Programming Assignment I

Yohandi [SID: 120040025]

1. 02 Representation

In this problem, we are asked to implement a program that shows a corresponding unique binary representation of an integer n with only two numbers: 0 and 2. Notice that, in a binary representation $10 = 8 + 2 = 2(3) + 2(1)$, we denote $a(b)$ as a^b. As required in the task, we are about to substitute a number that is not 0 or 2 with its binary representation. For instace, 10 as $2(2 + 2(0)) + 2$.

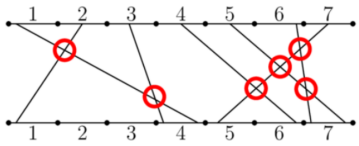
Denote $r(n)$ as a function that returns a string representation of an integer n. We are interested to find the value of $r(n)$. As for its base cases, $r(0)$ returns "0" and $r(2)$ returns "2". Note that, for any integer $n_{(10)}$, n is representable in the base-2 numeral system. For example, $13_{(10)} = 1101_{(2)}$. Every bits of the base-2 representation for integer n indicate the affected two-power that contribute to a summation of integer n. Suppose we denote B as the set of every bits of the base-2 representation for integer n that has value 1. Also, denote $s1 + s2$, an addition of two strings $s1$ and $s2$, as a concatenation string of $s1$ and $s2$ with a character '+' in between of those strings. Then, we have the function recursion as $r(n) = \sum_{b \in B}$ "$2(r(b))$".

As required in the main task, we need to write the result from right to left, in order of most significant digit to least significant digit. To handle this, we simply loop the bit possibilities decreasingly from 29 (since $log_2(10^9) \approx 29.9$) to 0. The function implementation can be found in the cpp document file.

This solution is arguably both simple and compact in its implementation and relatively efficient in terms of both time complexity and space complexity compared to the other naive methods. For instance, a method of bruteforcing all possibilites for an offline query does not work efficiently as it requires a huge storage and the main task only requires few calls for $r(n)$.

2. Crossing

In this problem, we are asked to implement a program that maximizes the number of crossing points for n wires that has each of the endpoints connected between two terminals. The i-th wire connects the i-th segment of the first terminal and the $a[i]$-th segment of the second terminal. In a case where $n = 7$ and $a = [4, 1, 4, 6, 7, 7, 5]$, we have 6 as the maximum number of crosses. The below figure shows the illustration.



Suppose we want to cross both wire i and j. Then, it is claimable that if $i < j$, then $a[i] \geq a[j]$ must be satisfied so that both wires can be crossed. On the contrary, if $i > j$, then we simply swap the value of i and j and use the previous claim. This works as a line $y = x/(a[i] - i) + c1$, which represents the i-th wire, and a line $y = x/(a[j] - j) + c2$, which represents the j-th wire, intersect at a point that lies in between the terminal (geometrically provable). Moreover, the claim is supported in the sample case as for every pairs of wires that cross, i.e. $(1, 2)$, $(1, 3)$, $(4, 7)$, $(5, 6)$, $(5, 7)$, $(6, 7)$, the condition of $a[i] \geq a[j]$ are satisfied. Notice that, crossing wire i with wire j and crossing wire j with wire i are counted as one intersection. Therefore, we can focus only for a case where $i < j$. The problem is now furthersimplified as counting the number of i such that $i < j$ and $a[i] \geq a[j]$. In other words, maintaining updates for an array of frequencies of the $a[i]$ values and queries the sum of frequencies from $a[i]$ to n are now the tasks.

To satisfy the condition of runtime limit, we need a data structure that supports an update for an element in an array in $\mathcal{O}(log(n))$ and a query that returns the sum of a particular subarray in $\mathcal{O}(log(n))$. It is found that a data structure segment tree supports the needs. Kindly check this link for the detail information about segment tree. In short, segment tree is a tree data structure that utilizes segments to store information on intervals which allows both updates and queries. The implementation for the segment tree can be found in the cpp document file. As a snap of code, the main function is as follows:

```cpp
int main(){
    cin >> n;
    for(int i = 1; i <= n; ++i){
        cin >> a[i];
    }
    build();
    for(int i = 1; i <= n; ++i){
        ans += query(a[i], n);      // increase ans by sum of freq[a[i]..n]
        update(a[i]);               // increase freq[a[i]] by 1
    }

    cout << ans << endl;
}
```

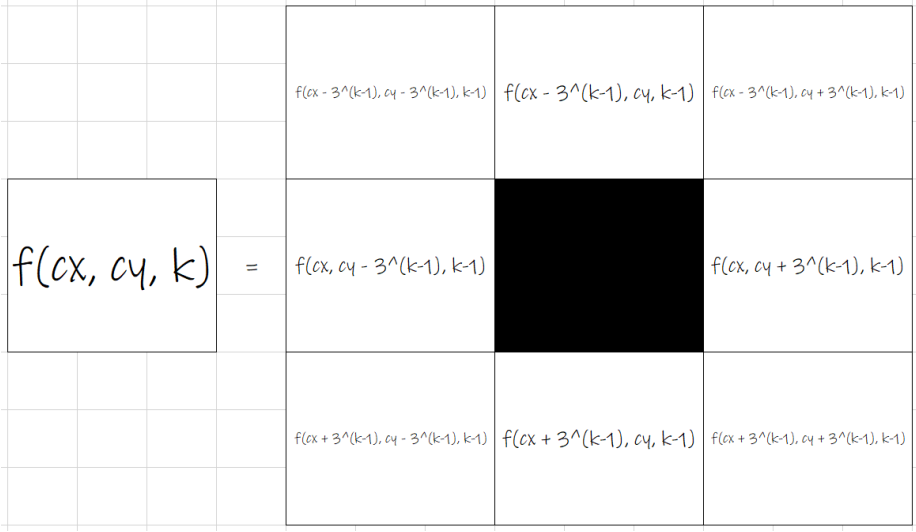
The code works efficiently (even though the implementation uses a kind of complex data structure) as each query and update require only $\mathcal{O}(log(n))$ of computation. This is due to the height of the tree that is preconcerted as $log(n)$. Since the queries are done n times and the updates are done n times (in the for loop), the time complexity of the main function becomes $\mathcal{O}(nlog(n))$. In a naive brute attempt, the update requires only $\mathcal{O}(1)$ computation; however, the query requires $\mathcal{O}(n)$ computation. Consequently, a naive attempt results in $\mathcal{O}(n^2)$ computation (way worse than the used method).

3. Sierpiński Carpet

In this problem, we are asked to implement a program to print out the Sierpiński carpet of size $3^k \times 3^k$ given a value of k.

Based on the pattern of the carpet, this problem is a recursion-related problem. Suppose we have a grid of size $3^k \times 3^k$ that is initially filled with empty character. Denote $f(cx, cy, k)$ as a method to fill the grid centered at (cx, cy) with the Sierpiński pattern of $3^k \times 3^k$ length. We are interested to perform $f(\lfloor \frac{3^k}{2} \rfloor, \lfloor \frac{3^k}{2} \rfloor, k)$ as a procedure before we print out the whole grid. As a base case we have $f(cx, cy, 0)$ fills $grid[cx][cy]$ as #.

In the recursion it is noticeable that $f(cx, cy, k)$ calls $f(cx + dx * 3^{k-1}, cy + dy * 3^{k-1}, k-1)$ such that $|dx| + |dy| > 0$ and $|dx|, |dy| \leq 1$. As a visualization, the recursion looks like



Therefore, the implementation of the function $f(cx, cy, k)$ is as follows:

```cpp
const int dx[] = {-1, -1, -1, 0, 0, 1, 1, 1};
const int dy[] = {-1, 0, 1, -1, 1, -1, 0, 1};

void fill(int cx, int cy, int k){
    if(k == 0){
        grid[cx][cy] = '#';
    } else {
        for(int n = 0; n < 8; ++n){
            fill(cx + dx[n] * power3[k - 1], cy + dy[n] * power3[k - 1], k - 1);
        }
    }
}
```

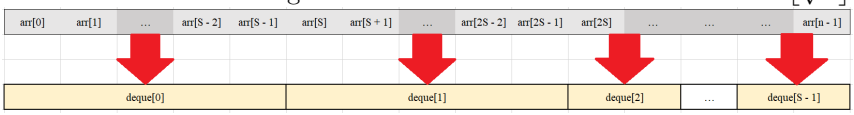
In terms of both time complexity and space complexity, this method seems to have similar ones with a straightforward bruteforce method. However, this recursion method allows such a simplicity in the implementation as the fill function only takes less than 10 lines of code.

4. Array Maintenance

In this problem, we are asked to implement a data structure that supports insert, delete, and range sum query in an array. As n is quite large, of course, a naive attempt where both updates and queries are done directly in the array, is not expected to be accepted in all cases.

In this task, a query optimization technique, namely square root decomposition, is applied. Moreover, double-ended queue, a data structure, is also used to support the application of the technique. The solution works as follows:

- The technique firstly decomposes the given array into small chunks (double-ended queues) with a specific size of $\sqrt{n}$. This means, we are expected to have a total of $\frac{n}{\sqrt{n}} = \sqrt{n}$ of double-ended queues. The illustration is shown in below figure. Note that we denote a variable S as $\lfloor\sqrt{n}\rfloor$.



- Suppose we want to insert a value of x to the k-th position. Then, k-th position must be located exactly in the $\lfloor\frac{k}{S}\rfloor$-th deque, and x must be located exactly one position after $k - (\lfloor\frac{k}{S}\rfloor \times S)$ first elements. To insert, we need to perform `pop_front` function to remove those elements and save it into another data structure (stack for example). We simply `push_front` x to the deque and and move back all elements that have been saved. This takes at most $2\sqrt{n}$ operations as the $|deque[i]| \leq \frac{n}{S} < 2\sqrt{n}$ for

$i = 0, 1, ..., S-1$. To maintain the size of the deque to always smaller or equal to S, we pop and transfer the back value of the $deque[i]$ to the front of the $deque[i+1]$ if we find $deque[i] > \frac{n}{S}$ and repeat this until $i = S-1$. This takes at most $\sqrt{n}$ operations. Hence, in an insert operation, we perform $\mathcal{O}(\sqrt{n})$ operations.

- Suppose we want to delete a value from the k-th position (in 0-based index). We can similarly perform the same method with the insert operation. However, after removing and saving $k - (\lfloor \frac{k}{S} \rfloor \times S)$ first elements, we permanently remove the next element. After returning it back, we need to maintain all $|deque[i]| \geq |deque[i+1]|$ for $i = 0, 1, ..., S-1$. This can be done by popping and transfering the front value of $deque[i+1]$ to the back of $deque[i]$. This delete operation also performs $\mathcal{O}(\sqrt{n})$ operations.
- Lastly, in a range sum query from L to R where $(L \leq R)$, we need to sneak an array namely sum to all the previous update operations. A $sum[i]$ denotes the sum of all elements inside $deque[i]$. This can be done along with insert and delete operation by adding or subtracting a value of x from the $sum[\lfloor \frac{pos}{S} \rfloor]$. Then, to answer the range sum query, we iterate an index from L to the nearest $k_1 \times S - 1$ such that $L \leq k_1 \times S - 1$. This takes at most $\sqrt{n} - 1$ operations. After that, we iterate an index from the nearest $k_2 \times S$ to R such that $k_2 \times S \leq R$. This also takes at most $\sqrt{n} - 1$ operations. Lastly, to calculate the sum of $arr[k_1 \times S \, ... \, k_2 \times S - 1]$ we simply iterate an index idx from k_1 to $k_2 - 1$ and add $sum[idx]$ to the answer, this takes at most $\sqrt{n}$ operations. Hence, the sum query performs $\mathcal{O}(\sqrt{n})$ operations.

As a complexity analysis, we have each type of update and query to perform in $\mathcal{O}(\sqrt{n})$ operations. Since there are n operations being done, the total executions is $\mathcal{O}(n\sqrt{n})$. This works pretty much well as $n \leq 2 \times 10^5$. Although, another data structure solutions such as treap or splay tree might provide $\mathcal{O}(nlog(n))$ total operations, square root decomposition seems to provide a relatively convenient implementation.