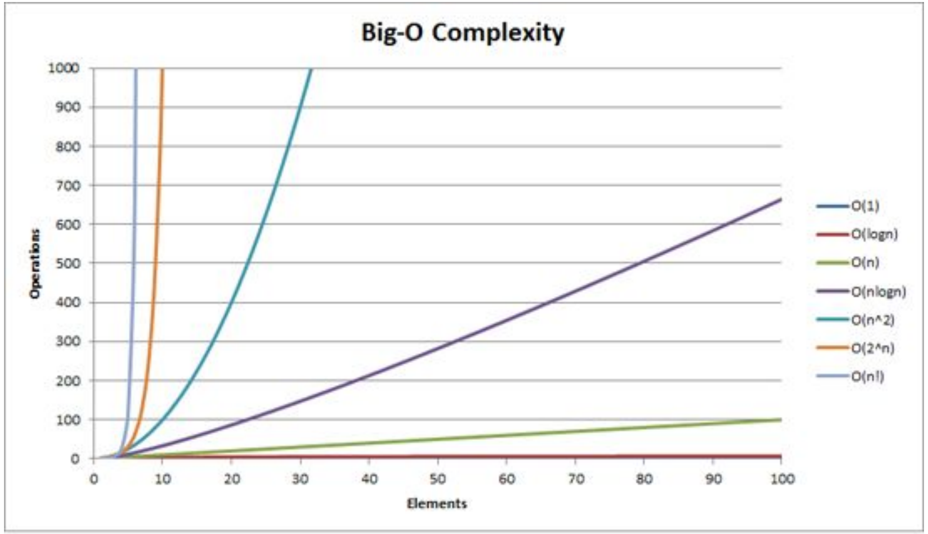
Optimization and Common Tricks

Jeremy Chow 16-2-2019

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 and Common Tricks

- 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)
- ❖ Finding cycle
- ❖ discretization (1D / 2D)

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

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

Query(2, 5) = 1+0+4+2 = 7
Query(4, 8) = 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

```
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[j];

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

- Good! But..

1D Partial sum - Naïve solution

- What happen when the input is like this

$$
\begin{array}{l}
N\ Q \\
\left.\begin{array}{l} 1\ n \\ 1\ n \\ 1\ n \\ 1\ n \\ \quad\ldots \end{array}\right\} Q
\end{array}
$$

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)

```
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[j];

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

Loop N times per query

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 *ps*
  - *Stands for Partial Sum*
- The i^{th} element = sum of the numbers in [1..i]
- Use this array to help us calculate the answer faster
- Avoid repeated computation over different queries

1D Partial sum - Optimised solution

- How to compute it ?
- By definition - ps[i] = sum(1, i) = a[1] + a[2] + a[3] + … + a[i]
- ps[i] = ps[i - 1] + a[i]

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

- Be careful if you are using 0-based

1D Partial sum - Optimised solution

- How to compute it ?
- By definition - ps[i] = sum(1, i) = a[1] + a[2] + a[3] + … + a[i]
- ps[i] = ps[i - 1] + a[i]

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

ps[5] = ps[4] + a[4] = 7 + 2 = 9

1D Partial sum - Optimised solution

- How to compute it ?
- By definition - ps[i] = sum(1, i) = a[1] + a[2] + a[3] + … + a[i]
- ps[i] = ps[i - 1] + a[i]

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

ps[5] = 2 + 1 + 0 + 4 + 2 = 9

1D Partial sum - Optimised solution

- How does this array help us?
- sum in [l..r] = sum in [1..r] - sum in [1..l-1] !
- For example
  - sum in [2..5] = sum in [1..5] - sum in [1..1]
  - = (a[1] + a[2] + a[3] + a[4] + a[5]) - (a[1])
  - = a[2] + a[3] + a[4] + a[5]

1D Partial sum - Optimised solution

- How does this array help us?
- sum in [l..r] = sum in [1..r] - sum in [1..l-1] !
- We can compute sum in [l..r] by just visiting 2 entries of ps!

idx	1	2	3	4	5	6	7	8
a[i]	2	1	0	4	2	0	1	8
ps[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
  - Use long long when the range of elements is large (e.g. ai <= 10^9)
  - Be careful if you use 0-based (l = 0)

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

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

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

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

```
for (int i = 0; i < q; i++) {
	int x1, y1, x2, y2;
	scanf("%d%d%d%d", &x1, &y1, &x2, &y2);

	long long sum = 0;
	for (int j = x1; j <= x2; j++) {
		for (int k = y1; k <= y2; 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

sum[i][l..r] = ps[i][r] - ps[i][l - 1]

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

Query((1,3), (3,5)) = (22-6)+(24-4)+(20-13) = 43

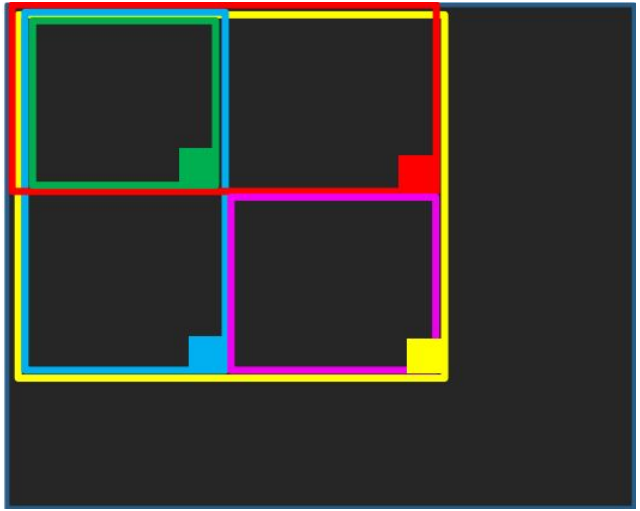
2D Partial sum - 1D optimised solution

- Time complexity : $O(QN)$ or $O(Q * \min(N,M))$
- Improved
- But not good enough
- Can we do 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((2,2), (3,3)) = sum((1,1), (3,3)) -
- sum((1, 1), (3, 1)) - sum((1, 1), (1,3)) +
- sum((1, 1), (1,1))

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

2D Partial sum - Solution

- Compute another array *ps*
- ps[i][j] = sum in [1..i][1..j]
  - = a[i][j] + ps[i - 1][j] + ps[i][j - 1] - ps[i - 1][j - 1];

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

i \ j	1	2	3
1	4	6	12
2	3	4	13
3	5	13	13

i \ j	1	2	3
1	4	10	22
2	7	17	42
3	12	35	73

2D Partial sum - Solution

- Ans = ps[x2][y2] - ps[x2][y1-1] - ps[x1-1][y2] + ps[x1-1][y1-1]
- Time complexity = O(Q+NM)
- Partial sum always appear in OI
- Should be able to code it

```
for (int i = 0; i < q; i++) {
    int x1, y1, x2, y2;
    scanf("%d%d%d%d", &x1, &y1, &x2, &y2);

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

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

1D Difference array - Problem

- There are Q queries, each query add value 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?

```
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

```
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];
```

1D Difference array - Variation

- Instead of just adding a constant
- We can actually add an arithmetic sequence to the subarray
- E.g. add an arithmetic sequence to a[2..5], where initial value = 4, difference = 5

idx	1	2	3	4	5
a_i	0	4	9	14	19

1D Difference array - Variation

- Let the initial value be A, difference be D
- an arithmetic sequence = a constant (A - D) + (D, 2D, 3D ....)
- E.g. (4, 9, 14, 19), A = 4, D = 5
- = [-1, -1, -1, -1, ...] + [5, 10, 15, 20, ...]

idx	1	2	3	4	5
a_i	0	4	9	14	19

1D Difference array - Variation

- E.g. (4, 9, 14, 19), A = 4, D = 5
- = [-1, -1, -1, -1, ….] + [5, 10, 15, 20, …..]
- The first part is just our original 1D difference array problem
- But how to do the second part?

idx	1	2	3	4	5
a_i	0	4	9	14	19

1D Difference array - Variation

- We can still apply the difference array technique
- a[i] = a[i - 1] + D
- Can be easily done by for loop
- However, there are multiple query, and the query does not always start from 1 and end in N

1D Difference array - Variation

- If i is involved in more than 1 arthmetic squence
- a[i] = a[i - 1] + sum[i], where sum[i] = sum of D which i is involved
- E.g add (3, 6, 9, …) to a[1..4] and add (4, 8, 12, …) to B[2..5]
- sum[3] = 3 + 4 = 7, a[3] = a[2] + sum[2] = 10 + 7 = 17

idx	1	2	3	4	5
a[i]	3	10	17	24	16

1D Difference array - Variation

- sum[i] can also be calculate by difference array
- So we can solve the problem now :)
- Remember to cancel the effect of query on (l..r) after r
- $sum[r + 1] -= (r - l + 1) * D$
- $sum[r + 2] += (r - l + 1) * D$

1D Difference array - Variation

- E.g. add {2, 5, 8, 11} (A = 2, D = 3) to B[1..4] -> C[1] += -1, C[5] -= -1
- add {2, 6, 10, 14} (A = 2, D = 4) to B[2..5] -> C[2] += -2, C[6] -= -2

idx	1	2	3	4	5
C[i]	-1	-2	0	0	1

idx	1	2	3	4	5
C[i]	-1	-3	-3	-3	-2

1D Difference array - Variation

- E.g. add {2, 5, 8, 11} (A = 2, D = 3) to B[1..4]
  - SUM[1] += 3, SUM[5] -= 3 + 12, SUM[6] += 12
- add {2, 6, 10, 14} (A = 2, D = 4) to B[2..5]
  - SUM[2] += 4, SUM[6] -= 4 + 16, SUM[7] += 16

idx	1	2	3	4	5
SUM[i]	3	4	0	0	-15

1D Difference array - Variation

idx	1	2	3	4	5
SUM[i]	**3**	**4**	**0**	**0**	**-15**

idx	1	2	3	4	5
SUM[i]	**3**	**7**	**7**	**7**	**-8**

idx	1	2	3	4	5
SUM[i]	**3**	**10**	**17**	**24**	**16**

1D Difference array - Variation

- A[i] = SUM[i] + C[i]
- Time Complexity = O(Q + N)
- Cool

```
for (int i = 0; i < q; i++) {
        int l, r, a, d;
        scanf("%d%d%d%d", &l, &r, &a, &d);
        c[l] += a - d;
        c[r + 1] -= a - d;
        sum[l] += d;
        sum[r + 1] -= d + (r - l + 1) * d;
        sum[r + 2] += (r - l + 1) * d;
}

for (int i = 1; i <= n; i++) sum[i] += sum[i - 1];
for (int i = 1; i <= n; i++) {
        sum[i] += sum[i - 1];
        c[i] += c[i - 1];
        a[i] = sum[i] + c[i];
}

for (int i = 1; i <= n; i++) printf("%d\n", a[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 [x1..x2, y1..y2]
- d[x1][y1] += v, d[x1][y2 + 1] -= v, d[x2 + 1][y1] -= v, d[x2 + 1][y2 + 1] += v;
- We only need to update 4 value per query

2D Difference array - Solution

- d[x1][y1] += v, d[x1][y2 + 1] -= v, d[x2 + 1][y1] -= v, d[x2 + 1][y2 + 1] += v;
- We only need to update 4 value per query

```
for (int i = 0; i < q; i++) {
        int x1, y1, x2, y2, v;
        scanf("%d%d%d%d%d", &x1, &y1, &x2, &y2, &v);

        d[x1][y1] += v;
        d[x1][y2 + 1] -= v;
        d[x2 + 1][y1] -= v;
        d[x2 + 1, y2 + 1] += v;
}
```

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]

0	0	0	0	0
0	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	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)

```
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];
	}
}
```

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)

```
for (int i = 0; i < n; i++) {
    if (s[i] == 'B') lt[i] = i;
    else if (i > 0) lt[i] = lt[i - 1];
    else lt[i] = -1;
}

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

    if (lt[x] == -1) printf("no B before x\n");
    else printf("%d\n", lt[x]);
}
```

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

- when C = 20

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

- 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 $b_j = C - a_i$

- 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

TARGET = 20

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

```
int j = n - 1;
int res = 0;

for(int i = 0; i < n; i++){
    while(j >= 0 && b[j] > c - a[i])
    j--;

    if(j >= 0 && a[i] + b[j] == c)
    res++;
}
```

Two pointers - When to use

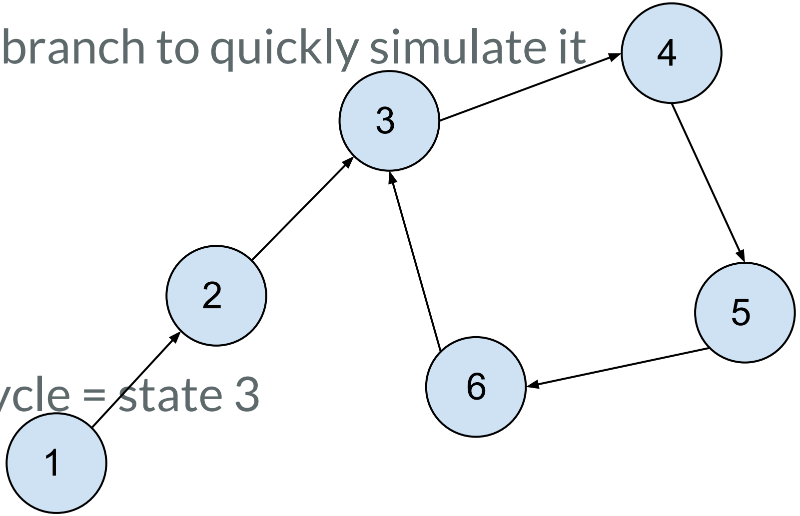
- On sorted array
- Things we want to find have monotonicity
- e.g. sum, count of sth etc.

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) + 3th 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

```
vector <int> v;

for (int i = 0; i < n; i++) v.push_back(a[i]);

sort(v.begin(), v.end());
v.resize(unique(v.begin(), v.end()) - v.begin());

for (int i = 0; i < n; i++) a[i] = lower_bound(v.begin(), v.end(), a[i]) - v.begin();
```

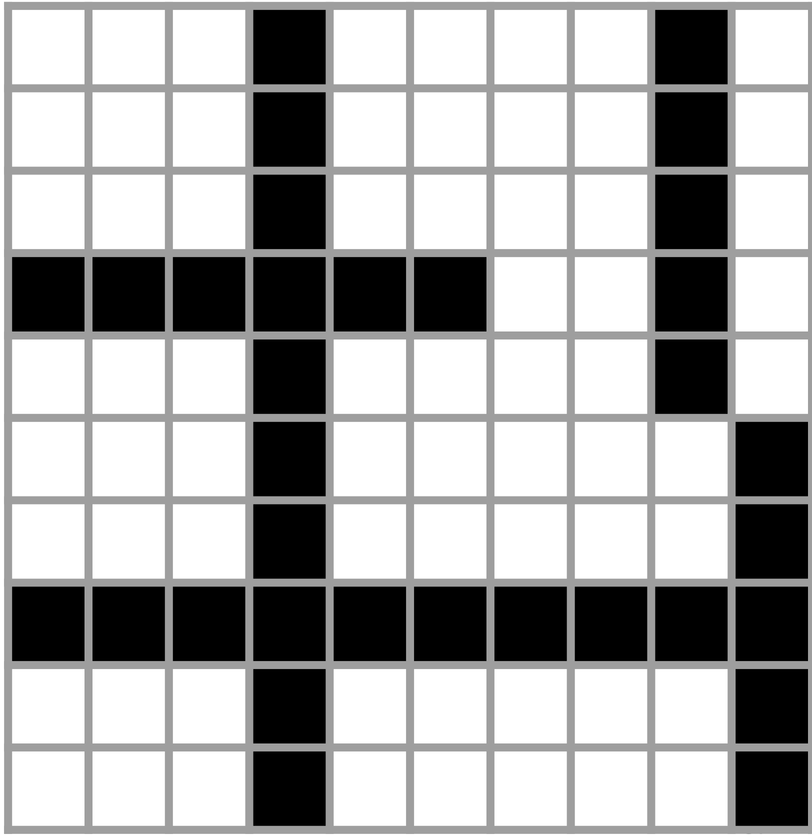
Other techniques - discretization

❖ 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