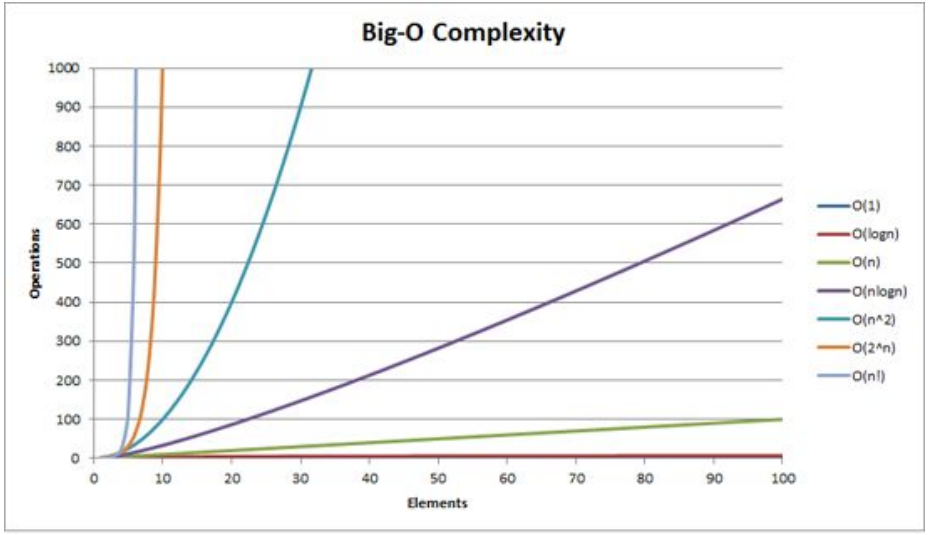
Optimization

Jeremy Chow 21-4-2018

Motivation

- As problem size increase, the run time required may not increase linearly
- For example, the bubble sort algorithm requires O(N^2) comparisons



Motivation

- If you received TLE verdicts, optimizations may help you
- Every experienced competitive programmers should know
- **Usually** we don't care about constant optimizations
- Our goal is to reduce the time complexity
  - e.g. from O(N^2) to O(N lg N) or O(N)
  - e.g. from O(QN) to O(Q lg N) or O(Q)

Optimization

- Avoid linear scans
- Avoid repeated computation
- Use memory to exchange time
- Scale down the numbers

Agenda

- Parital sum / difference array
- Precomputation
- Sliding window (Two Pointers)
- Other techniques

1D Partial sum - Problem

- Given an array of integers and Q queries, for each query, find out the sum of a contiguous section of the array

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

1+0+4+2=7

4+2+0+1+8=15

1D Partial sum - Naïve solution

- For each query
- loop over the required contiguous section of the array
- add up all numbers

```cpp
#include <bits/stdc++.h>

using namespace std;

int a[100005];

int main(){
	int n, q;
	scanf("%d%d", &n, &q);

	for(int i = 1; i <= n; i++)
	scanf("%d", &a[i]);

	for(int i = 0; i < q; i++){
		int l, r;
		scanf("%d%d", &l, &r);

		long long sum = 0;
		for(int j = l; j <= r; j++)
		sum += a[i];

		printf("%lld\n", sum);
	}
```

1D Partial sum - Naïve solution

- What happen when the input is like this
- For each query, you need to loop over the whole array
- Worse case time complexity : O(QN)

```
n m q
a_1 a_2 ... a_n
l n
l n
l n
.....
```

1D Partial sum - Naïve solution

- SLOW!!!!
- When N and Q are large (~10^5), you can't solve it within a second
- Need optimization
  - Avoid linear scans, precompute

1D Partial sum - Optimised solution

- We compute another array *y*
- The i^{th} element = sum of the numbers in [1..i]

idx	1	2	3	4	5	6	7	8
a[i]	2	1	0	4	2	0	1	8
y[i]	2	3	3	7	9	9	10	18

2+1+0+4+2=9

1D Partial sum - Optimised solution

- How do this array help us?
- sum in [l..r] = sum in [1..r] - sum in [1..l-1] !
- If we have this array *y*, we can compute sum in [l..r] by just visiting 2 entries

idx	1	2	3	4	5	6	7	8
a[i]	2	1	0	4	2	0	1	8
y[i]	2	3	3	7	9	9	10	18

9-2 = 7 = 1+0+4+2

1D Partial sum - Optimised solution

- Time complexity : O(Q+N)
- Much better than O(QN)
- Can pass even when N and Q is large
- Remainder : Remember to use long long when the range of elements is large (e.g. ai <= 10^9)

```cpp
#include <bits/stdc++.h>

using namespace std;

int a[100005];
long long y[100005];

int main(){
        int n, q;
        scanf("%d%d", &n, &q);

        for(int i = 1; i <= n; i++)
        scanf("%d", &a[i]);

        for(int i = 1; i <= n; i++)
        y[i] = y[i - 1] + a[i];

        for(int i = 0; i < q; i++){
                int l, r;
                scanf("%d%d", &l, &r);

                long long sum = y[r] - y[l - 1];

                printf("%lld\n", sum);
        }
```

precompute

2D Partial sum - Problem

- Given a 2D array of integers and Q queries, for each query, find out the sum of a rectangular region

i \ j	1	2	3	4	5
1	4	2	6	8	2
2	3	1	9	8	3
3	5	8	0	7	0
4	4	9	6	8	4
5	2	1	0	1	2

6+8+2+9+8+3+0+7+0=43

2D Partial sum - Naïve solution

- For each query
- loop over the required rectgular region
- add up all nubmers

```cpp
#include <bits/stdc++.h>

using namespace std;

int a[1005][1005];

int main(){
	int n, m, q;
	scanf("%d%d%d", &n, &m, &q);

	for(int i = 0; i < n; i++){
		for(int j = 0; j < m; j++)
		scanf("%d", a[i][j]);
	}

	for(int i = 0; i < q; i++){
		int x, y, xx, yy;

		scanf("%d%d%d%d", &x, &y, &xx, &yy);

		long long sum = 0;
		for(int j = x; j <= xx; j++){
			for(int k = y; k <= yy; k++)
			sum += a[j][k];
		}

		printf("%lld\n", sum);
	}
```

2D Partial sum - Naïve solution

- Again, when all Q queries ask for the whole array's sum
- Worst case time complexity = O(QNM)
- TLE when Q, N and M = 1000
- Use idea of 1D partial sum to optimise it

2D Partial sum - 1D optimised solution

- For each row, we apply 1D parital sum
- For each query, we loop over the required row
- add the required interval sum for each row

2D Partial sum - 1D optimised solution

i \ j	1	2	3	4	5
1	4	2	6	8	2
2	3	1	9	8	3
3	5	8	0	7	0
4	4	9	6	8	4
5	2	1	0	1	2

i \ j	1	2	3	4	5
1	4	6	12	20	22
2	3	4	13	21	24
3	5	13	13	20	20
4	4	13	19	27	31
5	2	3	3	4	6

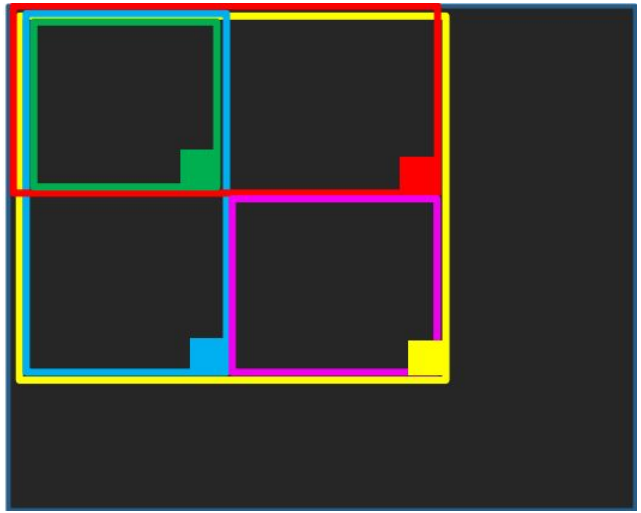
(22-6)+(24-4)+(20-13)=43

2D Partial sum - 1D optimised solution

- Time complexity : O(QN) or O(Qmin(N,M))
- Improved
- Can we do it better?

2D Partial sum - Solution

- In 1D version, sum in [l..r] = sum in [1..r] - sum in [1..l-1]
- Can we get some similar formula in 2D version?

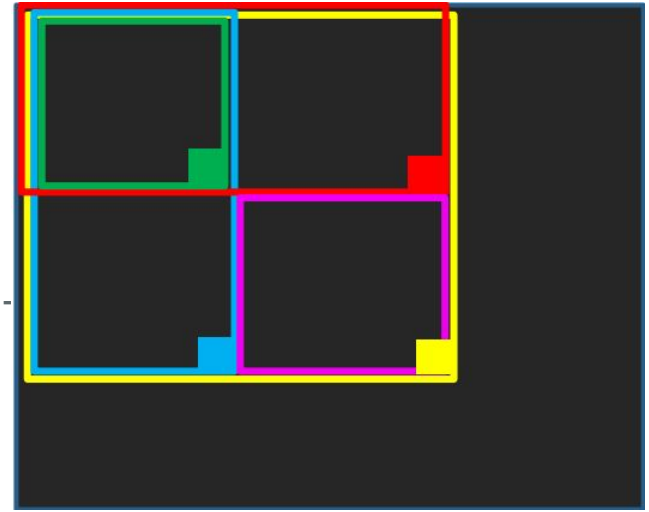


2D Partial sum – Solution

- Magenta = Yellow - Red - Blue + Green
- sum in [x1..x2][y1..y2] = sum in [1..x2][1..y2] -

  sum in [1..x1-1][1..y2] - sum in [1..x2][1..y1-1] -

  sum in [1..x1-1][1..y1-1]



2D Partial sum - Solution

- Compute another array *s*
- s[i][j] = sum in [1..i, 1..j]
  - = a[i][j] + s[i - 1][j] + s[i][j - 1] - s[i - 1][j - 1];
- Ans = s[x2][y2] - s[x2][y1-1] - s[x1-1][y2] + s[x1-1][y1-1]

2D Partial sum - Solution

- Time complexity = O(Q+NM)
- Partial sum always appear in OI
- Need to be able to code it

```
int n, m, q;
scanf("%d%d%d", &n, &m, &q);

for(int i = 1; i <= n; i++){
      for(int j = 1; j <= m; j++)
      scanf("%d", &a[i][j]);
}

for(int i = 1; i <= n; i++){
      for(int j = 1; j <= m; j++){
            s[i][j] = a[i][j] + s[i - 1][j] + s[i][j - 1] - s[i - 1][j - 1];
      }
}

for(int i = 0; i < q; i++){
      int x1, y1, x2, y2;

      scanf("%d%d%d%d", &x1, &y1, &x2, &y2);

      long long sum = s[x2][y2] - s[x2][y1 - 1] - s[x1 - 1][y2] + s[x1 - 1][y1 - 1];

      printf("%lld\n", sum);
}
```

1D Difference array - Problem

- There are Q queries, each query add vi to the contiguous section of the array, find the final value of the array
- ADD 3 to [2..5]

idx	1	2	3	4	5
a_i	0	3	3	3	3

- ADD 4 to [1..3]

idx	1	2	3	4	5
a_i	4	7	7	3	3

1D Difference array - Naïve solution

- For each query, loop that contiguous section
- add vi to them
- Time complexity O(QN)
- Can we do better?

```
#include <bits/stdc++.h>

using namespace std;

int a[100005]
long long s[1005][1005];

int main(){
	int n, m, q;
	scanf("%d%d", &n, &q);

	for(int i = 1; i <= n; i++)
	scanf("%d", &a[i]);

	for(int i = 0; i < q; i++){
		int l, r, v;
		scanf("%d%d%d", &l, &r, &v);
		for(int j = l; j <= r; j++)
		a[j] += v;
	}
```

1D Difference array - Solution

- Define a new array *d*
- d[i] = a[i] - a[i - 1]
- If we can find array *d*, we can get array *a* easily by a[i] = d[i] + a[i - 1]

idx	1	2	3	4	5
a_i	4	7	7	3	3

idx	1	2	3	4	5
d_i	4	3	0	-4	0

1D Difference array - Solution

- Imagine what happen when we add vi on the contiguous section [l..r]
- The difference between a[l] and a[l - 1] will + vi
- The difference between a[r+1] and a[r] will - vi
- We only need to update 2 values (d[l] and d[r+1]) instead of (l-r+1) values

1D Difference array - Solution

- Time complexity = O(N + Q)
- Way better than O(NQ)
- Frequently used technique

```cpp
#include <bits/stdc++.h>

using namespace std;

int a[100005]
long long d[100005];

int main(){
	int n, m, q;
	scanf("%d%d", &n, &q);

	for(int i = 1; i <= n; i++)
	scanf("%d", &a[i]);

	for(int i = 0; i < q; i++){
		int l, r, v;
		scanf("%d%d%d", &l, &r, &v);
		d[l] += v;
		d[r + 1] -= v;
	}

	for(int i = 1; i <= n; i++)
	a[i] = a[i - 1] + d[i];
```

2D Difference array - Problem

- There are Q queries, each query add vi to the rectangular region of the matrix, find the final value of the matrix
- Same as partial sum, difference array can be applied to 2D too
- Naïve solution works in O(QNM)
- ADD 5 to [1..3, 1..3]

0	0	0	0	0
0	5	5	5	0
0	5	5	5	0
0	5	5	5	0
0	0	0	0	0

2D Difference array - Solution

- Define a new array d
- d[i][j] = a[i][j] - a[i][j - 1] - a[i - 1][j] + a[i - 1][j - 1]
- Imagine what happen when we add vi on the rectangular region [l1..l2, r1..r2]
- d[l1][r1] += v, d[l1][r2 + 1] -= v, d[l2 + 1][r1] -= v, d[l2 + 1][r2 + 1] += v;
- We only need to update 4 value per query

2D Difference array - Solution

- After getting array d, we can get array a easily by
  - a[i][j] = a[i - 1][j] + a[i][j - 1] - a[i - 1][j - 1] + d[i][j]

5	0	0	-5	0
0	0	0	0	0
0	0	0	0	0
-5	0	0	5	0
0	0	0	0	0

0	0	0	0	0
0	5	5	5	0
0	5	5	5	0
0	5	5	5	0
0	0	0	0	0

2D Difference array - Solution

- Time complexity = O(NM + Q)

```cpp
#include <bits/stdc++.h>

using namespace std;

int a[1005][1005];
long long d[1005][1005];

int main(){
      int n, m, q;
      scanf("%d%d%d", &n, &m, &q);

      for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++)
            scanf("%d", &a[i][j]);
      }

      for(int i = 0; i < q; i++){
            int l1, r1, l2, r2, v;
            scanf("%d%d%d%d%d", &l1, &r1, &l2, &r2, &v);

            d[l1][r1] += v, d[l1][r2 + 1] -= v, d[l2 + 1][r1] -= v, d[l2 + 1][r2 + 1] += v;
      }

      for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++)
            a[i][j] = a[i - 1][j] + a[i][j - 1] - a[i - 1][j - 1] + d[i][j];
      }

      for(int i = 1; i <= n; i++){
            for(int j = 1; j <= m; j++)
            printf(" %d", a[i][j]);
            printf("\n");
      }
```

Precomputation – Problem

- Given a string consists of ‘A’ and 'B' and Q queries, for each query qi, you need to find the closest “B” which index is <= qi
- AABAAABBA
- Q 3 -> 3
- Q 9 -> 8

Precomputation - Naïve solution

- For each query, loop over all the index <= qi
- Time complexity = O(QN)
- With the help of precomputation, we can improve it !

Precomputation - Solution

- Build a array *lt*, which lt[i] means the last “B” which index <= i
- AABAAABBA

idx	1	2	3	4	5	6	7	8	9
lt_i	-1	-1	3	3	3	3	7	8	8

Precomputation - Solution

- You can build that array easily in O(N)
- For each query, you just need to print the precomputed lt[qi]
- Time complexity = O(Q+N)

Precomputation - Solution

- useful array to be precomputed
  - Prefix / suffix sum (partial sum)
  - Prefix / suffix max / min
  - Prefix / suffix xor sum
  - Prefix / suffix count
    - number of odd numbers
    - number of “*”
    - index of last special element

Two pointers - Problem

- Given two sorted array of integers *a* and *b*, find the number of pair (i, j) such that

  ai + bj = C

1	5	8	10	12	14	15	18	25
3	6	6	7	13	14	15	18	19

- when C = 20
- ANS = 3 {(1, 9), (2, 7), (6, 3)}

Two pointers - Naïve solution

- For each element in a, loop over array b
- count how many ai + bj = C
- Time complexity = O(N^2)
- Hint : **Sorted** array

Two pointers - Binary search solution

- For each element in a, binary search the count of numbers

  such that bj = C - ai

- Need two binary search if the numbers are not distinct
- However, we can improve it more

Two pointers - Solution

- Just like binary search, two pointers can improve the algorithm by avoiding impossible case
- Also, it avoid repeated checking.

Two pointers - Solution

- Notice array a and b is **sorted**, let's assume we are loop the array a
- For each ai, our target is the elements in b equal to C - ai
- When i grow, ai is increasing, so our target C - ai is decreasing
- For the number larger than C - ai, we don’t need to consider it in i + 1, i + 2, …, n
- Avoid impossible case

Two pointers - Solution

1	5	8	10	12	14	14	18	25
3	5	6	7	13	14	15	18	19

1	5	8	10	12	14	14	18	25
3	5	6	7	13	14	15	18	19

1	5	8	10	12	14	14	18	25
3	6	6	7	13	14	15	18	19

1	5	8	10	12	14	14	18	25
3	6	6	7	13	14	15	18	19

1	5	8	10	12	14	14	18	25
3	6	6	7	13	14	15	18	19

Two pointers - Solution

- As both pointers traverse the array once
- Time complexity = O(N)
- We usually use while loop to implement
- Easy to code

Assist pointer

Main pointer

```
1   int j = n - 1;
2   int res = 0;
3
4   for(int i = 0; i < n; i++){
5       while(j >= 0 && b[j] > c - a[i])
6       j--;
7
8       if(j >= 0 && a[i] + b[j] == c)
9       res++;
10  }
11
```

Two pointers - When to use

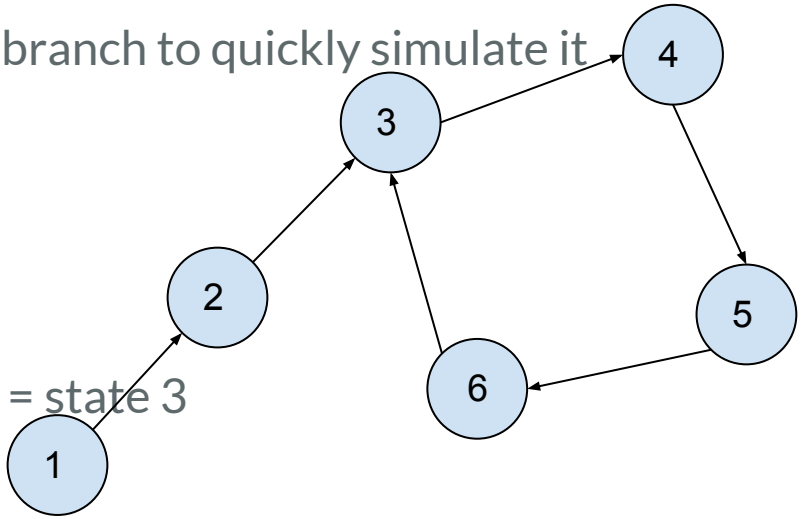
- On sorted array
- Things we want to find have monotonicity
- e.g. sum, count of sth etc.

Other techniques - finding cycles

- In some simulation problems, we may need to simulate N steps
- However, N is really large (e.g. ~10^18)
- TLE if you do O(N) simulate

Other techniques - finding cycles

- Usually in this type of problems, some state will form a cycle
- You need to find out the cycle and the branch to quickly simulate it
- branch = 2, cycle = 4
- E.g. walk 98 steps from 1
- ANS = ((98 - 2) % 4)th state in the cycle = state 3
- Time complexity = O(no. of state)



Other techniques - discretization

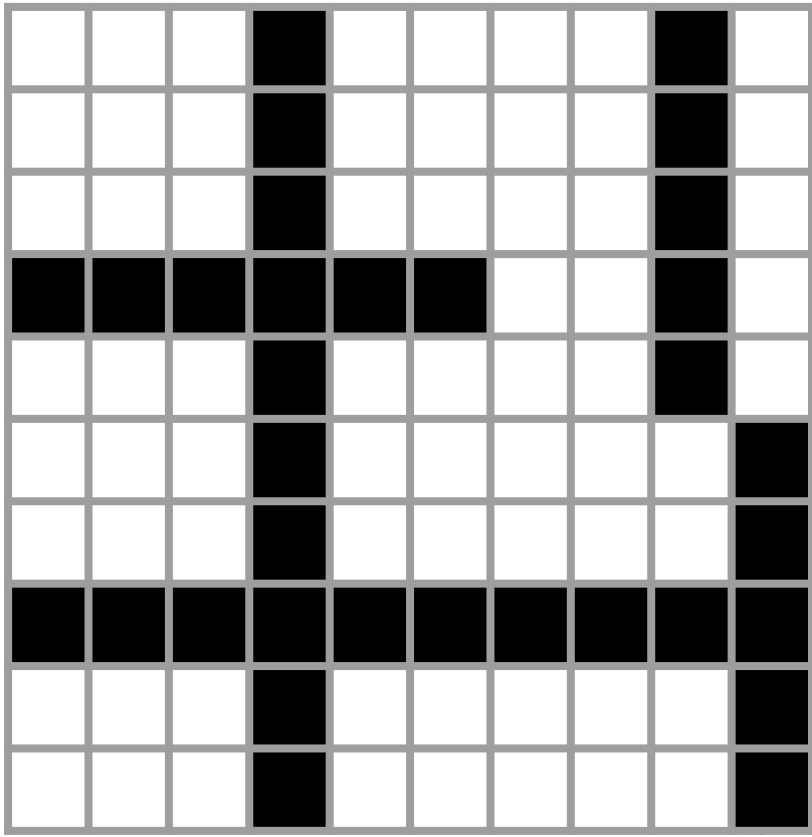
- Discretization (離散法) is a technique that converts values (not necessarily integers) into integers, while maintaining their relative order
- Example: 7654321, 123456, 934602, 123456789
  -> 3, 1, 2, 4 (or 2, 0, 1, 3)
- Put the values into an array, sort the array
  - 123456, 934602, 7654321, 123456789
- For each value in the original array, find its rank using binary search

Other techniques - discretization

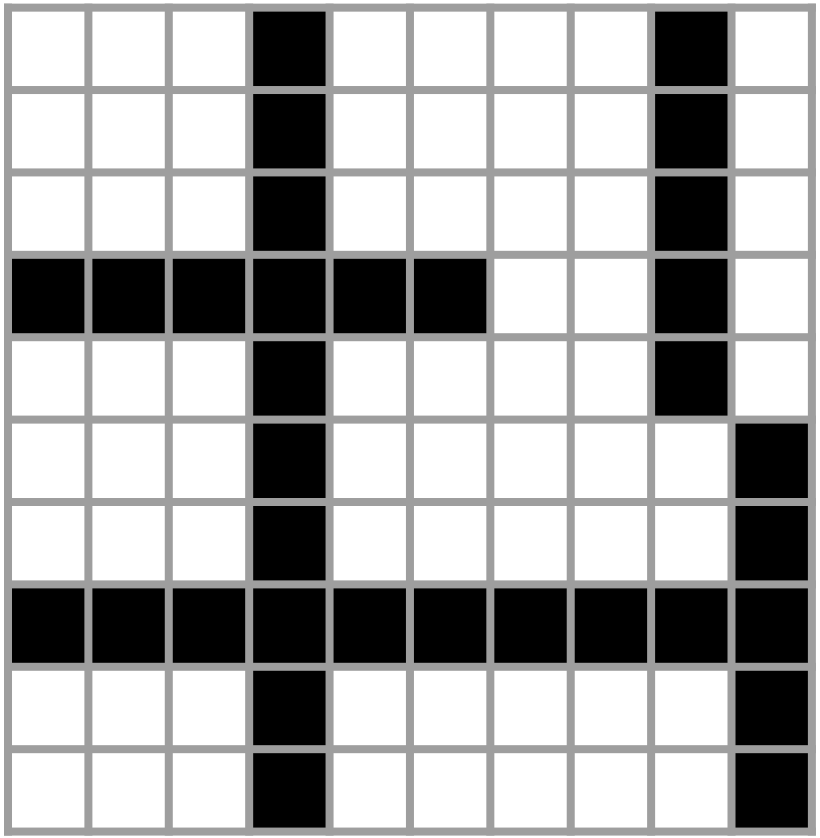
- Discretize large numbers into smaller numbers
- Handle data easily
- E.g count the number of occurrence of some numbers in array a
- ai <= 10^9
- Counting with array after discretization

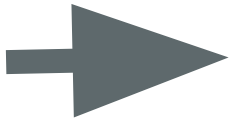
Other techniques - discretization

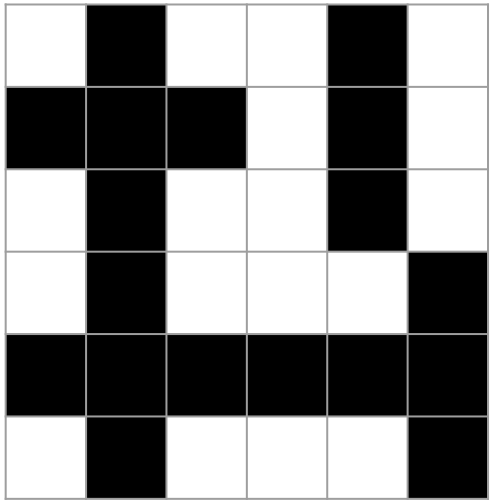
- 座標壓縮
- useful when the coordinates are large
- can perform dfs / bfs on the compressed grid
  - e.g. find the number of connected component



Other techniques - discretization







Q&A