M1811 Almost Constant

2018-3-3

Problem Statement

- Given a sequence $a_1, \ldots, a_n$
- Find number of almost constant continuous subsequences.

- A sequence is almost constant if different of any two number is less than k

- Constraints:

  $1 <= N <= 500000$, $0 <= |a_i|$, $K <= 2000000000$

Observation

- Since for any element x > y, we have x – y <= max – min

- A sequence is almost constant iff max element – min element <= k

Subtask 1

- a_i is monotone
- Then for any sequence a_i, … a_j
  - We have $\begin{cases} max = a_i \\ min = a_j \end{cases}$ or $\begin{cases} max = a_j \\ min = a_i \end{cases}$
  - So $range_{i,j}$ = max – min = $|a_i - a_j|$
  - Since a_i is monotone, for fixed i, $diff_{i,j}$ increases as j increases
- So for every fixed i, we can binary search for the largest j such that $range_{i,j}$ <= k, denote the largest j as f_i
- Then ans = $\sum_{i=1}^{n}(f_i - i + 1)$

Subtask 1

- Time complexity: O(n lg n)
- An other solution is to use two-point and the fact that max – min = $|a_i - a_j|$
- Two-pointer solutions will be discussed later

Subtask 2

- K <= 1
- This subtask is designed for some kind of brute force algorithm, so there is no specific algorithms for this subtask.

Subtask 3

- 1 <= N <= 500

- We can find the max and min element for every continuous sequence

Subtask 3

```
for (int i = 1; i <= n; i++)
    for (int j = i; j <= n; j++) {
        int max = a[i], min = a[i];
        for (int k = i; k <= j; k++) {
            if (a[j] > max) max = a[j];
            if (a[j] < min) min = a[j];
        }
        if ((long long)max – min <= k) ans++;
    }
```

Subtask 3

- Time complexity: $O(n^3)$
- Remember the long long, since 2e9 – (-2e9) = 4e9 > 2e9

Subtask 4

- 1 <= N <= 5000

- Actually, we don’t need to calculate the max and min value for every continuous subsequence separately.

- We can reuse the value for longer subsequences which start at the same position.

Subtask 4

```
for (int i = 1; i <= n; i++) {
    int max = a[i], min = a[i];
    for (int j = i; j <= n; j++) {
        if (a[j] > max) max = a[j];
        if (a[j] < min) min = a[j];
        if ((long long) max – min <= k) ans++;
    }
}
```

Subtask 4

- Time complexity: $O(n^2)$
- Remember the long long, since 2e9 – (-2e9) = 4e9 > 2e9

Subtask 5

- No additional constraints

- There are many solutions to this question
- And many of them used the fact that
  - Every subsequence of an almost constant sequence is also almost constant.

- The proof is left as an exercise.

Subtask 5

- Here comes a two-pointer (aka sliding windows) algorithm
- First assume that we have a fast enough data structure which can tell us the minimum and maximum value over a continuous subsequence.
- Then for any fixed left point i, we can check that whether moving the right point j still gives us max – min <= k
  - If so, increase j by 1
  - Otherwise ans += j – i + 1 and then increase i by 1
- Then the time complexity will be O(n*f(n)) where f(n) is the time complexity for the query of max and min (for subtask 1, we don’t even need any data structure)

Subtask 5

- Now, let’s go back and see what data structure can be used to achieve this.
- Here we need the data structure to have update and query speed as slow as O(lg n).
- Actually, many data structure can do this, eg. RMQ, Segment tree
  - But these are not taught yet!
- Then you can use heap (aka priority queue), good news for C++ users
- There is one more container can be used in C++: std::multiset
- So learn more about C++ helps a lot.

Subtask 5

- Time complexity: $O(n \lg n)$

- This is fast enough to get full marks, can it be faster?
- I don't know many data structure, is there easier solution?

- The answer is yes.

Subtask 5++

- Actually, we can use two monotone queues to find a max and min element within the range, one increasing, one decreasing

- For C++ users, you can use std::deque to implement it, or you can simply use arrays.

Subtask 5++

- To push an element a[i] into the increasing deque, we need to check if the element at the back a[b] satisfy a[b] < a[i], if no pop until the deque is empty or the inequality is satisfied.
- Then a maximum element will always appear at the front of the decreasing deque
- A minimum element will always appear at the front of the increasing deque
- Remember to pop_front when the front element is out of range when using the two-pointer algorithm

Rainbow Necklace

solution

Lau Chi Yung

March 3, 2018

Subtask 1

- $c_i \leq 2$
- In other words, c_i is either 1 or 2
- Answer is also either 1 or 2
- If all c_i are equal, answer is 1
- otherwise, if M is 1, answer is 1
- otherwise, answer is 2

Subtask 2

- $M = N$
- Make a necklace with as many colors as possible with at most M beads
- Without worsening our answer, we can make that necklace with exactly M beads
- i.e. use the whole long string of colorful beads
- Just need to count the number of distinct colors. Possible methods:

counting array	$O(100000)$
sorting and scanning	$O(N \log N)$
binary search tree	$O(N \log N)$
`std::set`	$O(N \log N)$

Subtask 3

- $M, N \leq 1000$
- Till now, we should have realized the question is actually asking us to
  - Find the maximum number of distinct integers in a length-M contiguous subsequence in a length-N array of integers.
- There are $N - M + 1$ different contiguous subsequences to consider
- For each of these subsequences, calculate the number of distinct integers:

counting array	$O(100000)$
sorting and scanning	$O(N \log N)$
binary search tree	$O(N \log N)$
`std::set`	$O(N \log N)$

- Overall time complexity: $O(N^2 \log N)$ (for `std::set`)
- Be careful when $M > N$

Subtask 4

- $1 \leq N, M, c_i \leq 100000$
- Those $N - M + 1$ subsequences overlap each other
- We can derive the number of distinct integers of one subsequence from another subsequence

Subtask 4

$N = 10, M = 5$

store the first length-M subsequence in data structure

5	1	2	2	2	3	4	5	6	6
0	1	2	3	4	5	6	7	8	9

remove the subsequence's first item from data structure

5	1	2	2	2	3	4	5	6	6
0	1	2	3	4	5	6	7	8	9

add the item after the subsequence to data structure

5	1	2	2	2	3	4	5	6	6
0	1	2	3	4	5	6	7	8	9

- We derived the 2^{nd} length-M subsequence from the 1^{st} length-M subsequence
- Repeating the process, we can derive all subsequences
- The data structure should support:
  - Insert an integer
  - Remove an integer
  - Query the number of distinct integers

Subtask 4

$N = 10, M = 5$

store the first length-M subsequence in data structure

5	1	2	2	2	3	4	5	6	6
0	1	2	3	4	5	6	7	8	9

remove the subsequence's first item from data structure

5	1	2	2	2	3	4	5	6	6
0	1	2	3	4	5	6	7	8	9

add the item after the subsequence to data structure

5	1	2	2	2	3	4	5	6	6
0	1	2	3	4	5	6	7	8	9

- `std::map<int, int> m;`

insert i	`m[i]++;`	$O(\log N)$
remove i	`if (!--m[i]) m.erase(i);`	$O(\log N)$
query	`m.size()`	$O(1)$

- Overall time complexity: $O(N \log N)$
- Be careful when $M > N$
- This technique is called "sliding window"

Subtask 4

$N = 10, M = 5$

store the first length-M subsequence in data structure

5	1	2	2	2	3	4	5	6	6
0	1	2	3	4	5	6	7	8	9

remove the subsequence's first item from data structure

5	1	2	2	2	3	4	5	6	6
0	1	2	3	4	5	6	7	8	9

add the item after the subsequence to data structure

5	1	2	2	2	3	4	5	6	6
0	1	2	3	4	5	6	7	8	9

- `int m[100001] = {0}, colors = 0;`

insert i	`if (!m[i]) colors++; m[i]++;`	$O(1)$
remove i	`m[i]--; if (!m[i]) colors--;`	$O(1)$
query	`colors`	$O(1)$

- Overall time complexity: $O(N)$
- Be careful when $M > N$
- This technique is called "sliding window"

M1813 - Counting Good Strings

Alex Tung
alex20030190@yahoo.com.hk

3 March 2018

Subtasks 1 + 2

- Subtask 1: Check whether the given string is good

Subtasks 1 + 2

- Subtask 1: Check whether the given string is good
- The algorithm is simple:

Condition 1 (no K consecutive same bits)

```
LASTBIT := 0, CONSEC := 0
for i = 0...N − 1
   if s[i] = LASTBIT + '0'
      CONSEC := CONSEC + 1
   else
      LASTBIT := s[i] - '0', CONSEC := 1
   if CONSEC ≥ K
      return BAD
```

Subtasks 1 + 2

- To check condition 2, we use greedy matching.

Condition 2 (T is not a subsequence)

```
T_POS := 0
for i = 0...N − 1
   if T_POS < M and s[i] = t[T_POS]
      T_POS := T_POS + 1
if T_POS = M
   return BAD
```

Subtasks 1 + 2

- To check condition 2, we use greedy matching.

Condition 2 (T is not a subsequence)

```
T_POS := 0
for i = 0...N − 1
   if T_POS < M and s[i] = t[T_POS]
      T_POS := T_POS + 1
if T_POS = M
   return BAD
```

- For subtask 2, exhaust all possible strings and check one by one.

Full solution: DP

- Observe that we only need $LASTBIT$, $CONSEC$, and T_POS for the checking. This allows for a DP formulation with four parameters ($S\_POS, LASTBIT, CONSEC, T\_POS$).

Full solution: DP

- Observe that we only need $LASTBIT$, $CONSEC$, and T_POS for the checking. This allows for a DP formulation with four parameters $(S\_POS, LASTBIT, CONSEC, T\_POS)$.
- Transition is done by appending '0' or '1' to the current string, whichever is allowed.

Full solution: DP

- Observe that we only need $LASTBIT$, $CONSEC$, and T_POS for the checking. This allows for a DP formulation with four parameters $(S\_POS, LASTBIT, CONSEC, T\_POS)$.
- Transition is done by appending '0' or '1' to the current string, whichever is allowed.
- Append NEW_BIT. New state: $(S\_POS + 1, NEW\_BIT, x, y)$

Full solution: DP

- Observe that we only need $LASTBIT$, $CONSEC$, and T_POS for the checking. This allows for a DP formulation with four parameters $(S\_POS, LASTBIT, CONSEC, T\_POS)$.
- Transition is done by appending '0' or '1' to the current string, whichever is allowed.
- Append NEW_BIT. New state: $(S\_POS + 1, NEW\_BIT, x, y)$
- If $NEW\_BIT = BIT$, then $x = CONSEC + 1$.
  Otherwise, $x = 1$.

Full solution: DP

- Observe that we only need $LASTBIT$, $CONSEC$, and T_POS for the checking. This allows for a DP formulation with four parameters $(S\_POS, LASTBIT, CONSEC, T\_POS)$.
- Transition is done by appending '0' or '1' to the current string, whichever is allowed.
- Append NEW_BIT. New state: $(S\_POS + 1, NEW\_BIT, x, y)$
- If $NEW\_BIT = BIT$, then $x = CONSEC + 1$.
  Otherwise, $x = 1$.
- If NEW_BIT + '0' $= t[T\_POS]$, then $y = T\_POS + 1$.
  Otherwise, $y = T\_POS$.

Full solution: DP

- Observe that we only need $LASTBIT$, $CONSEC$, and T_POS for the checking. This allows for a DP formulation with four parameters $(S\_POS, LASTBIT, CONSEC, T\_POS)$.
- Transition is done by appending '0' or '1' to the current string, whichever is allowed.
- Append NEW_BIT. New state: $(S\_POS + 1, NEW\_BIT, x, y)$
- If $NEW\_BIT = BIT$, then $x = CONSEC + 1$.
  Otherwise, $x = 1$.
- If NEW_BIT + '0' $= t[T\_POS]$, then $y = T\_POS + 1$.
  Otherwise, $y = T\_POS$.
- Kill states corresponding to bad strings.

Full solution: DP

- Observe that we only need $LASTBIT$, $CONSEC$, and T_POS for the checking. This allows for a DP formulation with four parameters $(S\_POS, LASTBIT, CONSEC, T\_POS)$.
- Transition is done by appending '0' or '1' to the current string, whichever is allowed.
- Append NEW_BIT. New state: $(S\_POS + 1, NEW\_BIT, x, y)$
- If $NEW\_BIT = BIT$, then $x = CONSEC + 1$.
  Otherwise, $x = 1$.
- If NEW_BIT + '0' $= t[T\_POS]$, then $y = T\_POS + 1$.
  Otherwise, $y = T\_POS$.
- Kill states corresponding to bad strings.
- Time complexity: $O(NMK)$.

M1814 - Polynomial Quest

Alex Tung
alex20030190@yahoo.com.hk

3 March 2018

Problem Simplification

- The condition on A really means: for any $d = 0, 1, \ldots, N$, there exists a_i, $a_j \in A$, such that $a_i + a_j = d$.

Problem Simplification

- The condition on A really means: for any $d = 0, 1, \ldots, N$, there exists a_i, $a_j \in A$, such that $a_i + a_j = d$.
- Because when we expand $p_A(x)$, we get a sum of terms like $x^{a_i + a_j}$.

Problem Simplification

- The condition on A really means: for any $d = 0, 1, \ldots, N$, there exists a_i, $a_j \in A$, such that $a_i + a_j = d$.
- Because when we expand $p_A(x)$, we get a sum of terms like $x^{a_i + a_j}$.
- In simpler terms, $A + A$ contains $0, 1, \ldots, N$.

Subtask 1: $N \leq 8$

- Method 1: Solve by hand, hardcode

Subtask 1: $N \leq 8$

- Method 1: Solve by hand, hardcode
- Method 2: Print $0, 1, \ldots, \lfloor\sqrt{4 \times N + 1}\rfloor - 1$

Subtask 1: $N \leq 8$

- Method 1: Solve by hand, hardcode
- Method 2: Print $0, 1, \ldots, \left\lfloor \sqrt{4 \times N + 1} \right\rfloor - 1$
- Solving small cases give a "feeling" for the general solution

Subtask 2: $N \leq 18$

- Method 1: Solve by hand, hardcode

Subtask 2: $N \leq 18$

- Method 1: Solve by hand, hardcode
- Method 2: Exhaust all subsets of $\{0, 1, \ldots, N\}$, output suitable one

Subtask 2: $N \leq 18$

- Method 1: Solve by hand, hardcode
- Method 2: Exhaust all subsets of $\{0, 1, \ldots, N\}$, output suitable one
- Time complexity of Method 2: $O(2^N N^2)$

Subtask 3: $N = k^2$

- A can have $2k$ elements

Subtask 3: $N = k^2$

- A can have $2k$ elements
- Idea: Write numbers in base k

Subtask 3: $N = k^2$

- A can have $2k$ elements
- Idea: Write numbers in base k
- Choose $A = \{0, 1, \ldots, k-1, k, 2k, \ldots, k^2\}$

Subtask 3: $N = k^2$

- A can have $2k$ elements
- Idea: Write numbers in base k
- Choose $A = \{0, 1, \ldots, k-1, k, 2k, \ldots, k^2\}$
- $A + A$ contains $0, 1, \ldots, N = k^2, N+1, \ldots, k(k+1)$

Subtask 3.5: $N = k(k+1)$

- Consider one more type of input: $N = k(k+1)$

Subtask 3.5: $N = k(k+1)$

- Consider one more type of input: $N = k(k+1)$
- In this case, A can have $2k+1$ elements

Subtask 3.5: $N = k(k+1)$

- Consider one more type of input: $N = k(k+1)$
- In this case, A can have $2k+1$ elements
- Choose $A = \{0, 1, \ldots, k-1, k, 2k, \ldots, k^2, k^2+k\}$

Subtask 3.5: $N = k(k+1)$

- Consider one more type of input: $N = k(k+1)$
- In this case, A can have $2k+1$ elements
- Choose $A = \{0, 1, \ldots, k-1, k, 2k, \ldots, k^2, k^2+k\}$
- $A + A$ contains $0, 1, \ldots, N = k(k+1), N+1, \ldots, (k+1)^2 - 1$

The general case

- For any given integer N, we can find:
  - k such that $k^2 \leq N \leq k(k+1)$, or

The general case

- For any given integer N, we can find:
  - k such that $k^2 \leq N \leq k(k+1)$, or
  - k such that $k(k+1) \leq N < (k+1)^2$
- In the first case, replace N by k^2 and solve as in subtask 3

The general case

- For any given integer N, we can find:
  - k such that $k^2 \leq N \leq k(k+1)$, or
  - k such that $k(k+1) \leq N < (k+1)^2$
- In the first case, replace N by k^2 and solve as in subtask 3
- In the second case, replace N by $k(k+1)$ and solve as in subtask 3.5

The general case

- For any given integer N, we can find:
  - k such that $k^2 \leq N \leq k(k+1)$, or
  - k such that $k(k+1) \leq N < (k+1)^2$
- In the first case, replace N by k^2 and solve as in subtask 3
- In the second case, replace N by $k(k+1)$ and solve as in subtask 3.5
- The time complexity is $O(\sqrt{N})$

The general case

- For any given integer N, we can find:
  - k such that $k^2 \leq N \leq k(k+1)$, or
  - k such that $k(k+1) \leq N < (k+1)^2$
- In the first case, replace N by k^2 and solve as in subtask 3
- In the second case, replace N by $k(k+1)$ and solve as in subtask 3.5
- The time complexity is $O(\sqrt{N})$
- P.S. `No solution` will never appear :P