2021 Mini Competition 1 Editorial

2021-03-13

香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

M2111
Marshmallows

Subtask 1

- N = 1
- There is only 1 chef
  - The second number in each output line is always 1 (chef number 1)
- $W <= 10^6$, $T_i <= 10^9$
- How do we find the position of the marshmallow efficiently?

Observation 1

- Cycle width = 2W
- Since W <= 10^6, we can simulate the first cycle and store the position of the chef for t = 0 .. 2W-1
- The answer for query T_i will be same as the answer for T_i % (2W)
- 9 points

t	1	2	3	4	5	6
0			1			
1				1		
2					1	
3						1
4						1
5					1	
6				1		
7			1			
8		1				
9	1					
10	1					
11		1				
12			1			
13				1		
14					1	
15						1
16						1
17					1	
18				1		
19			1			
20		1				
21	1					
22	1					
23		1				
24			1			
25				1		
26					1	
27						1
28						1

Subtask 2

- $X = 1 \quad P_1 = 1 \quad Y = 0$
- Initially, the fork with a raw marshmallow is held by the chef at cell 1
- Observe that the cycle length is still $2W = 16$

t	1	2	3	4	5	6	7	8
0	1			2			3	
1		1		2			3	
2			1	2			3	
3			1	2			3	
4			1		2		3	
5			1			2	3	
6			1			2	3	
7			1			2		3
8			1			2		3
9			1			2	3	
10			1			2	3	
11			1		2		3	
12			1	2			3	
13			1	2			3	
14		1		2			3	
15	1			2			3	
16	1			2			3	
17		1		2			3	
18			1	2			3	
19			1	2			3	
20			1		2		3	
21			1			2	3	
22			1			2	3	
23			1			2		3
24			1			2		3
25			1			2	3	
26			1			2	3	
27			1		2		3	
28			1	2			3	
29			1	2			3	
30		1		2			3	
31	1			2			3	
32	1			2			3	
33		1		2			3	
34			1	2			3	
35			1	2			3	
36			1		2		3	
37			1			2	3	

Observation 2

- Even though the chefs stopped when they hit another chef, the marshmallow does not stop
- The marshmallow, just like N = 1, remains to be a triangular wave pattern
- Therefore, instead of tracking the chefs' position, we only need to track the marshmallow's position
    - The task does not ask you to output the chefs' position

t	1	2	3	4	5	6	7	8
0	1			2			3	
1		1		2			3	
2			1	2			3	
3			1	2			3	
4			1		2		3	
5			1			2	3	
6			1			2	3	
7			1			2		3
8			1			2		3
9			1			2	3	
10			1			2	3	
11			1		2		3	
12			1	2			3	
13			1	2			3	
14		1		2			3	
15	1			2			3	
16	1			2			3	
17		1		2			3	
18			1	2			3	
19			1	2			3	
20			1		2		3	
21			1			2	3	
22			1			2	3	
23			1			2		3
24			1			2		3
25			1			2	3	
26			1			2	3	
27			1		2		3	
28			1	2			3	
29			1	2			3	
30		1		2			3	
31	1			2			3	
32	1			2			3	
33		1		2			3	
34			1	2			3	
35			1	2			3	
36			1		2		3	
37			1			2	3	

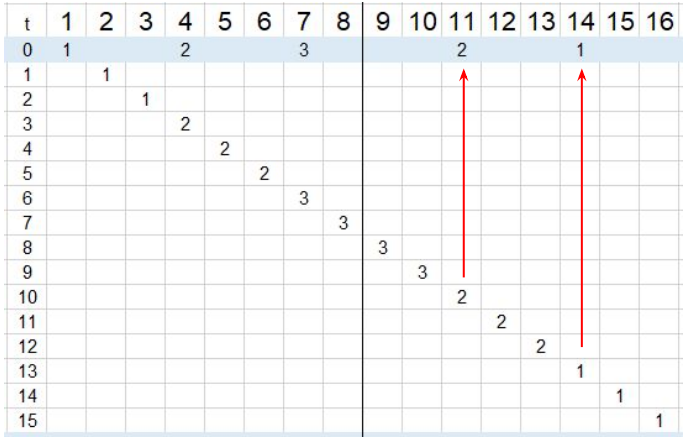
Observation 3

- The movement of a chef is limited by the positions of the previous and next chefs
- In this subtask, Chef **i** can only move between P_i and P_{i+1}-1 inclusive. The last chef can move between P_N and W inclusive.
- Since W <= 10^6, we can simulate the first cycle and store the position of the marshmallow and the chef ID for t = 0 .. 2W-1
- Alternatively, once we know the position and direction, we can **binary search** for the chef ID

t	1	2	3	4	5	6	7	8
0	1			2			3	
1		1		2			3	
2			1	2			3	
3			1	2			3	
4			1		2		3	
5			1			2	3	
6			1			2	3	
7			1			2		3
8			1			2		3
9			1			2	3	
10			1			2	3	
11			1		2		3	
12			1	2			3	
13			1	2			3	
14		1		2			3	
15	1			2			3	
16	1			2			3	
17		1		2			3	
18			1	2			3	
19			1	2			3	
20			1		2		3	
21			1			2	3	
22			1			2	3	
23			1			2		3
24			1			2		3
25			1			2	3	
26			1			2	3	
27			1		2		3	
28			1	2			3	
29			1	2			3	
30		1		2			3	
31	1			2			3	
32	1			2			3	
33		1		2			3	
34			1	2			3	
35			1	2			3	
36			1		2		3	
37			1			2	3	

Trick 1

- How to binary search for the chef ID when the cooked marshmallow is travelling from right to left?
- We can append the opposite direction to the binary search array
- For chefs i to N-1, append
  pos = $2W-P_{i+1}+2$, id = i
- Remember to either insert in reverse order, or sort the array again



t	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1			2			3				2			1		
1		1														
2			1													
3				2												
4					2											
5						2										
6							3									
7								3								
8									3							
9										3						
10											2					
11												2				
12													2			
13														1		
14															1	
15																1

Subtasks 3, 4, 5

- These subtasks give points to inefficient simulation solutions
- O(NQT): 17 points
- O(N+QT): 39 points
- O(N+Q+T): 50 points

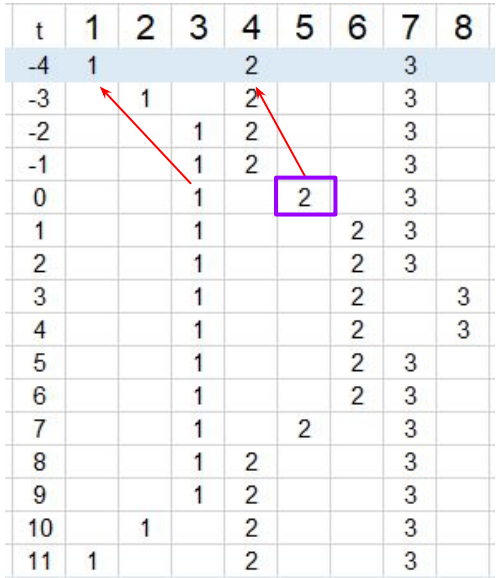
Full solution

- Since the input is in the middle of a cycle, it is no longer true that only move between P_i and P_{i+1}-1 inclusive
- We can no longer directly build an array to binary search for the chef id
- Still, we can simulate the first cycle and store the position of the marshmallow and the chef ID for t = 0 .. 2W-1
- O(W+Q) will get you 100 points

t	1	2	3	4	5	6	7	8
0			1		2		3	
1			1			2	3	
2			1			2	3	
3			1			2		3
4			1			2		3
5			1			2	3	
6			1			2	3	
7			1		2		3	
8			1	2			3	
9			1	2			3	
10		1		2			3	
11	1			2			3	
12	1			2			3	
13		1		2			3	
14			1	2			3	
15			1	2			3	
16			1		2		3	
17			1			2	3	
18			1			2	3	
19			1			2		3
20			1			2		3

Trick 2

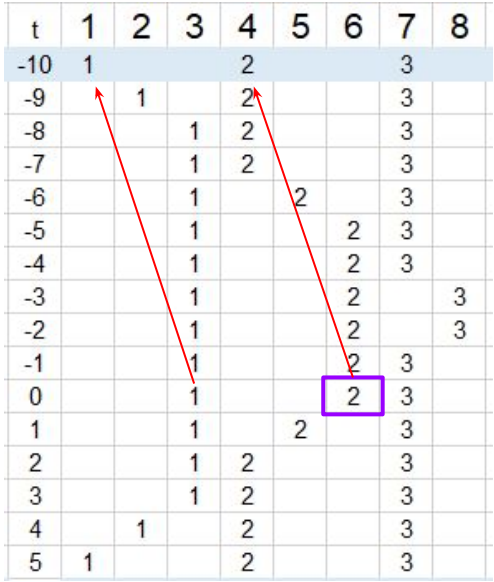
- What if the constraints are modified to W <= 10^9?
- To achieve O(N+QlgN), we can use the binary search solution in Subtask 2 by **rebasing**.
- We rewind the state until Chef 1 is at position 1 with an raw marshmallow.
- Assuming Y = 0 (raw):
  - Chef 1 will have $P_i = 1$
  - Chef i = 2..X will have $P_i = P_{i-1}+1$
- Also, increase all query timestamps by P_X-1



t	1	2	3	4	5	6	7	8
-4	1			2			3	
-3		1		2			3	
-2			1	2			3	
-1			1	2			3	
0			1		2		3	
1			1			2	3	
2			1			2	3	
3			1			2		3
4			1			2		3
5			1			2	3	
6			1			2	3	
7			1		2		3	
8			1	2			3	
9			1	2			3	
10		1		2			3	
11	1			2			3	

Trick 2

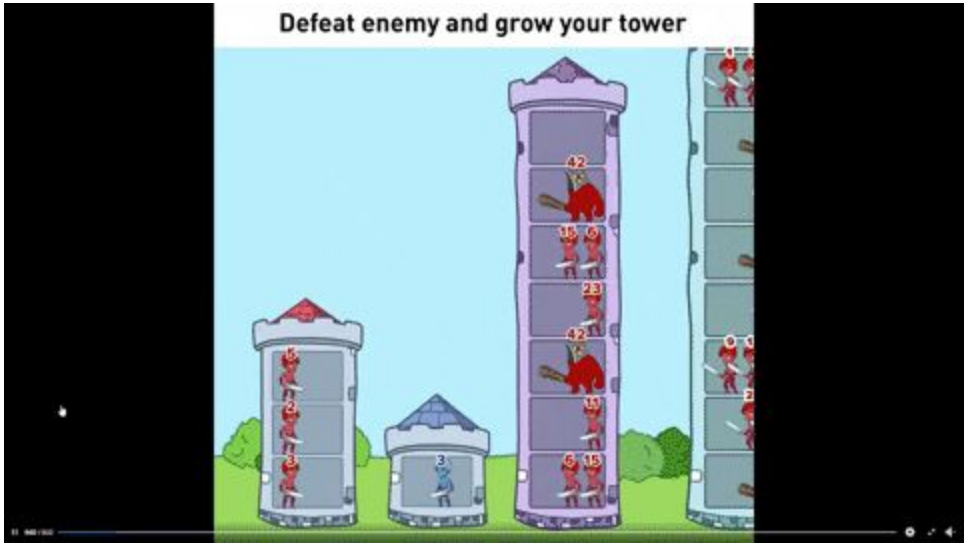
- What if Y = 1 (cooked)?
  - Chef 1 will have $P_i = 1$
  - Chef i = 2..X will have $P_i = P_{i-1}+1$
  - Same as Y = 0 :)
- Increase all query timestamps by $2W-P_X$



t	1	2	3	4	5	6	7	8
-10	1			2			3	
-9		1		2			3	
-8			1	2			3	
-7			1	2			3	
-6			1		2		3	
-5			1			2	3	
-4			1			2	3	
-3			1			2		3
-2			1			2		3
-1			1			2	3	
0			1			2	3	
1			1		2		3	
2			1	2			3	
3			1	2			3	
4		1		2			3	
5	1			2			3	

M2112
Annoying Advertisements

Background



Subtask 1

P is sorted

Obviously, the order [1, 2, 3, ..., N] is optimal.

Simply simulate the process to check if it is possible or not.

O(N)

Possible Careless Mistake

Forgot to use 64 bit integer for accumulating the power

Subtask 2

$1 \leq N \leq 20$

Exhaust all possible permutations?

- O(N * N!), results in TLE

Subtask 2

There are too many invalid permutations (e.g. 12543, 13524, 54312, etc).

Instead of exhausting permutations, we try to start at every floor from 1 to N and simulate to check if it is possible to beat all the monsters.

After beating the monster at current floor, we can choose to go to upper floor or lower floor.

We can exhaust all possible orders of going up / down.

Subtask 2

It can be implemented with recursion, but using bitwise operations is generally easier and faster.

- For more information about bitwise operations, attend **DP(II)** training session or check past years' slides

$O(N^2 * 2^N)$ or $O(N * 2^N)$

Possible Careless Mistakes

Forgot to use 64 bit integer for accumulating the power

Forgot to reset initial power for every possible order

Forgot to reset initial power for every possible starting floor

Subtask 3

$1 \le N \le 4000$

In fact, we can greedily choose the smaller one in upper / lower floor of each round for every starting floor.

$O(N^2)$

P.S. Personally I think this greedy solution is much more intuitive than the exhaustion one XD

Possible Careless Mistakes

Forgot to use 64 bit integer for accumulating the power

Forgot to reset initial power for every possible starting floor

Full Solution

Instead of trying all N possible starting floors, how can we find a possible starting floor (or determine it is impossible) faster?

Observation 1

Consider the followings, where P_K is the largest:

P_1	P_2	...	P_{K-1}	P_K	P_{K+1}	...	P_{N-1}	P_N

If P_0(initial power) + P_1 + P_2 + ... + P_{K-1} < P_K and P_0 + P_{K+1} + P_{K+2} + ... + P_N < P_K, we know that the answer must be impossible.

How about if $P_0 + P_1 + P_2 + ... + P_{K-1} \geq P_K$ or $P_0 + P_{K+1} + P_{K+2} + ... + P_N \geq P_K$?

Observation 1

Let say if we can beat the monsters from 1 to K - 1,

and $P_0 + P_1 + P_2 + \ldots + P_{K-1} \geq P_K$, then we must be able to beat all the remaining monsters.

Then how do we know if we can beat the monsters from 1 to K - 1?

We repeat the process for P[1, 2, ..., K - 1]!

We stop when we find a $P_i \leq P_0$.

Pseudocode

```
Recursion(L, R)
  if L = R
    check p0 >= p[L]
  MAX = query_max(L, R)
  LSum = query_sum(L, MAX - 1)
  RSum = query_sum(MAX + 1, R)
  if (P0 + LSum >= MAX)
    Recursion(L, MAX - 1)
  if (P0 + RSum >= MAX)
    Recursion(MAX + 1, R)
```

Full Solution

The recursion will be executed at most O(N) times.

`query_sum`: can be done by partial sum

- Preprocessing: O(N)
- Query: O(1)

`query_max`: can be done by sparse table / segment tree (appear in DS(III))

- Preprocessing: O(N log N) (sparse table) / O(N) (segment tree)
- Query: O(1) (sparse table) / O(log N) (segment tree)

Overall: O(N log N)

Full Solution 2

Subtask 3's solution is $O(N^2)$ because we simply simulate from the beginning for every starting floor, which leads to O(N) simulation.

Let say if we start on floor i, loses after beating some monsters, and the farthest floors we beat are L (leftest) and R (rightest).

Full Solution 2

It is unnecessary to try starting from floor [L, R] as we already know it will fail.

We can try from R + 1.

Time Complexity?

Full Solution 2

For every P_i > Current Power, it will be visited at most O(1) times.

For every $P_i \leq$ Current Power, it will be visited at most O(log max{P_i}) times.

Overall: O(N log max{P_i})

M2113
Locating Light Poles

Subtask 1

Query all possible `measure(x,y)` for $0 \le x,y \le N-1$ and $x \ne y$

- $N(N-1)=90$ queries

Simplest idea is to exhaust and check if it fulfills all the information

- $O(N! \cdot N^2)$ results in TLE
- $O(2^N \cdot N^2)$ by only exhausting the signs of every P
  (as `measure(0,i)` tells the magnitude of `P[i]`)

Subtask 2

Query all possible `measure(x,y)` for $0 \le x, y \le N-1$ and $x \ne y$

- $N(N-1)=159600$ queries
- Find all pairs with distance = 1, they must be the only neighbors
- We can now form a chain
- Perform shift and/or reflection to satisfy $P[0]=0$ and $P[1]>0$
- Score: 7.036/12

We can actually reduce half the queries by only asking $x < y$

- $N(N-1)/2=79800$ queries
- Score: 12.000/12

Subtask 3

Query all possible `measure(x,y)` for $0 \le x < y \le N-1$ and $x \ne y$

- $N(N-1)/2=179700$ queries
- Find all pairs with distance = 2 or 3, they must be the only neighbors
- We can now form a chain
- Perform shift and/or reflection to satisfy $P[0]=0$ and $P[1]>0$
- Score: 6.592/13

Subtask 3

Instead of first asking all distances, we can only ask on demand

- Starts from `0`, keep querying an unreached `i`, i.e. `measure(0,i)`
- Repeat until a new neighbor is found
- Every time we find a new neighbor, we start from this point (say `cur`)
- Perform `measure(cur,i)` for unreached `i` until next neighbor found

- If until a point we could not find a neighbor, meaning we reach the end
- We can start from `0` again to find the other side (if needed)

Subtask 3

Use any data structure to maintain the unreached points, e.g. `std::set`

If we always ask the queries in the order of `1..N-1`

- Worst case need to perform `N(N-1)/2` queries:
- P's order = `0 N-1 N-2 N-3 .. 1`
- `(N-1)+(N-2)+(N-3)+...+1+0=179700` queries (Score: `6.592/13`)
- This case, and some order particular order cases, are included in the tests

Instead, if the query order is in random:

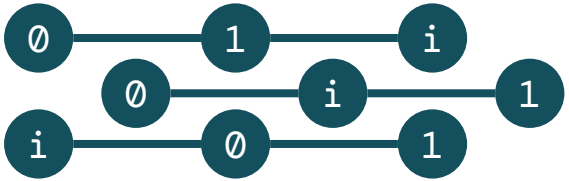
- When having X unreached points, expect `X/2` queries to find the neighbor
- `(N-1)/2+(N-2)/2+(N-3)/2+...+1/2+0=89850` queries (expected)
- Score: `13.000/13`

Subtask 4: 199997 queries

Perform 1 query `measure(0,1)` to determine `P[1]` (recall that `P[1] > 0`)

For each of the remaining `i = 2..N-1`:

- Perform 2 queries `measure(0,i)` and `measure(1,i)`
- 3 possible situations:
  - `d(0,1) + d(1,i) == d(0,i)`
  - `d(0,i) + d(1,i) == d(0,1)`
  - `d(0,1) + d(0,i) == d(1,i)`



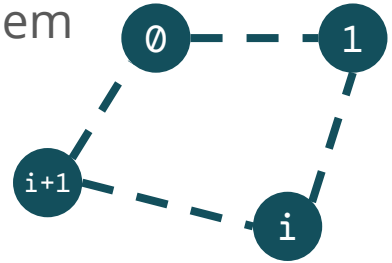
- Score: `61.270/100`

Subtask 4: 149998+ queries

Perform 1 query `measure(0,1)` to determine `P[1]`

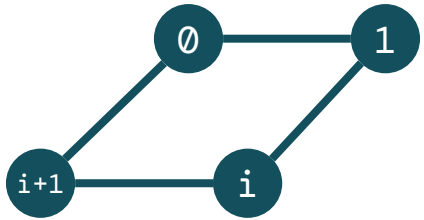
For each of the remaining `i` = 2..`N-1`:

- Group them as pairs `(2,3)`, `(4,5)`, ..., `(N-2,N-1)`, denote as `(i,i+1)`
- 3 queries: `measure(1,i)`, `measure(i,i+1)`, `measure(i+1,0)`
- For most of the scenarios, only one way to arrange them
- Except when:
  - `d(0,1)==d(i,i+1)`, and
  - `d(1,i)==d(0,i+1)`
  - i.e. forming a parallelogram



Subtask 4: 149998+ queries

- 3 queries: `measure(1,i)`, `measure(i,i+1)`, `measure(i+1,0)`
- For most of the scenarios, only one way to arrange them
- Except when:
  - `d(0,1)==d(i,i+1)`, and
  - `d(1,i)==d(0,i+1)`
  - i.e. forming a parallelogram



- Either:
  - `i+1` and `i` are on the left of `0` and 1 respectively
  - `i+1` and `i` are on the right of `0` and 1 respectively
- An extra query `measure(0, i)` uniquely determines the placement

Subtask 4: 149998^{+} queries

In best scenario, we perform 3 queries to locate 2 more points

- $1+49999\times3=149998$ queries (Score: `74.366/100`)

Worst case scenario, perform 3+1 queries to locate 2 more points

- $1+49999\times4=199997$ queries (Score: `61.270/100`)

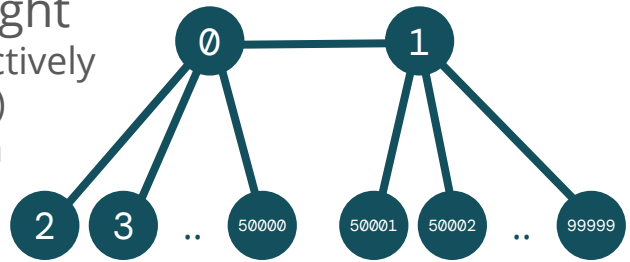
In fact, we can try to avoid worst case scenario by randomly selecting pairs

- It's unlikely to encounter parallelogram scenario
- Our solution in the worst test: `150000` queries (Score: `74.365/100`)

Subtask 4: 149998 queries

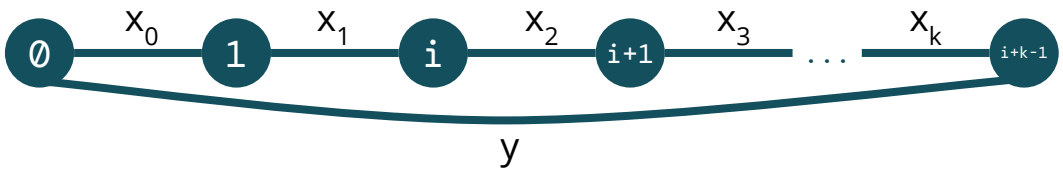
There exists a deterministic solution with exactly 149998 queries

- Similar idea to the previous solution
- Try to avoid parallelograms by first query on the followings:
- `measure(0,1)`, `measure(0,2..50000)`, `measure(1,50001..99999)`
- At most 2 points have the same distance from same point
  - Why? As the locations are distinct
- Find a way to pair up points on the left and right
  - s.t. no pairs have same distance from 0 and 1 respectively
  - Let's say pairing up x and y, then ask `measure(x,y)`
  - Guaranteed that none of them form a parallelogram



Subtask 4: $100000(1+1/K)^+$ queries

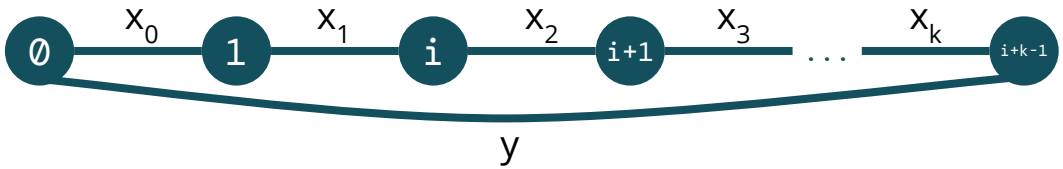
- Generalize the idea of “avoiding parallelogram”



- Ambiguity appears when $|x_0 \pm x_1 \pm x_2 \pm \ldots \pm x_k| = y = |x_0 \pm x_1 \pm x_2 \pm \ldots \pm x_k|$ with different lists of signs
- Before finding y, maintain a list of all possible $\{x_0 \pm x_1 \pm x_2 \pm \ldots \pm x_k\}$
  - and some necessary signals which help finding the list of signs given y
- Increase k until
  - the list will contains duplicates when k:=k+1 (to avoid ambiguity) OR
  - k==K (to avoid TLE)

Subtask 4: 100000(1+1/K)$^+$ queries

- Generalize the idea of “avoiding parallelogram”

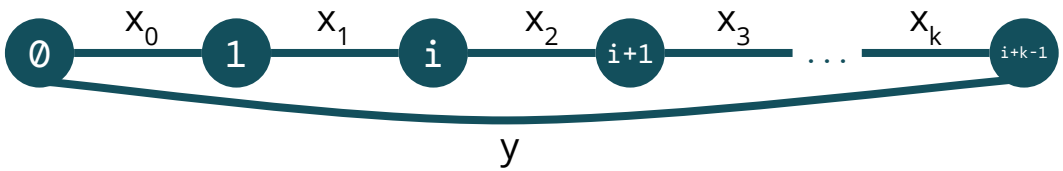


- Increase k until …
- Find y, i.e. `measure(0,i+k-1)`
- Deduce the corresponding sign (+ or -) for each x_j
- Calculate the corresponding `P[j]`

- Used k+1 queries for finding k new values of `P[]`

Subtask 4: 100000(1+1/K)+ queries

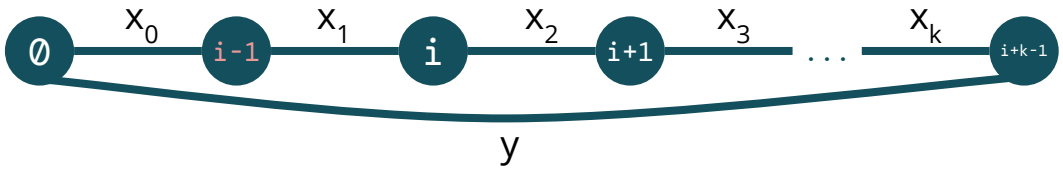
- Generalize the idea of “avoiding parallelogram”



- Time complexity: O(N/K * 2^K)
  - the log factor can be avoided by
    - keep merging 2 sorted lists instead of using `std::set`
- Number of queries >= 100000 * (1 + 1/K)
  - increase when k cannot reach K
- Our solution (K = 11): 110086 queries (Score: 96.430/100)
  - 100000(1 + 1/11) ≈ 109091

Subtask 4: 100000(1+1/K)$^+$ queries

- Generalize the idea of "~~avoiding~~ parallelogram"

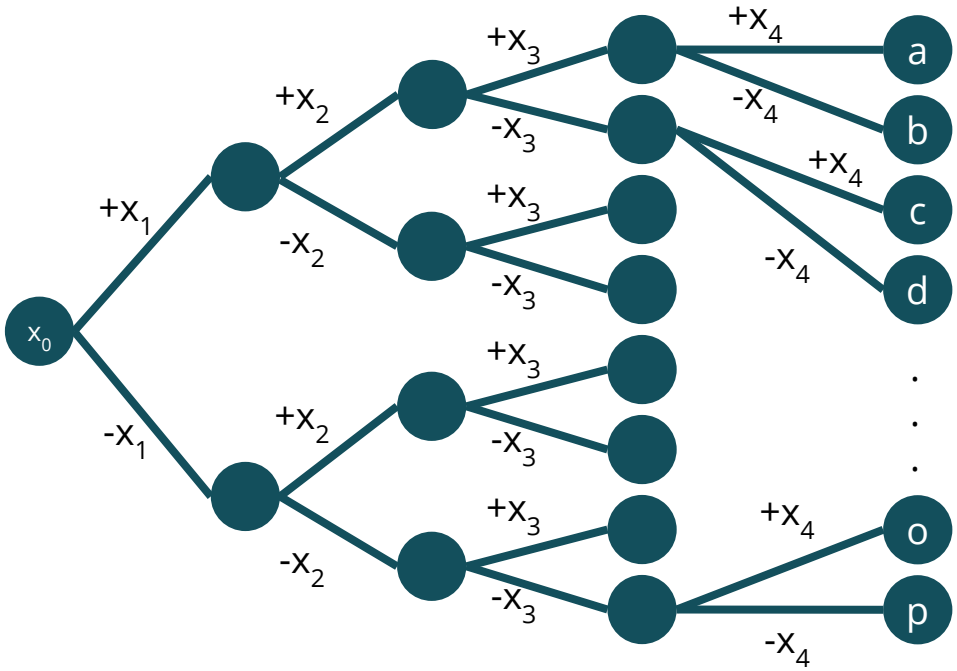


- Maintain 2 lists instead of 1
  - One list for half of x[], another list for another half
- After finding y, use meet-in-the-middle to deduce the list of signs

Subtask 4: $100000(1+1/K)^+$ queries

- $x_0 \pm x_1 \pm x_2 \pm \ldots \pm x_k = y$
- $k = 4$

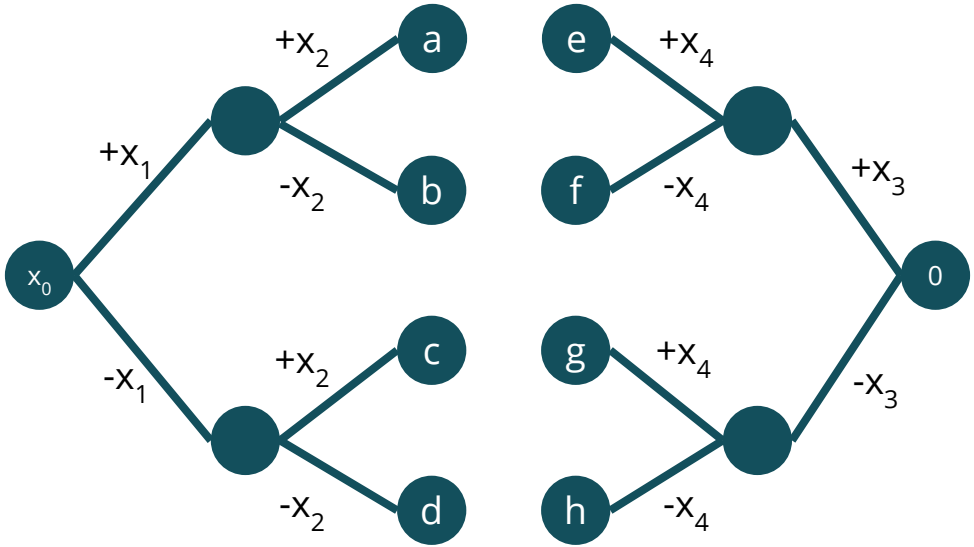
- Find y from {a, b, ..., p}



Subtask 4: $100000(1+1/K)^+$ queries

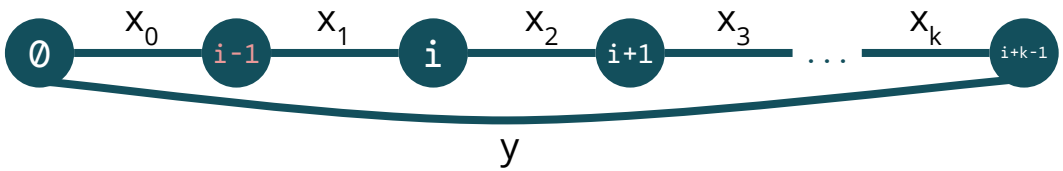
- $x_0 \pm x_1 \pm x_2 \pm \ldots \pm x_k = y$
- $k = 4$

- Find y - a from {e, f, g, h}
- Find y - b from {e, f, g, h}
- Find y - c from {e, f, g, h}
- Find y - d from {e, f, g, h}



Subtask 4: 100000(1+1/K)$^+$ queries

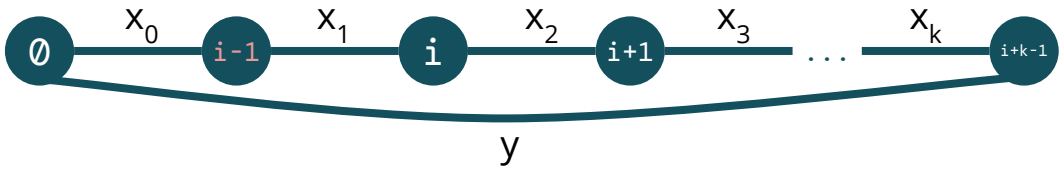
- Generalize the idea of "~~avoiding~~ parallelogram"



- If there is ambiguity, recursively solve for the first half, then the second half
  - Note that only the y in the first half requires an extra query
- There are some other strategies to tackle ambiguity cases while this might be the simplest one

Subtask 4: 100000(1+1/K)⁺ queries

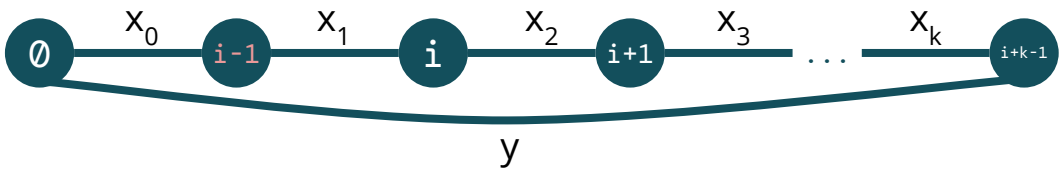
- Generalize the idea of "~~avoiding~~ parallelogram"



- Time complexity: $O(N/K * 2^{0.5K})$
  - the log factor can be avoided by
    - keep merging 2 sorted lists instead of using `std::set`
    - use combined linear search instead of binary search for meet-in-the-middle
- Number of queries >= 100000 * (1 + 1/K)

Subtask 4: 100000(1+1/K)$^+$ queries

- Generalize the idea of "~~avoiding~~ parallelogram"



- Our solution (K = 14): 108296 queries (Score: 100)
  - 100000(1 + 1/14) ≈ 107143
- Note that the number of queries might increase when K increases

M2114
Harmonious Music

Subtask 3

Subtask 3 (22%): A[i] are distinct

In the following discussion, we assume A[i] is sorted in ascending order. All time complexities stated excluded the time needed for sorting.

- It should be obvious that the final sequence should be {A[N] - N + 1, A[N] - N + 2, ...,, A[N] - 1, A[N]} / interval [A[N] - N + 1, A[N]]
- Simply sum up the difference of each element's original value and its final value = ans
- Note that for subtasks 3 - 5, long long is required for storing the answer

Score: 22

Time Complexity: O(N)

Subtask 1, 2

Subtask 1 (15%): N, A[i] ≤ 100, Subtask 2 (12%): N, A[i] ≤ 2000

- We know that the final sequence must be in one of [1, N], [2, N + 1], ..., [2000, N + 1999] (The values of N and A[i] are small so you can exhaust a bunch of them)
- Exhaust these possibilities, for each valid interval, calculate the cost needed similar to solution for subtask 3. Take the minimum cost as the answer.

Score: 27 (Cumulative: 49)

Time Complexity: O(A[N] * N)

Subtask 4

Subtask 4 (16%): $N \leq 2000$

What may the final sequence be?

- Suppose the final sequence is described by an array B. We know that, for B to be valid, $A[i] \leq B[i]$ for all $i = 1..N$.
- Since all elements of $A \leq A[N]$, B = interval $[A[N], A[N] + N - 1]$ must be valid. However, there may be intervals that are achievable with lower cost.

Possible final sequences:

- interval $[A[N] - N + 1, A[N]]$, $[A[N] - N + 2, A[N] + 1]$, ..., $[A[N], A[N] + N - 1]$

Subtask 4

Possible final sequences:

- interval [A[N] - N + 1, A[N]], [A[N] - N + 2, A[N] + 1], ..., [A[N], A[N] + N - 1]

Exhaust these N possibilities, for each valid interval, calculate the cost needed similar to solution for subtask 3. Take the minimum cost as the answer.

Score: 43 (Cumulative: 65)

Time Complexity: O(N^2)

Full solution

- interval [A[N] - N + 1, A[N]], [A[N] - N + 2, A[N] + 1], ..., [A[N], A[N] + N - 1]

Actually, for the solution in subtask 4, finding the first valid interval is sufficient to obtain the answer.

- Proof: if interval [a, b] is valid, then interval [a + 1, b + 1] must be valid, with an additional cost of N.

How can we use this to our advantage to make the program run faster?

Full solution

Let's write out the pseudo code for the solution first.

```
For i = 0 .. N - 1 // exhausting possible interval
  reset cost to 0
  For j = N .. 1 // checking
    target[j] = (A[N] - N + i) + j
    if (A[j] <= target[j]) cost += target[j] - a[j]
    else if (A[j] > target[j]) // invalid
      set flag to show invalid, break
```

Full solution

Suppose an invalidity appear at j, i.e. A[j] > target[j]

That implies for all k > j, A[k] ≤ target[k]. When we update the i loop, (to consider next possible interval), we just have to add (N - j) to cost, and can continue the check from j.

Full solution

```
i = 0, j = N, cost = 0
While (j >= 1)
  target[j] = (A[N] - N + i) + j
  if (A[j] <= target[j])
    cost += target[j] - a[j], --j
  else if (A[j] > target[j]) // invalid
    cost += N - j, ++i
```

Score: 100

Time Complexity: O(N)

Alternative Full solution

- Try to maintain a interval that satisfy the “harmonious” definition. Insert the element 1 by 1 from A[N] to A[0]
- Suppose the interval maintained is [L, R], for the next element inserted (A[i]), two cases have to be considered

  1. If A[i] < L, perform operation to make A[i] become L - 1. Insert and update the interval

  2. If A[i] = L, perform operation to make A[i] become R + 1. Insert and update the interval

  A[i] must be ≤ L since the array is sorted and each operation we do increases the value of the elements inserted before it

Alternative Full solution

- Insert the element 1 by 1 from A[N] to A[0]

  1. If A[i] < L, perform operation to make A[i] become L - 1. Insert and update the interval

  2. If A[i] = L, perform operation to make A[i] become R + 1. Insert and update the interval

- A std::set can be used to maintain the interval, but in this case we only care about the lower and upper bounds of the interval so it is not needed.

Score: 100

Time Complexity: O(N)