CP Gym

by Kevin Foong

1 Time complexity

If log(n) is involved, means that problem is divided and split (usually into 2/half). Almost always binary search/Merge Sort/pivoting solution.

2 Maths

2.1 Check number parity fast

$$x\&0b1 == 0?even : odd$$

2.2 Quick multiply/divide by powers of 2

$$x >>= n$$

to divide by 2^n,

$$x <<= n$$

to multiply by 2^n

2.3 XOR

Properties:
XOR is associative
$x \oplus x = 0$
$x \oplus 0 = x$
hence, no matter how many times 0 is XOR'ed, it'll always be 0. for 1s, the result is if $occurrence\&0b1 == 1$ then 1, else 0.

For some applications: XOR Trick

2.3.1 Application: log2base with shift right logical

we know log base 2 is the number of times a number n has to be divided by 2 to reach 1.
express n in binary representation. (e.g. 5 = 0b101) hence, number of bitwise left shifts = 2.

C++ Implementation of log base 2:

```
int log2base(int64_t n) {
// count number of shifts(which represent divisions by 2) required to get n == 1
int shifts = 0;
while (n >> shifts > 1) {
shifts++;
}
}
```

2.4 Fast exponentiation by squaring/binary exponentiation

To find a^b in $O(logb)$, let b be represented as powers of 2, simplest way to express in binary. e.g. $10 = 1010 = 2^3 * 2^1$

1 when pow is odd, convert to even pow by: $a^{pow} = a^{pow-1} * a$

2 when pow is even, $a^{pow} = a^{pow/2} * a^{pow/2} = (a^{pow/2})^2 = (a^2)^{pow/2}$

Since the depth of the recursion pow / 2 for each call, the max number of calls is $O(logn)$.

Read more: Fast power algo explanation

C++ Binary Exponentiation Implementation:

```
int64_t exp(int64_t a, int64_t pow) {
     int64_t res = 1;
// according to Errichto, iterative is faster than recursive impl
     while (pow > 0) {
         if ((pow & 1) == 1) {
             res = res * a;
         }
         pow /= 2;
         a = a * a;
     }
    return res;
}
```

2.5 Gaussian Addition

Sum of 1 to n, $f(n) = (n+1) * n/2$
Can be applied to compute multiples of k, $g(k) = k + 2k + 3k... = k(1 + 2 + ... + n)$

Intuition:

1 + 2 + ... + n = n + n-1 + ... + 1
hence 2 * f(n) = (n + 1) + (n + 1) ... n times
f(n) = n * (n-1) / 2

2.6 GCD and Euclidean Algorithm

Brute force is to iterate through the space and search for i s.t $a\%i == 0 \&\& b\%i == 0$ where $i \leq a$ and $i \leq b$. Otherwise, use euclidean algo,

$$gcd(a, 0) = gcd(0, a) = a \quad \text{(base case)}$$
$$gcd(a, b) = gcd(b, a\%b) \quad \text{(recursive case)}$$

Intuition:

for (2), lets assume $a > b$ and express a in remainder-quotient form where $a = x * b + r$
gcd(a,b) is s.t $k * gcd(a, b) = a$ and $y * gcd(a, b) = b$ exists.
hence, $r = k * gcd(a, b) - x * y * gcd(a, b)$ and hence $r | gcd(a, b)$

next, we know gcd(a, b) divides b and c.
hence, $gcd(b, c) \geq gcd(a, b)$

next, prove that gcd(b,c) also divides a. (same proof as gcd(a,b) divides c)
then, $gcd(a, b) \geq gcd(b, c)$
hence, $gcd(a, b) = gcd(b, a - b)$
$r = a\%b = a - b - b... - b$
hence, $gcd(a, b) = gcd(b, a\%b)$

C++ STL

```
#include <algorithm>
  __gcd(int, int)
```

Read more: Euclidean Algo Khan Acad

2.7 GCD and LCM

$$gcd(a, b) * lcm(a, b) = a * b$$

Intuition:

suppose a = 6 and b = 15, represent in unique prime factorisation by fundamental theorem of arithmetic

$$a = 2 * 3 \tag{1}$$
$$b = 2 * 3 * 5 \tag{2}$$

gcd(a,b) is the common prime factors which is $2 * 3$
lcm(a,b) is the max power of each prime factor of a and b which is $2^1 * 3^1 * 5^1$.

$$a * b = 2^2 * 3^2 * 5 \tag{3}$$
$$= gcd(a, b) * 2 * 3 * 5 \qquad = \tag{4}$$

2.8 Find primes with Sieve of Eratosthenes

There are no known formulas to find prime numbers without some kind of search in the number space. Sieve is a complicated sounding but actually simple and brute-force like algorithm to finding primes.
Sieve video

0 we know that 0 and 1 are not prime numbers (for fundamental theorem of arithmetic to hold)

1 store a prime number tracker array from 0 -¿ n (hence size is n + 1), let all elems be initially marked as prime.

2 for each integer from 2 till n, if prime, then update all multiples of i to non-prime starting from i * j where j = i until $j * i > n$. else, skip elem. (this is the 'sieve'/'filter' part)

3 the tracker array stores all primes up to n.

2.8.1 To find unique prime factorization of n:

finding prime factorization with sieve of Eratosthenes Construct the sieve and for each composite number c, store the smallest prime number p s.t $k * p = c$
then, we can reconstruct each composite number out of its prime factors by doing n / p = new n and repeating till new n = 1.
The prime numbers used is the unique prime factorization of n.

2.8.2 Reusing sieve for multiple test cases

Check question constraints, possible to generate reusable prime numbers list.

2.8.3 Finding unique prime divisors with sieve:

we can count the number of unique prime divisors for composite numbers which is the number of times a composite number is visited when i * j occurs.
Be sure to initialise j from 2 instead of from i since we want to count all the visits. (then for the primes, the number of prime divisors = 1 by prime property)

2.8.4 Notes on prime

max number of primes per number is $\sqrt{n}$

2.9 Finding factors by reducing search space to $\sqrt{n}$

find all a and b such that $a * b = n$
$min(a, b) \leq \sqrt{n}$ (proof is intuitive and can be found here: Proof)
hence, only check for divisibility from 1 till $\sqrt{n}$ and the corresponding factor to find all factors.

2.10 Modular Arithmetic

Also known as clock arithmetic, where %12 is used.

2.10.1 Congruence:

For $a \equiv b \pmod{x}$,

1. same remainder mod x
2. x divides $(a - b)$

Addition, Multiplication, Exponentiation and Division properties:

2.11 Fibonacci numbers are periodic modulo any number:

e.g. the period for fibo sequence mod 10 is 60. this means every 60 elements, the same sequence repeats itself. Different periods for different mod n but they will eventually repeat.

3 Data Structures

3.1 Disjoint Set Problems - maintain sets which can be combined and check if they are connected:

Union-find: 2 methods provided (find - check if connected and union - combine 2 sets)
for log(n) find and union operations, use weighted union-find which links the smaller set to the larger set. implement with 2 int arrays. (if non integer values, use hashing + open addressing)
Use path compression with grandparent assigning strategy for better performance.

Java method implementation:

```
int getParent(int a, int[] componentId) {
                while (a != componentId[a]) {
                    // path compression by setting to grandparent
                    componentId[a] = componentId[componentId[a]];
                    a = componentId[a];
                }
                return a;
            }

            boolean find(int a, int b, int[] componentId) {
                return getParent(a, componentId) == getParent(b, componentId);
            }

            void union(int a, int b, int[] componentId, int[] size) {
                int pa = getParent(a, componentId);
                int pb = getParent(b, componentId);
                // weighted union for log(n) time complexity
                if (size[pa] > size[pb]) {
                    componentId[pb] = pa;
                    size[pa] = size[pa] + size[pb];
                } else {
                    componentId[pa] = pb;
                    size[pb] = size[pb] + size[pa];
                }
            }
```

3.2 Heap

4 Techniques

4.1 Complete search/Backtracking

Idea: generate all possible solutions to the problem (solution space) using brute force, then select/count the optimal solution.
Usually involves recursion (or iteratively).
Many backtracking implementations are actually just DFS implemented.

If no TLE concerns, can solve almost any problem.
Otherwise, optimise with greedy or find the optimal substructure(recursive definition) for DP.

Applications: generating subsets or permutations

To generate the search cases, we consider the cases (e.g. either include or dont include) and generate all possible solutions

for subsets, we can represent each subset as a binary sequence
Generating subsets containing 0 to n iteratively:

```
for (int b = 0; b < (1 << n); i++) {
// b is the binary sequence representing a subset
}

// generate the subset from binary sequence
// e.g. 1001 is {0, 3}
```

```
vector<int> subset;
for (int i = 0; i < n; i++) {
if(b \& (1 << i)) subset.push_back(i);
}
```

Generating permutations containing 0 to (n-1) recursively:

```
#include<vector>
using namespace std;

int searchPerm(vector<int>& perm, vector<int>& chosen, int n) {
    if (perm.size() == n) {
        // perm generated
    } else {
        for (int i = 0; i < n; i++) {
            if (chosen[i]) continue;
            chosen[i] = true;
            perm.push_back(i);
            // search assuming i added at this position
            searchPerm(perm, chosen, n);

            // else i not added in this position, add another (next) elem at
            // current position
            chosen[i] = false;
            perm.pop_back();
        }
    }
}

int main() {
    vector<int> perm;
    int n = 5;
    // form perm including {0, 1, 2, 3, 4}
    vector<bool> chosen;
    for (int i = 0; i < n; i++) {
        chosen[i] = false;
    }
}
```

4.1.1 Backtracking:

Backtracking is a form of complete search:
It begins with an empty solution and then extends the solution step-by-step checking all step possibilities if its valid and then forming a solution of valid steps.
See Coin change, N-Queens and Word Search problems.

Time complexity is usually exponential.
Hence, we want to **optimise via pruning the search space and noticing ASAP when a partial solution cannot be extended to a full solution and pruning that out.**

Another technique is the **Meet in the middle** technique which works in some cases.

Idea is to split the set into 2 and perform a complete search in both, then combine them together. This reduces the search space from $2^n to 2^(n/2)$

e..g. to find the subsets that sum up to n. we can split the set into 2 sets, find the subsets and corresponding sums of each set, then solve via 2-sum problem instead of generating 2^n subsets.

4.2 Dynamic Programming

key is to find the recursive formulation/recurrence + transition (ie. **Optimal substructure**
may not always be easy to find this formulation
then memoize ie. **Overlapping subproblems**

Example of possible recurrence:
Let dp(x) be ...
dp(0) = 0 (base case)
dp(x) = dp(x-1) + a[x]

Applications: used for optimisation(may be asked to reconstruct the optimal solution) and counting questions

```
class Solution {
    public int lengthOfLIS(int[] nums) {
        // idea is d[i] = smallest elem s.t i is the LIS
        // d[i] = elem after d[i - 1] s.t d[i] > d[i - 1]
        // what if d[i - 1] is updated to something smaller? d[i] will still hold true for earlie
        // otherwise d[i] is updated and must hold true for newest d[i - 1]
        if (nums.length <= 0) {
            return 0;
        }
        int[] d = new int[nums.length];
        for (int i = 0; i < nums.length; i++) {
            d[i] = 10001;
        }
        int len = 0; // +1 to get actual
        d[0] = nums[0];

        for (int i = 1; i < nums.length; i++) {
            int elem = nums[i];
            // check if can append to end
            if (elem > d[len]) {
                // System.out.println(elem);
                len++;
                d[len] = elem;
            }
            else {
                // find index of largest x < elem, update the one next to it
                // d[i - 1] < d[i] hence can use bn search
                int lo = 0;
                int hi = len;
                while (lo < hi) {
                    int mid = lo + (hi - lo) / 2;
                    // 1 2 6 7 8 -> 3 find smallest larger than 3
                    if (d[mid] > elem) {
                        hi = mid;
                    } else {
                        lo = mid + 1;
                    }
```

```
                }

                // deal with duplicates - dont update if a duplicate exists
                if ((lo + 1 <= len && (d[lo + 1] == elem) || d[lo] == elem) || (lo - 1 >= 0 && d[

                }
                else if (d[lo] > elem) {
                    d[lo] = elem;
                } else if (lo + 1 <= len && d[lo + 1] > elem) {
                    d[lo + 1] = elem;
                }
            }
        }
        return len + 1;
    }
}
```

5 Sorting Algorithms

5.1 Counting sort

O(n+k) runtime

If range k of values is known, we can apply counting sort to sort fast!

1 iterate through the elems and count the occurrence of each elem.

2 store the cumulative count of elems ¡ i and i itself.

3 iterate through each list elem, lookup the count from (2), position (one-based) = stored count.

4 place list elem into position and decrement the stored count from (2) by 1.

The intuition for steps 3 and 4 are that step 2 stores the position of the last elem with the value, hence once we place list elem, we decrement the position for the next one.

6 Tutorials

6.1 CodeChef Problem Code:CHEFSQRS

Find min x which is a square which when added to n also results in a square.

Solving technique for math definition problems - convert question by expressing as an equation to solve

want to find m and x s.t

$$n + m * m = x * x \tag{5}$$
$$n = (x * x) - (m * m) \tag{6}$$
$$n = (x - m) * (x + m) \tag{7}$$

want the min square, so want min m. find factors of n = a * b such that a - b is min. hence, start closest to the $\sqrt{n}$ since the diff between factors will be the min.

6.2 CodeChef Problem Code:CHEFADV

either we use the power up or dont use the power up.
after which, the solve() function is the same just for different arguments.
so we can make a solve() function with 2 different param and check if either is solved.
e.g. solve(1, 1) OR solve(2, 2)

6.3 Question 1(Watermelon 800):

Concept: Parity of addition of odd and even numbers

Let n be even if $n = 2k$. Let n be off if $n = 2k + 1$.

Hence,

- for x,y are odd, $x + y = 2w + 1 + 2z + 1 = 2(w + z + 1)$, hence x + y is even (by definition of even numbers)
- for x is even and y is odd, $x + y = 2w + 2z + 1 = 2(w + z) + 1$, hence x + y is odd (by definition of odd numbers)
- for x,y are even, $x + y = 2w + 2z = 2(w + z)$, hence x + y is even (by definition of even numbers)

Concept: LSB of odd numbers is 1.

Let x be an odd number and y be an even number. Use this property to check parity fast.

$$x\&0b1 = 1 \tag{8}$$
$$y\&0b1 = 0 \tag{9}$$

6.4 Question 2(Weights Assignment for Tree Edges 1500):

n-node trees have n-1 edges. There is a unique path between root to all nodes.

We can augment the tree with a corresponding array ie. index i stores weight of node i.

The question wants an increasing path weight for each node in the given order. hence, we can pre-allocate the path weights.

Since a unique path exists to the root to all nodes, hence, dist[root, node i] = dist[root, node i's parent] + weight[i, i's parent]

Hence, to determine each edge weight, just take the pre-allocated distances and minus to get the weight of the edge.

To check if ordering is invalid, check if the $dist[root, nodei'sparent] > dist[root, nodei]$ which is impossible since i's parent to root is a subpath of i to root.

6.5 Question 3(Escape the maze hard 1900):

Simply perform BFS on all friends and vlad.
If vlad's new frontier is empty, then no path can be found to a leaf.
Else, if vlad's new frontier has a new node that has only 1 neighbour i.e. a leaf, then the path has been found.

To count the minimum number of friends needed, simply count the number of LCA. We know that Vlad will **only ever encounter LCA during his traversal.** Hence, number of LCA = number of times vlad visits a node visited by a friend = min friends.

6.6 Question 4(ATM and Students 1800)

Initially, i attempted this question using a modified Zadane's algorithm - but this didn't consider that the solution must be locally optimal at all times. e.g. ATM bal = 0, (-5, 5) is invalid despite the total sum >= which seems valid.

6.7 CF763Div2 B: GameOnRanges:

Notice a few heuristics:

1. Each element from 1 to n must be chosen exactly once.

2. Each bigger range comprises of each smaller range.

Hence, we can sort the ranges by size. For the smallest range with only 1 element, choose that element. Then, choose in the bigger ranges what the smallest elements have not yet chosen.

To implement, use a TreeSet to store the elements from 1 to n. Sort the ranges from min length to max length. Iterate through ranges and get the ceiling(left) or floor(right) = d for each range, both work!

Make sure to remove ' package ' when submitting Java solutions to the judge, it will fail otherwise. (Cost me over 5 hours of debugging, but now I know)

6.8 CF763Div2 C: Balanced Stone Heaps

Trying to find the min (optimisation problem)
Hence, in the toolbox, we can use DP or binary search.

Notice that if we can find possible x for min value of smallest heap, then y ¡ x is also possible. Hence, this fulfils binary search condition.
Simply perform binary search for largest x.

The challenge now is how to check if each x is valid?
Idea: **start from right to left**. Why? at the rightmost, the value of each heap cannot be changed. we can set an 'x' and then greedily donate from heap i such that at least x left.
However, theres another catch. What if initially the value at the heap is smaller than donated i - x?
hence, we can only pass the min between original i and (donated i - x)

Proof that each index greater or equals 2 will always have at least x:
we need to check that for each heapAfter[i] greater or equals min
if give heapAfter[i] - x, will always have at least x.
if give original, that means original ¿ heapAfter[i] - x, hence, will have ¿ x left.
hence, will always have at least x.

```
static boolean isValidMin(int[] heaps, int n, int min) {
       // how to check if min is valid
       // start from back
       int[] heapAfter = heaps.clone();
       for (int i = n - 1; i >= 2; i--) {
           if (heapAfter[i] < min) {
               return false;
           }
           // how much to give? as much st. still have k
           int toGive = Math.min(heapAfter[i] - min, heaps[i]);
```

```
            int d = toGive / 3;
            heapAfter[i - 1] = heapAfter[i - 1] + d;
            heapAfter[i - 2] = heapAfter[i - 2] + 2 * d;
        }
        return heapAfter[0] >= min && heapAfter[1] >= min;
    }
```

6.9 Goodbye2021 B: Mirror on a String

find smallest lexicographical string after mirror.

Heuristic: Find longest decreasing sequence, if duplicates, see the following 2 cases.
Quite a simple question with some edge cases which may not be easy to find if you dont test all the possible scenarios.

if only 1 decreasing, then make the string shortest.
e.g. 'aabab' : 'aa' is smaller than 'aaaa'

if more than 1 decreasing, then make the string longest. e.g. 'cbbbaaad': 'cbbaaaaaabbc' is smaller than 'cbbaabbc'.

Proof that solution works: first part of mirror will always be a prefix of original string
second part we want to have the smallest, hence, we want it to be as small as possible hence increasing as index grows (hence, the first part has to be decreasing order as second part is a mirror)
if the decrement has ¿1 unique character, then we want to make the increasing subsequence as late as possible, hence find longest duplicate.
if only 1, then the mirror will not be increasing but constant, hence we make it as short as possible.

```
    public static void main(String[] args) {
        FastScanner fs = new FastScanner();

        long t = fs.nextInt();
        while(t-- >0) {
            // get input
            int n = fs.nextInt();
            String s = fs.next();
            int i = 0;
            while (i + 1 < n && s.charAt(i) > s.charAt(i + 1)) {
                i++;
                while (i + 1 < n && s.charAt(i) == s.charAt(i + 1)) {
                    i++;
                }
            }
            System.out.println(new StringBuilder(s.substring(0, i + 1)).append(new StringBuilder(s
        }
    }
```