

Range queries

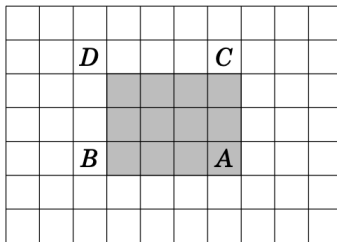
- **How to compute sum between two items in an array?**
- **How to compute minimum between two items?**
- Cool reference: <https://cses.fi/book/book.pdf>

Sums: cumulative sums

- Compute a prefix sum array:

```
arr = [1, 3, 4, 8, 6, 1, 4, 2]
cum_sum = arr[:]
for i in range(1, len(arr)):
    cum_sum[i] = cum_sum[i - 1] + arr[i]
print(cum_sum)
# 0 1 2 3 4 5 6 7
# [1, 4, 8, 16, 22, 23, 27, 29]
def sum(i, j):
    return cum_sum[j - 1] - cum_sum[i - 1]
sum(3, 7) # 19
```

- Higher dimensions:



The sum of the gray subarray can be calculated using the formula

$$S(A) - S(B) - S(C) + S(D),$$

Sums

- Example: <https://qoj.ac/problem/5585>

```
# Read all input from standard input efficiently
input_data = [i.split() for i in input()]
n = int(input_data[0][0])
l = int(input_data[0][1])
h = int(input_data[0][2])

# Read daily profits
profits = [int(x) for x in input_data[1:]]

# Build prefix sum array (1-indexed for convenient interval queries)
pref = [0] * (n + 1)
for i in range(n):
    pref[i + 1] = pref[i] + profits[i]
m, M = float('-inf'), -float('inf')
for x in range(l, h + 1):
    for shift in range(1, x+1):
        cur = 0
        # 1. Handle the leading segment (if shift > 0)
        if pref[shift] > 0:
            cur += 1
        # 2. Handle the full segments
        while shift < n:
            if pref[min(shift + x, n)] - pref[shift] > 0:
                cur += 1
            shift += 1
        m, M = min(m, cur), max(M, cur)
print(f"{m} {M}")
```

Minimum queries

- What is minimum in an array range: $\min_q(a, b)$?
 - Range minimim query
- Sparse tables
 - Precompute $\min_q(a, b)$ for all subarrays whose length is power of two $\min_q(a, b) = \min(\min_q(a, a + w - 1), \min_q(a + w, b))$ where $b - a + 1$ is a power of two and $w = \frac{b - a + 1}{2}$.
 - $O(n \log n)$ processing time (powers of 2)
 - Then, $\min_q(a, b) = \min(\min_q(a, a + k - 1), \min_q(b - k + 1, b))$.
- Practice: <https://www.spoj.com/problems/RMQSQ/>

```
arr = [1, 3, 4, 8, 6, 1, 4, 2]

# log[i] = floor(log2(i))
log = [0] * (len(arr) + 1)
for i in range(2, len(arr) + 1):
    log[i] = log[i // 2] + 1

# st[k][i] = minimum of arr[i : i + 2^k]
st = [[0] * len(arr) for _ in range(log[-1] + 1)]
st[0] = arr[:]
for k in range(1, log[-1] + 1):
    length = 1 << k      # 2 ** k
    half = length >> 1   # len / 2
    for i in range(len(arr) - length + 1):
        st[k][i] = min(st[k - 1][i], st[k - 1][i + half])

# st:
# [[1, 3, 4, 8, 6, 1, 4, 2],
#  [1, 3, 4, 6, 1, 1, 2, -],
#  [1, 3, 1, 1, 1, -, -, -],
#  [1, -, -, -, -, -, -, -]]

def query(left, right):
    length = right - left
    k = log[length]
    return min(st[k][left], st[k][right - (1 << k)])

print(query(1, 5))
print(query(3, 7))
print(query(1, 7))
```

Dynamic arrays

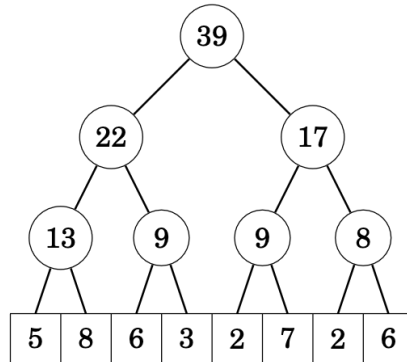
- All these are **static**: array does not change
- For dynamic arrays, you must use BITs (Fenwick trees)
 - Check <https://cses.fi/book/book.pdf>
 - Can give a lecture if smo is interested later on
 - Div 1 only

Segment trees

- A cumulative sum but implemented as a binary tree: each node keeps the sum of the children leaves

0	1	2	3	4	5	6	7
5	8	6	3	2	7	2	6

The corresponding segment tree is as follows:



- Binary, hence $O(n \log n)$ reads/updates
- You can actually implement it as an array of $2n$ elements!

```
arr = [5, 8, 6, 3, 2, 7, 2, 6]
tree = [0] * (2 * len(arr)) # assume len(arr) is power of 2
for i in range(len(arr)): # leaves
    tree[len(arr) + i] = arr[i]
for i in range(len(arr) - 1, 0, -1): # internal nodes
    tree[i] = tree[2 * i] + tree[2 * i + 1]
```

Segment trees

- Updates are simple:

```
def update(idx, val):  
    pos = index + len(arr)  
    tree[pos] = val  
    pos //= 2  
    while pos >= 1:  
        tree[pos] = tree[2 * pos] + tree[2 * pos + 1]  
        pos //= 2
```

- Queries as well:

```
def query(left, right):  
    left += len(arr)  
    right += len(arr)  
  
    ans = 0  
    while left <= right:  
        if left % 2 == 1:  
            ans += tree[left]  
            left += 1  
        if right % 2 == 0:  
            ans += tree[right]  
            right -= 1  
        left //= 2  
        right //= 2  
    return ans
```

Problem I: Salary Inequity

- <https://acmicpc-pacnw.org/ProblemSet/2014/div1.pdf>
- Given a tree of employees (defining manager/subordinate relationship) and their salaries, quickly answer:
 - Difference of employee's min-max salaries for each manager
 - Update salary of manager and each employee

```
def dfs(u, tin, tout, order, pos):
    tin[u] = pos
    order[pos] = u
    for v in children[u]:
        pos = dfs(v, pos + 1)
    tout[u] = pos - 1
    return pos

T = int(input())
for _ in range(T):
    n = int(input())
    children = [[] for _ in range(n)]
    parents = list(map(int, input().split()))
    for v in range(1, n):
        p = parents[v - 1] - 1 # adjust to 0-based indexing
        children[p].append(v)
    salary = list(map(int, input().split()))

    # Flatten the tree!
    tin = [0] * n # node -> pos; position where you entered node
    tout = [0] * n # position where you left node
    order = [0] * n # reverse index: pos -> node
    pos = dfs(0, tin, tout, order, 0)

    flat = [0] * n
    for i in range(n):
        flat[i] = salary[order[i]]

    # ...next slide...
```

Problem I: Salary Inequity

```
def build(node, l, r):
    if l == r:
        mn[node] = mx[node] = flat[l]
        return
    m = (l + r) // 2
    build(node * 2, l, m)
    build(node * 2 + 1, m + 1, r)
    mn[node] = min(mn[node * 2], mn[node * 2 + 1])
    mx[node] = max(mx[node * 2], mx[node * 2 + 1])

for _ in range(T):
    # ...see previous slide...

    # min-max segment trees
    mn = [0] * (4 * n) # 4 because n might not be power of 2!
    mx = [0] * (4 * n)
    lazy = [0] * (4 * n) # lazy updates
    build(1, 0, n - 1)

    q = int(input())
    for _ in range(q):
        parts = input().split()
        if parts[0] == "R":
            u, x = int(parts[1]) - 1, parts[2]
            add(1, 0, n - 1, tin[u], tout[u], x)
        else:
            u = int(parts[1]) - 1
            lo, hi = query(1, 0, n - 1, tin[u], tout[u])
            print(hi - lo)
```

Problem I: Salary Inequity

```
def lazy_push(node):
    if lazy[node]:
        x = lazy[node]
        for child in (node * 2, node * 2 + 1):
            mn[child] += x; mx[child] += x; lazy[child] += x
        lazy[node] = 0

def add(node, l, r, ql, qr, x):
    if r < ql or qr < l: return
    if l >= ql and r <= qr:
        mn[node] += x; mx[node] += x; lazy[node] += x
        return
    lazy_push(node)
    m = (l + r) // 2
    add(node * 2, l, m, ql, qr, x)
    add(node * 2 + 1, m + 1, r, ql, qr, x)
    mn[node] = min(mn[node * 2], mn[node * 2 + 1])
    mx[node] = max(mx[node * 2], mx[node * 2 + 1])

def query(node, l, r, ql, qr):
    if r < ql or qr < l: return 10**30, -10**30
    if l >= ql and r <= qr: return mn[node], mx[node]
    lazy_push(node)
    m = (l + r) // 2
    a1, b1 = query(node * 2, l, m, ql, qr)
    a2, b2 = query(node * 2 + 1, m + 1, r, ql, qr)
    return min(a1, a2), max(b1, b2)
```

Union find

- A union-find structure maintains a collection of sets. The sets are disjoint, so no element belongs to more than one set.
- Two nearly-constant time operations are supported:
 - the **union** operation joins two sets
 - the **find** operation finds the representative of the set that contains a given element
- Each member points to another member of the set (root) or itself (set size = 1)
- We keep the size of set for each set
- Implementation:
 - **Find**: just find the root of the set
 - **Is same?**: check if roots match
 - **Union**: point smaller set to the larger set



- `find(8)` is 2
- `same(2, 6)` is true

Union find

```
# we have n elements
link = [i for i in range(n)]
size = [1 for i in range(n)]

def find(x): # just find the root of the set
    if x == link[x]: return x
    return find(link[x])

def same(x, y):
    return find(x) == find(y)

def union(x, y):
    x, y = find(x), find(y)
    if size[x] < size[y]: x, y = y, x
    size[x] += size[y]
    link[y] = x
```

- **Problem:** you might end up with long chains, making `find` linear ($O(n)$)!

Union find: path compression

- Trick: anytime you see a node, point it to the root

```
# we have n elements
link = [i for i in range(n)]
size = [1 for i in range(n)]

def find(x): # just find the root of the set
    if x == link[x]: return x
    link[x] = find(link[x]) # yup, just this line!
    return link[x]
```

Virtual Friends

- <https://open.kattis.com/contests/rva2u2/problems/virtualfriends>
- Just do union find and print size after each union

These days, you can do all sorts of things online. For example, you can use various websites to make virtual friends. For some people, growing their social network (their friends, their friends' friends, their friends' friends' friends, and so on), has become an addictive hobby. Just as some people collect stamps, other people collect virtual friends.

Your task is to observe the interactions on such a website and keep track of the size of each person's network.

Assume that every friendship is mutual. If Fred is Barney's friend, then Barney is also Fred's friend.

Input

The first line of input contains one integer specifying the number of test cases to follow. Each test case begins with a line containing a positive integer F , the number of friendships formed. Each of the following F lines contains the names of two people who have just become friends, separated by a space. A name is a string of 1 to 6 letters (uppercase or lowercase). The sum of F over all test cases in the file is at most 100 000.

Output

Whenever a friendship is formed, print a line containing one integer, the number of people in the social network of the two people who have just become friends.

Sample Input 1

```
1
3
Fred Barney
Barney Betty
Betty Wilma
```

Sample Output 1

```
2
3
4
```

Lone Knight: <https://qoj.ac/problem/5597> (Div1)

- You have lots of rooks and knights
- Can you jump from cell X to cell Y with a knight without being attacked by a rook? You need to answer many queries!
- **Nobody solved this problem!** 🤖
- **Solution:**
 - Rooks cut infinite board to rectangles (sets!)
 - Max. $O(n^2)$ rectangles (n is below 1000)
 - Key trick: transform rectangles to sets!