$



Prefix Sums

- This allows us to compute the prefix sum

$$b_i = a_1 + a_2 + \ldots + a_i$$

  by analysing the binary expansion of i.
- We add t_i to the total, remove the lowest set bit of i, and repeat. This process takes $O(\log n)$ time.
- Example: consider 13 ($=$ 1101 in binary)

$$\begin{aligned}
t_{1101} &= a_{1101} \\
t_{1100} &= a_{1001} + a_{1010} + a_{1011} + a_{1100} \\
t_{1000} &= a_{0001} + a_{0010} + \ldots + a_{1000} \\
\therefore b_{1101} &= t_{1101} + t_{1100} + t_{1000}.
\end{aligned}$$

Isolating the Lowest Set Bit

- Suppose `i` has binary expansion of the form `a1b`, where `a` is some sequence of 0s and 1s and *b* is all zeroes.
- Then its complement `~i` has binary expansion `(~a)0(~b)`. Notably, `~b` is all ones.
- Thus `(~i)+1` is of the form `(~a)1b`, so it agrees with `i` up to the lowest set bit and differs thereafter. But `(~i)+1` is just the 2's complement of `i`, so it is simply `-i`.
- Therefore `i&(-i)` exactly isolates the lowest set bit.

Prefix Sums

- **Implementation**

```
// computes the sum up to and including i
int prefix_sum(int i) {
  int ret = 0;
  while (i > 0) {
    ret += t[i];
    i -= i & (-i);
  }
  return ret;
}
```

Updating Tree Sums

- If $a[i]$ increases by k, then all the tree sums $t[j]$ where j is responsible for i will also increase by k, and the same is true for decreasing by k.
- We add k to t_i, add the lowest set bit of i, and repeat. This process takes $O(\log n)$ time.

Updating Tree Sums

- **Implementation**

```
// increase a[i] by k
void update(int i, int k) {
  while (i <= n) {
    t[i] += k;
    i += i & (-i);
  }
}
```

Scaling

- This data structure can 'easily' handle the extra operation 'multiply all a_i by a constant c'.
- This operation would multiply all t_i by c, as follows:

```
void scale(int c) {
  for (int i = 1; i <= n; i++) t[i] *= c;
}
```

BIT or Range Tree?

- A BIT has the same $O(\log N)$ complexity for all operations as a range tree.
- However, due to simpler implementation, it is quite a bit faster.
- **Range trees are more flexible.** A number of problems can be solved by range trees that cannot be solved by BITs. Specifically:
    - Range trees can be constructed over any *associative* operation. This includes GCD, min/max, etc.
    - BITs can only be constructed over *invertible* operations: addition is invertible (given $A + B$ and B, we can find A), as is multiplication and matrix multiplication, but GCD and min/max are not.
    - The query operation of a range tree can be customised to find e.g. the index i such that the sum of $[0, i)$ is k, in $O(\log N)$ time. In a BIT this can only be achieved with binary search in $O(\log^2 N)$ time.

Table of Contents

Floating-point arithmetic

- Try to keep things in integers as much as possible
- Avoid unnecessary divisions and other arithmetic operations
- Single precision floating point numbers fail very often, but doubles are mostly reliable
  - They are not foolproof though!
  - Going to long doubles is not a panacea

Floating-point comparisons

- When comparing floating point numbers, compare up to a small constant ϵ to allow for rounding errors

```
const double EPS = 1e-8;
if (fabs(x-y) < EPS) {
  // equal
}
```

- Many problems that require floating point answers will specify how many decimal places they want.
  If you are e.g. binary searching for an answer, that tells you how small your ϵ needs to be until you can stop.

Table of Contents

Solutions to Contests 1 & 2

- This section will cover brief solution outlines for the problems in the first two contests.
- Code is not provided. If you didn't solve these problems in the contest, coding up the solutions is good practice!
- Log in at `https://cs4128.omelaen.co` with the credentials that were emailed to you, and submit your solutions.
    - You'll also need this login for contest 3, so don't forget it!

Contest 1: A. Cartography

- The appreciation for this flavourtext was appreciated.
- **Problem Statement** Given a grid consisting of `X` for roof and `.` for ground, count the number of disconnected areas present in the grid. Cells are connected if adjacent horizontally or vertically.
- **Solution** Iterate over all cells, starting a BFS/DFS whenever we reach a roof cell which hasn't been "coloured" yet. Use the BFS/DFS to "colour" all found cells with a number, then increment the counter for the next colouring.
  The number of different colours used gives the answer.
- This is a very standard graph problem, and is often present as a basic task that needs to be performed as part of the solution to a harder problem.
- Union-find, à la Masking, is *not* a good idea.

Contest 1: B. Map War

- **Problem Statement** Your revolutionary startup is solving the all-pairs shortest paths problem, except new freedom-related restrictions prevent people who leave Earth from returning.
  Given a graph with up to $1,000$ vertices, some on Earth and some not, answer $Q \leq 50,000$ queries asking for shortest paths between two points.
- **Solution** Simply throw away edges that go from $\neg$Earth to Earth, since they can never be used. Then use Floyd-Warshall as normal.
- A small number of people got full marks by simply performing a Dijkstra for each query (several significant micro-optimisations were needed). This was not the intended solution and only passed in time because of an overly generous time limit and insufficiently strong test data (a good candidate for an assignment next year...).

Contest 1: C. Atlantis

- **Problem Statement** Given a 2D landmass consisting of $N \leq 10^6$ (40% subtask: $N \leq 50,000$) vertical bars of varying heights and widths 1, submerge it and then remove it from underwater. How many squares of "air" remain flooded?
- **Subtask Solution** For each bar, find the tallest bar to the left and right of it. The difference between the shorter of the two and the current one (if positive) is the depth of flooding for this bar. ($O(N^2)$)
- **Solution** We are redoing a lot of work. If we know the tallest bar to the left of bar i, we can compute it for bar $i+1$ in constant time.
  Compute all left tallest bars in one left-to-right pass, and then all the right tallest bars in a right-to-left pass. In a third pass, calculate the answers. This takes $O(N)$ time in total.

Contest 1: C. Atlantis

- **Problem Statement** Given a 2D landmass consisting of $N \leq 10^6$ (40% subtask: $N \leq 50,000$) vertical bars of varying heights and widths 1, submerge it and then remove it from underwater. How many squares of "air" remain flooded?
- **Subtask Solution** For each bar, find the tallest bar to the left and right of it. The difference between the shorter of the two and the current one (if positive) is the depth of flooding for this bar. ($O(N^2)$)
- **Solution** We are redoing a lot of work. If we know the tallest bar to the left of bar i, we can compute it for bar $i+1$ in constant time.
  Compute all left tallest bars in one left-to-right pass, and then all the right tallest bars in a right-to-left pass. In a third pass, calculate the answers. This takes $O(N)$ time in total.

Contest 2: A. Enrolment

- **Problem Statement** Initially, each student office either processes requests or tells you go to another one. Updates are only of the form: a student office starts processing received requests instead of forwarding them. These are intermixed with queries: if you go to a specific student office, which one will process your request, or will you be stuck in a loop?
- **Solution** Problem 4C (Ancient Berland Roads) was emphasised many times. Reverse the input, so we are only ever joining things together, not breaking them apart. Make an auxiliary array for the union-find which stores the "endpoint" for all requests in a group (or if it is a loop). This only has to be correct for the representative element. When performing a join, update this for the new representative element. Loops are created *only* if you join a group to itself.

Contest 2: B. Largest Sum Designation

- **Problem Statement** Lizzy enjoys the taste of her exam paper. From a sequence of N integers, select three non-intersecting subarrays, each of length exactly H, so as to maximise the sum.
- **Solution** Only dynamic programming problems have descriptions this short...
  Subproblem select exactly $k \leq H$ subarrays out of the subsequence from i to N, maximising their sum.

  $$f(\text{index we are at}, \#\text{subarrays left}) = \text{highest sum}$$

  Recurrence either select a subarray starting at the current index, or move on to the next index.

  $$f(i, k) = max(f(i + H, k - 1) + sum(i \to i + H), f(i + 1, k))$$

  Base cases trivial
  Complexity $O(3N) = O(N)$

Contest 2: C. Logo Changes

- **Problem Statement** Given a string of up to $100,000$ (subtask: $5,000$) opening and closing parentheses, process updates that flip a single character, and queries asking whether a given substring is balanced.
- **Subtask Solution** Naïvely process each query by looping through the array with a counter, and receive 45 points. Specifically, increment the counter for each `(`, and decrement it for each `)`. If the counter goes negative, you've closed too many times and can't recover, so the answer is no. If you finish the loop and the counter is nonzero, the answer is no.
- **Solution** Build a range tree. Each node will need to store two numbers: a count of outgoing parentheses to the left and right. Merging the nodes has two cases but is relatively straightforward.

Table of Contents

Contest 3 and final exam: info & tips

- Contest 3 will have 3 problems of equal weight, as before, and one will have a subtask.
- The final will have approximately 8 problems (unconfirmed) of equal weight. Some of them may have subtasks.
- **Read and attempt the subtasks!** As has been the case in the previous contests, subtasks are often free marks, and are worth a significant percentage of the problem. Even if you can't solve the full problem, chances are you can easily code up a correct subtask solution. Don't let free marks go to waste.
- **Read *ALL* the problems!** My idea of the in-contest difficulty of the problems doesn't always line up with yours. *Note that it was never specified that the problems would be in difficulty order.* **This will not be the case in contest 3**.

Table of Contents

Course Feedback

- This is a difficult course with a lot of complicated content.
- Any student feedback we can get is valuable for improving the structure and teaching for future years.
- If you haven't done so already, please go to `https://myexperience.unsw.edu.au` and fill out the survey for this course.

Table of Contents

Problem Set 4: F. Lucky Array

- This problem was more difficult than intended.
- A fairly large number of people took advantage of Codeforces' substandard test data for this problem, and wrote incorrect-complexity (linear update) solutions that used a Binary Indexed Tree.
- These solutions passed within the time limit, because a BIT is faster than a range tree (sometimes over 2x), even though the time complexity for all operations is the same.
- You can try submitting that solution, or the intended one that follows, to the original problem set, or directly to Codeforces.

Problem Set 4: F. Lucky Array

- **Problem Statement** Numbers consisting only of the digits 4 and 7 are considered lucky.
  Given an array of $n \leq 10^5$ numbers, perform $m \leq 10^5$ operations of the following types:
  - add l r d: add a positive integer $d \leq 10^4$ to all elements in $[l, r]$.
  - count l r: count and output the number of lucky numbers in $[l, r]$.
- Conveniently (luckily?) all numbers given are not greater than 10^4, and are guaranteed to still not be greater than 10^4 after all operations have been performed.

Problem Set 4: F. Lucky Array: Solution

- **Initial observation** How many lucky numbers are there in total? 30.
- Let a_i be the ith number in the array, and d_i be the difference between a_i and the next lucky number (or 0 if a_i is lucky).
- Implement a lazy-update range tree storing both a_i and d_i that implements the following operations (note that each node will store more than one value):
  - subtract(l, r, d): subtract d from all d_i for $i \in [l, r]$ (and add d to corresponding a_i). This is a standard lazy update.
  - minimum(l, r): find the minimum d_i in $[l, r]$. This is a standard min-query.
  - count(l, r): count how many zeroes occur in $[l, r]$. This is also a fairly standard query.
  - left(l, r): find the smallest $i \in [l, r]$ where $d_i = \text{minimum}(l, r)$. Also a standard query.
  - set(i, d): set d_i to d. This is a standard update.

Problem Set 4: F. Lucky Array: Solution

- To summarise the operations:
    - subtract(l, r, d): subtract d from all d_i for $i \in [l, r]$.
    - minimum(l, r): find the minimum d_i in $[l, r]$.
    - count(l, r): count how many zeroes occur in $[l, r]$.
    - left(l, r): find the smallest $i \in [l, r]$ where $d_i =$ minimum(l, r).
    - set(i, d): set d_i to d.
- To do a count operation, we check if minimum(l, r) is zero. If it is, the answer is count(l, r). Otherwise, the answer is zero. Recall that minimum(l, r) will give the smallest distance of any a_i in this range to the next lucky number.
- To do an add operation, first subtract(l, r, d). Now, some of the d_i could be negative. While minimum$(l, r) \leq 0$, find $j =$ left(l, r), and a_j. We can find the new, correct d'_j in constant time, and set(d, d'_j).

Problem Set 4: F. Lucky Array: Solution

- You might notice that the amount of work we do in an add operation isn't necessarily logarithmic.
- In fact, after performing subtract(l, r, d) in $O(\log n)$ time, we might spend $O(r - l)$ time updating the d_js.
- However, recall that we are guaranteed that no a_i will ever exceed 10^4, and also that there are only 30 lucky numbers. This means that we will only have to do this extra work at most 30 times per i, which adds just $30n \log n$ to our time complexity.
- Thus, we have a total time complexity of $O(m \log n + n \log n)$.

Example Problem: Second Shortest Paths

- **Problem Statement** Given a weighted directed graph with non-negative edge weights, what is the length of the second shortest path on the graph from s to t? The second shortest path is the walk with the shortest length that is distinct from the shortest path; in particular, it is the same length as the shortest path if and only if there are multiple shortest paths.
- **Input** A weighted directed graph with N vertices and M edges ($1 \leq N, M \leq 100,000$).
- **Output** A single integer, the length of the second shortest walk.

Example Problem: Second Shortest Paths

- Observe that the second shortest walk will differ from a given shortest path by at least one edge.
- Then after finding the shortest path, we can iterate over all edges on the shortest path, and see what happens if we go along a different edge instead.

Example Problem: Second Shortest Paths

- Let the shortest path from u to v have length d_{uv}. If some edge (u, v) is blocked, and we take some other edge (u, w) instead, with weight e_{uw} we get a path of length $d_{su} + e_{uw} + d_{wt}$.
- Since we're iterating over all smallest possible differences to the shortest path, one of these distances will be our answer.
- So we just try them all (i.e. all choices of (u, w)) and take the one of minimum length.
- How do we efficiently compute all the d_{wt}?

Example Problem: Second Shortest Paths

- **Implementation**

```
// dist1[v] = shortest distance from v to 1
vector<int> dist1(n+1, -1);
vector<int> prev1(n+1, -1);
{
  set<edge> s;
  vector<bool> seen(n+1);
  prev1[1] = -2;
  for (s.insert(edge(0, 1)); !s.empty(); s.erase(s.begin())) {
    edge cur = *s.begin();
    if (seen[cur.y]) continue;
    dist1[cur.y] = cur.x;
    seen[cur.y] = true;
    foreach(v, adj[cur.y]) {
      if (!seen[v->y])
        if (dist1[v->y] == -1 || dist1[v->y] > cur.x+v->x) {
          prev1[v->y] = cur.y;
          s.insert(edge(cur.x+v->x, v->y));
        }
    }
  }
}

// to be continued
```

Example Problem: Second Shortest Paths

- **Implementation (continued)**

```
// dist2[v] = shortest distance from v to n
vector<int> dist2(n+1, -1);
vector<int> prev2(n+1, -1);
{
  set<edge> s;
  s.insert(edge(0, n));
  vector<bool> seen(n+1);
  prev2[n] = -2;
  for (s.insert(edge(0, n)); !s.empty(); s.erase(s.begin())) {
    edge cur = *s.begin();
    if (seen[cur.y]) continue;
    dist2[cur.y] = cur.x;
    seen[cur.y] = true;
    foreach(v, adj[cur.y]) {
      if (!seen[v->y])
        if (dist2[v->y] == -1 || dist2[v->y] > cur.x+v->x) {
          prev2[v->y] = cur.y;
          s.insert(edge(cur.x+v->x, v->y));
        }
    }
  }
}

int res = 1<<30;
for (int v = n; v != -2; v = prev1[v])
  foreach(e, adj[v])
    res = min(res, dist1[v] + e->x + dist2[e->y]);
```

Various DP Problems

(TBA)