Contest debriefing

Scientific Committee

Result

Total: 50 teams

First 4 hours only	A	B	C	D	E	F	G	H	I	J	K	L
Solved / Tries	18/97 (19%)	38/71 (54%)	43/93 (46%)	7/33 (21%)	0/7 (0%)	20/44 (45%)	8/18 (44%)	3/10 (30%)	4/10 (40%)	47/49 (96%)	7/12 (58%)	41/117 (35%)
Average tries	3.73	1.69	1.98	2.75	1.75	1.76	2.00	1.43	1.67	1.04	1.50	2.34
Averages tries to solve	3.22	1.63	1.98	3.14	-	1.75	2.12	1.67	1.75	1.04	1.57	1.78

- Problem J: Free Food
- Problem C: SG Coin
- Problem L: Non-prime factors
- Problem B: Hopper
- Problem A: Largest Triangle

Ken

- Problem D: Bitwise
- Problem K: Conveyorbelts

Felix

- Problem I: Prolonged Password
- Problem E: Magical String

Suhendry

- (For problems G, H, F, please read the solution by yourself ☺)

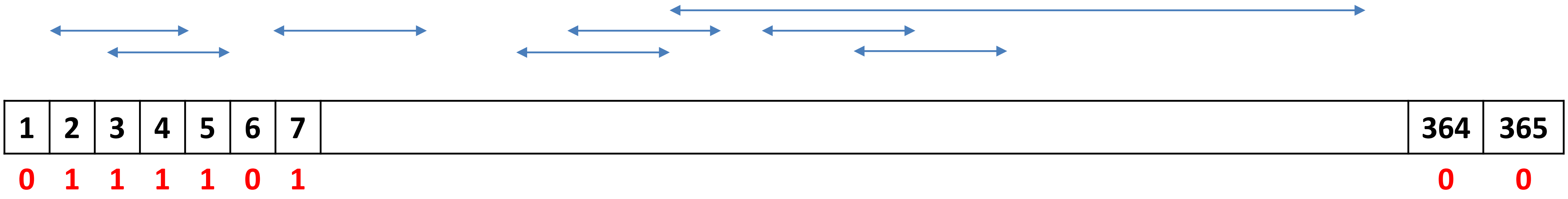
Free Food

Problem J

Author: Dr. Suhendry Effendy (NUS)
Tester: Dr. Felix Halim (Google), Dr. Steven Halim (NUS)

Problem

- Input: N intervals ($1 \leq N \leq 100$)
- Output: the number of days in which free food is served.



Solution

- Note that
  - there are only 100 intervals and
  - each interval is of length at most 365.
- We use brute-force solution.

- Initialize the bit array B[1..365] as zeros
- For each interval (s_i, t_i), mark $B[s_i..t_i] = 1$.
- Report the number of bits B[i] equal 1

1	2	3	4	5	6	7		364	365
0	0	0	0	0	0	0		0	0

Solution

- Note that
  - there are only 100 intervals and
  - each interval is of length at most 365.
- We use brute-force solution.

- Initialize the bit array B[1..365] as zeros
- For each interval (s_i, t_i), mark $B[s_i..t_i] = 1$.
- Report the number of bits B[i] equal 1

1	2	3	4	5	6	7		364	365
0	1	1	1	0	0	0		0	0

Solution

- Note that
  - there are only 100 intervals and
  - each interval is of length at most 365.
- We use brute-force solution.

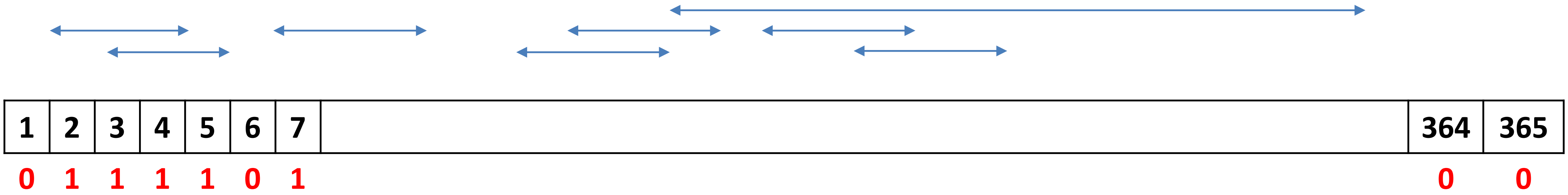
- Initialize the bit array B[1..365] as zeros
- For each interval (s_i, t_i), mark $B[s_i..t_i] = 1$.
- Report the number of bits B[i] equal 1

1	2	3	4	5	6	7		364	365
0	1	1	1	1	0	0		0	0

Solution

- Note that
  - there are only 100 intervals and
  - each interval is of length at most 365.
- We use brute-force solution.

- Initialize the bit array B[1..365] as zeros
- For each interval (s_i, t_i), mark $B[s_i..t_i] = 1$.
- Report the number of bits B[i] equal 1



SG Coin

Problem C

Author: Dr. Felix Halim (Google)
Tester: Dr. Suhendry Effendy (NUS), Dr. Steven Halim (NUS)

Problem

- Given a block Z with HashValue α_Z (with 9 digits and 7 trailing zeros), you need to generate two blocks A and B such that
  - $\alpha_A = H(\alpha_Z, S_A, T_A)$ has 9 digits and 7 trailing zeros; and
  - $\alpha_B = H(\alpha_A, S_B, T_B)$ has 9 digits and 7 trailing zeros.

prevHashValue: ??
Transaction: ??
Token: ??
HashValue: α_Z

Block Z

prevHashValue: α_Z
Transaction: S_A
Token: T_A
HashValue: $\alpha_A = H(\alpha_Z, S_A, T_A)$

Block A

prevHashValue: α_A
Transaction: S_B
Token: T_B
HashValue: $\alpha_B = H(\alpha_A, S_B, T_B)$

Block B

Simple solution

- 1. Randomly generate S_A and T_A.
- 2. Compute $\alpha_A = H(\alpha_Z, S_A, T_A)$
- 3. Randomly generate S_B and T_B.
- 4. Compute $\alpha_B = H(\alpha_A, S_B, T_B)$
- 5. Output S_A, T_A, S_B, T_B.

- The running time is slow since you need to compute H().

Block Z		Block A		Block B
prevHashValue: ?? Transaction: ?? Token: ?? HashValue: α_Z	→	prevHashValue: α_Z Transaction: S_A Token: T_A HashValue: $\alpha_A = H(\alpha_Z, S_A, T_A)$	→	prevHashValue: α_A Transaction: S_B Token: T_B HashValue: $\alpha_B = H(\alpha_A, S_B, T_B)$

Speedup 1: Use a short

- The running time of H() depends on the length of transaction. So, we use one character “X” for transaction.
- 1. Randomly generate T_A.
- 2. Compute α_A=H(α_Z,“X”,T_A)
- 3. Randomly generate T_B.
- 4. Compute α_B=H(α_A,“X”,T_B)
- 5. Output “X”, T_A, “X”, T_B.

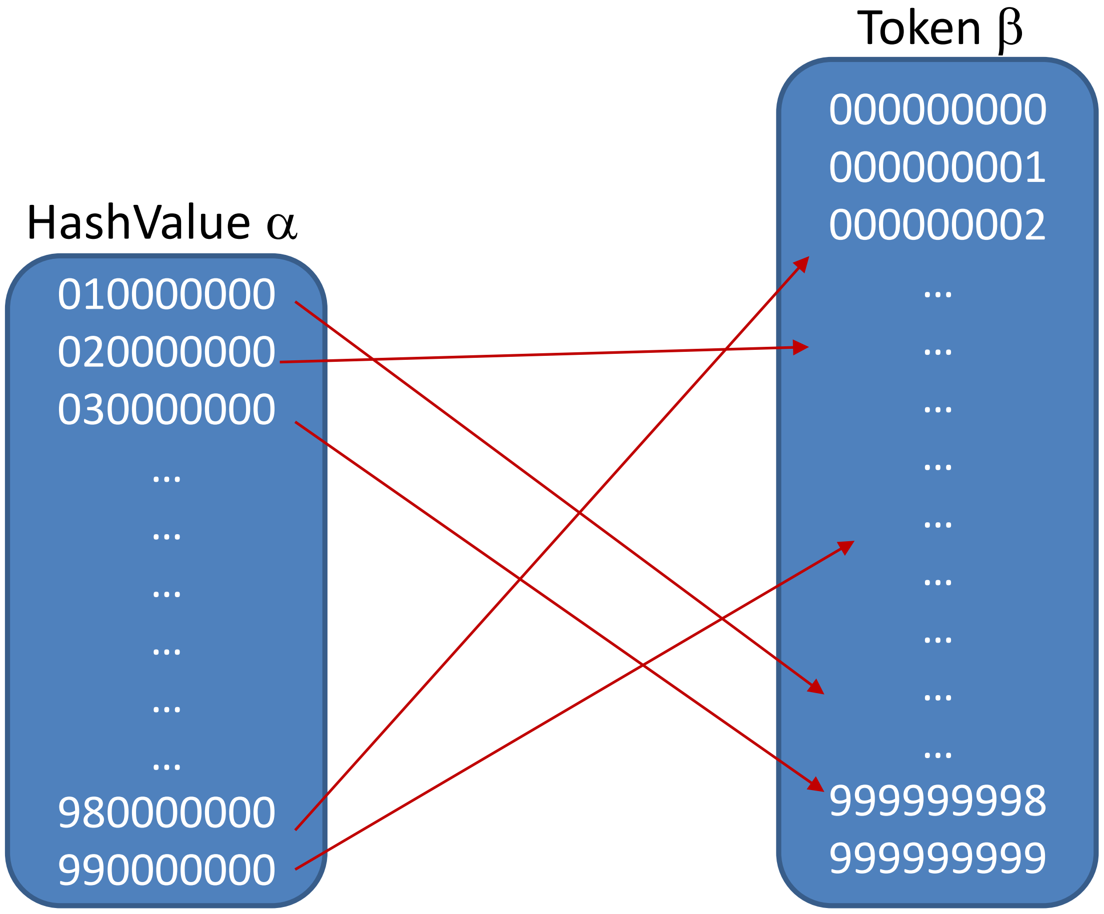
Block Z	Block A	Block B
prevHashValue: ??	prevHashValue: α_Z	prevHashValue: α_A
Transaction: ??	Transaction: “X”	Transaction: “X”
Token: ??	Token: T_A	Token: T_B
HashValue: α_Z	HashValue: α_A=H(α_Z, “X”, T_A)	HashValue: α_B=H(α_A, “X”, T_B)

Observation

- Since all HashValues have 9 digits and 7 trailing zeros, there are 99 different HashValues:
  - 010000000
  - 020000000
  - 030000000
  - …
  - 990000000

Build Lookup table

- For each hashValue α, we find a token such that $990000000=H(\alpha, \text{“X”}, \beta)$.
- Since there are only 100 HashValues, we can precompute a table T[] where
  - $T[\alpha]$ equals the token β such that $990000000=H(\alpha, \text{“X”}, \beta)$



$T[\alpha]=\beta$ such that $000000000=H(\alpha, \text{“X”}, \beta)$

Solution

- If the hashValue of block Z is α_Z, the output is
  - "X", T[α_Z]
  - "X", T[990000000]

- By table lookup, O(1) time.

Block Z	Block A	Block B
prevHashValue: ??	prevHashValue: α_Z	prevHashValue: 990000000
Transaction: ??	Transaction: "X"	Transaction: "X"
Token: ??	Token: T[α_Z]	Token: T[990000000]
HashValue: α_Z	HashValue: 990000000=H(α_Z, "X", T[α_Z])	HashValue: 990000000=H(α_A, "X", T[990000000])

Remark

- Accidentally, this problem is very similar to the problem H in Yangon 2018 (on last Sunday, 9 Dec).
- Note that we submit the problem last month.
- This is just a coincidence.

Non-Prime Factors

Problem L

Author: Dr. Steven Halim (NUS)
Tester: Dr. Felix Halim (Google), Dr. Suhendry Effendy (NUS)

Problem

- Input: an integer i
- Output: NPF(i), which is the number of non-prime factors of i.

- Example: i = 40.
  - 40 has 8 factors:
    - 1, <u>2</u>, 4, <u>5</u>, 8, 10, 20, 40.
  - 40 has 2 prime factors: 2, 5.
  - 40 has 6 non-prime factors:
    - 1, 4, 8, 10, 20, 40.
  - NPF(40)=6.

Theorem

- The prime factorization of i = ${p_1}^{q_1}{p_2}^{q_2} \dots {p_m}^{q_m}$.
- Then, the number of factors of i = $(q_1+1)(q_2+1)\dots(q_m+1)$.
- The number of prime factors of i = m.
- The number of non-prime factors of i = $(q_1+1)(q_2+1)\dots(q_m+1)$ - m.

- Example:
  – i = 40 = 2^3*5^1.
  – 40 has 8=(3+1)*(1+1) factors:
    - 2^0*5^0, $\underline{2^0*5^1}$, $\underline{2^1*5^0}$, 2^1*5^1, 2^2*5^0, 2^2*5^1, 2^3*5^0, 2^3*5^1.
  – 40 has 2 prime factors: $\underline{2^0*5^1}$, $\underline{2^1*5^0}$.
  – 40 has 6=(3+1)(1+1)-2 non-prime factors:
    - 2^0*5^0, 2^1*5^1, 2^2*5^0, 2^2*5^1, 2^3*5^0, 2^3*5^1.

Solution

- Given a number i,
  - For p = 2 to $\sqrt{i}$
    - Check if p is a prime factor if i.
  - Then, we obtain the prime factorization of i = $p_1{}^{q_1}p_2{}^{q_2} \ldots p_m{}^{q_m}$.
- This takes $\mathrm{O}\left(\sqrt{i}\right)$ time.

- After that, report NPF(i) = $(q_1+1)(q_2+1)\ldots(q_m+1)$ - m.

Another solution

- Given a number i,
  - Find all non-prime factors of i using a modified sieve of eratosthenes algorithm
  - Basically, run sieve of eratosthenes algorithm but cross out all the prime number
  - Then, we count the number of non-prime numbers

Further speedup

- It is still not fast enough!

- **Speedup 1**: File I/O is slow.
- C language: Instead of using cin/count, use scanf/printf.
- Java language: Instead of using Scanner/System.out.println, use BufferedReader/PrintWriter.

Further speedup

- **Speedup 2**: Observe that
  - There are at most $3*10^6$ queries.
  - The maximum value of i is $2*10^6$.
- By pigeon-hole principle, some queries NPF(i) are **duplicates**.

- To save computational time, you can store the answers in a hash table.

Remark

- Since this question requires a lot of I/O, **python** will die miserably.

Hoppers

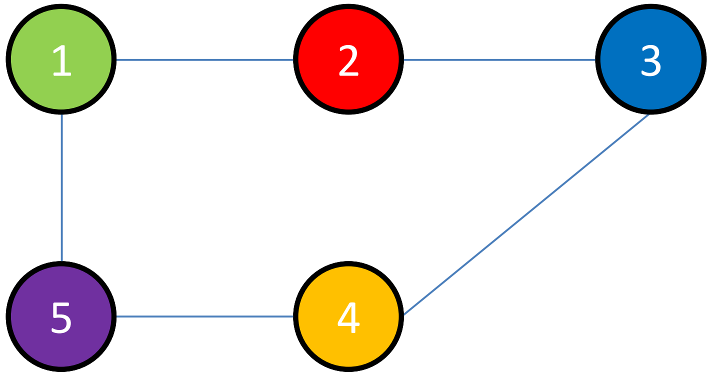
Problem B

Author: Hubert Teo Hua Kian (Stanford University)
Tester: Dr. Suhendry Effendy (NUS), Dr. Steven Halim (NUS)

Problem

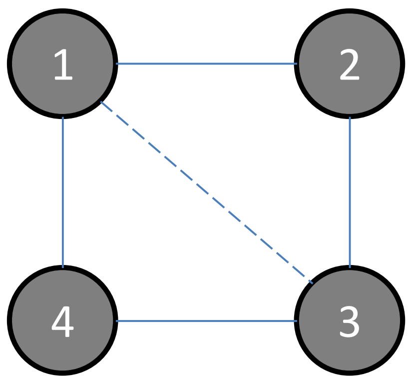
- Input: An undirected network with N nodes and M edges
- Malware ‘hopper’: If a node is infected, its neighbors’ neighbors will be infected.
- A network is unsafe if one node v is infected by ‘hopper’, all nodes in the network will be infected.
- Output: The minimum of number of additional edges to make the network unsafe.

- Example 1: Add zero edge to make G unsafe.
  - If we infect node 1,
  - Node 2 will be infected since 1-5-4-3-2 is of even length.
  - Node 3 will be infected since 1-2-3 is of even length.
  - Node 4 will be infected since 1-5-4 is of even length.
  - Node 5 will be infected since 1-2-3-4-5 is of even length.



Problem

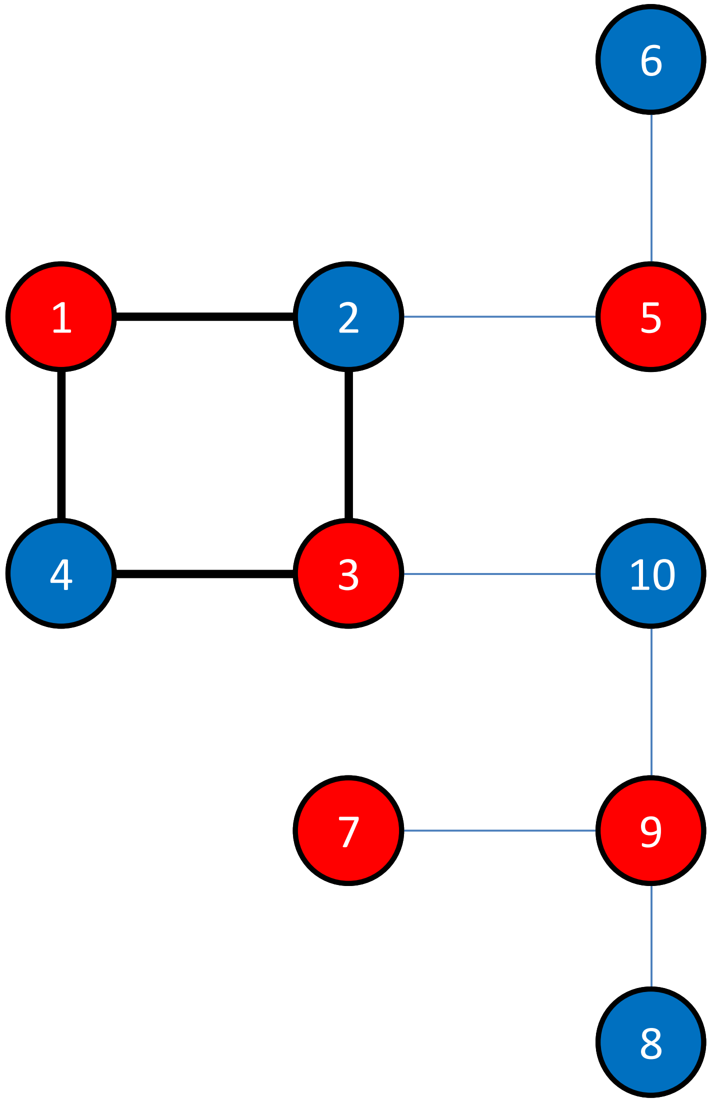
- Example 2: The original graph G is safe.
  - If we infect node 1,
    - Node 3 will be infected since 1-2-3 is of even length.
    - Cannot further propagate.
  - If we infect node 2,
    - Node 4 will be infected since 2-3-4 is of even length.
    - Cannot further propagate.



- After we add 1 edge (1, 3), G is unsafe.
  - If we infect node 1,
    - Node 2 will be infected since 1-3-2 is of even length.
    - Node 3 will be infected since 1-2-3 is of even length.
    - Node 4 will be infected since 1-3-2 is of even length.

Idea

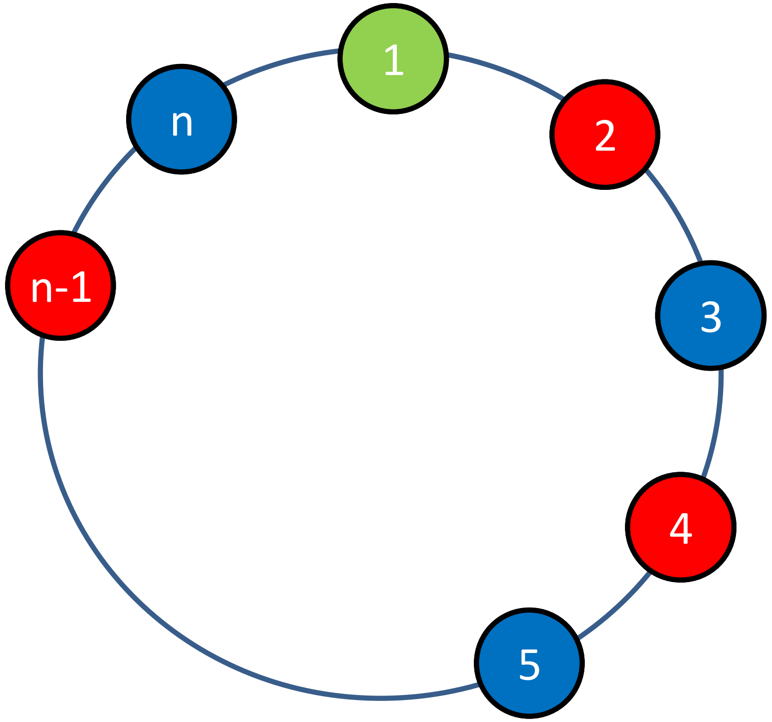
- Lemma: If G does not have odd cycle, then G is safe.
- Proof: If G does not have odd cycle, then G is 2-colorable, say red and blue.
- If you infect a red node, all red nodes will be infected but not blue nodes.
- If you infect a blue node, all blue nodes will be infected but not red nodes.
- So, G is safe.



Idea

- Lemma: Consider an odd cycle 1 – 2 – 3 – … – n. For any node j,
  - Either 1-2-3-…-j or 1-n-(n-1)-…-j is of even length.

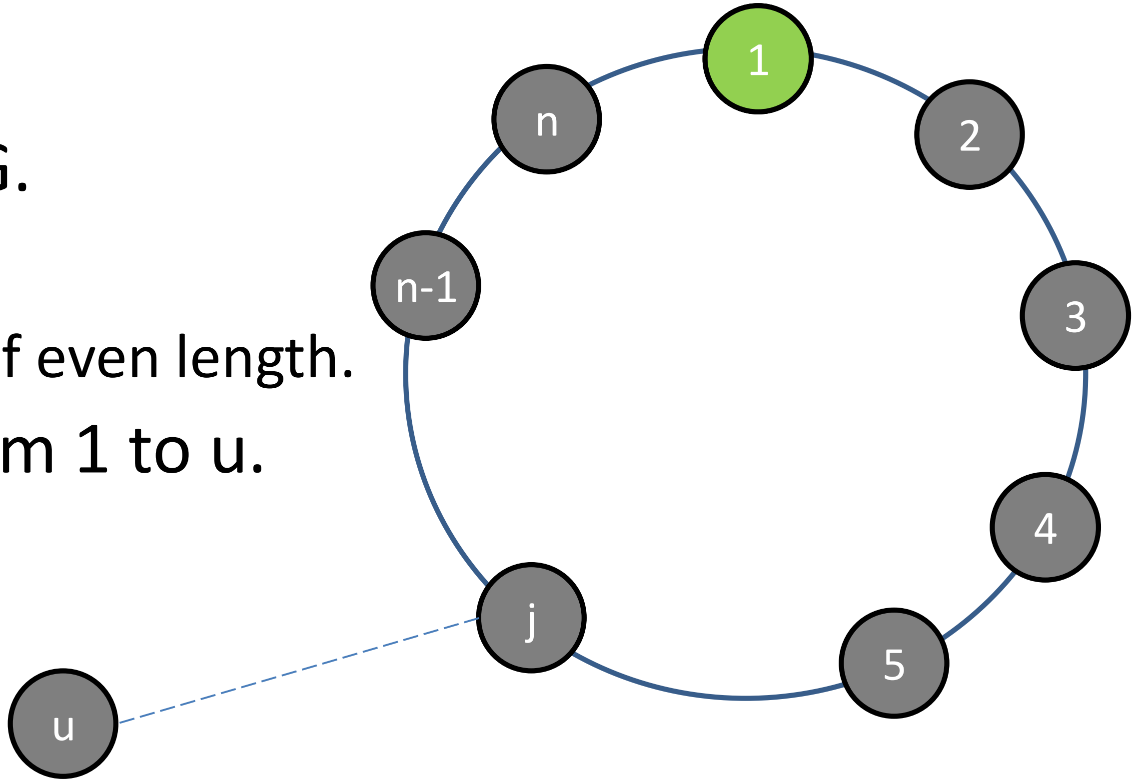
- Proof:
- For odd j,
  - 1-2-3-…-j is of even length.
- For even j,
  - 1-n-(n-1)-…-j is of even length.



Idea

- Lemma: Suppose the graph G is connected and has an odd cycle. G is unsafe.
  – After we infect a node v in the odd cycle, all nodes will be infected.

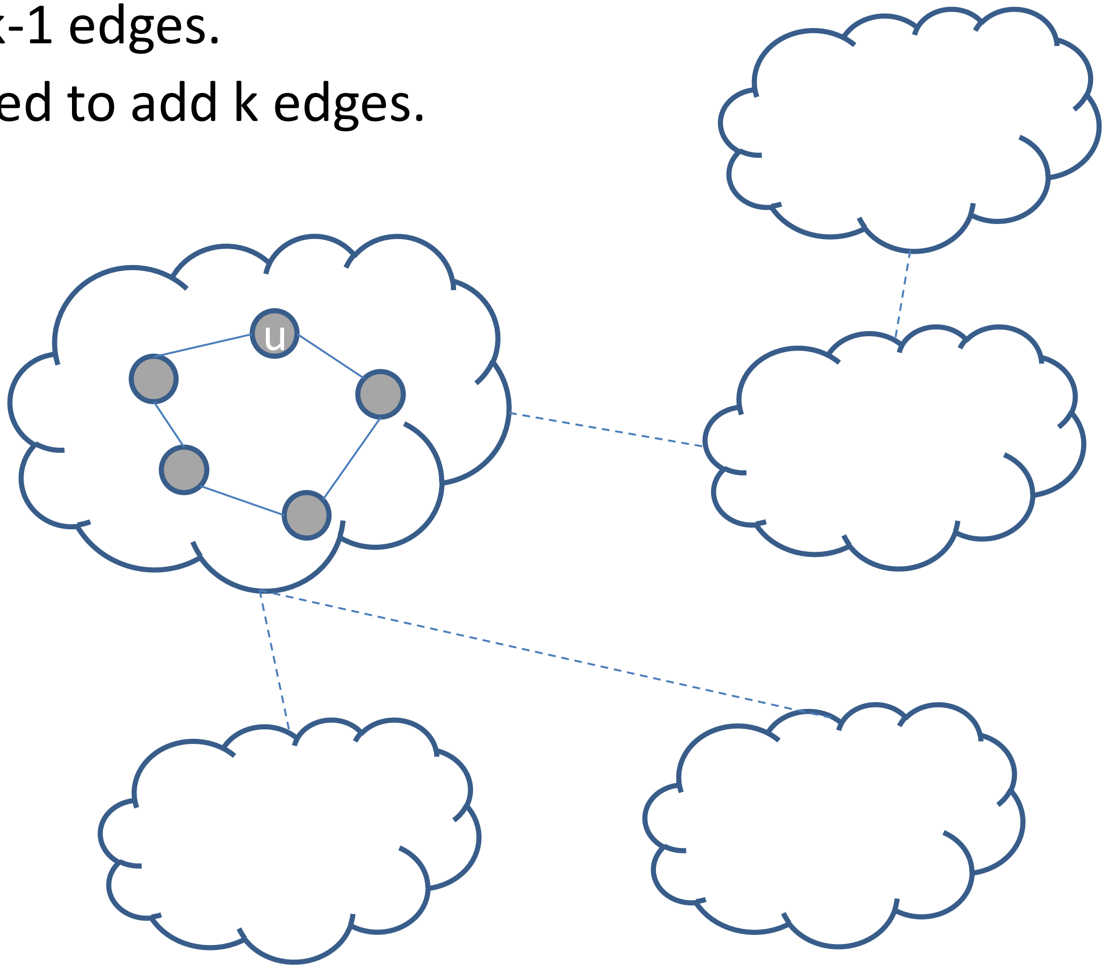
- Proof: Let 1-2-…-n be the odd cycle in G.
- For any node u in G,
  – either 1-2-…-j-…-u or 1-n-(n-1)-…-j-…-u is of even length.
- Hence, there is an even-length path from 1 to u.
- All nodes are infected.
- G is unsafe.



Theorem

- Lemma: Suppose the graph G has k connected component.
  - Case 1: If G has an odd cycle, we need to add k-1 edges.
  - Case 2: If G does not have an odd cycle, we need to add k edges.

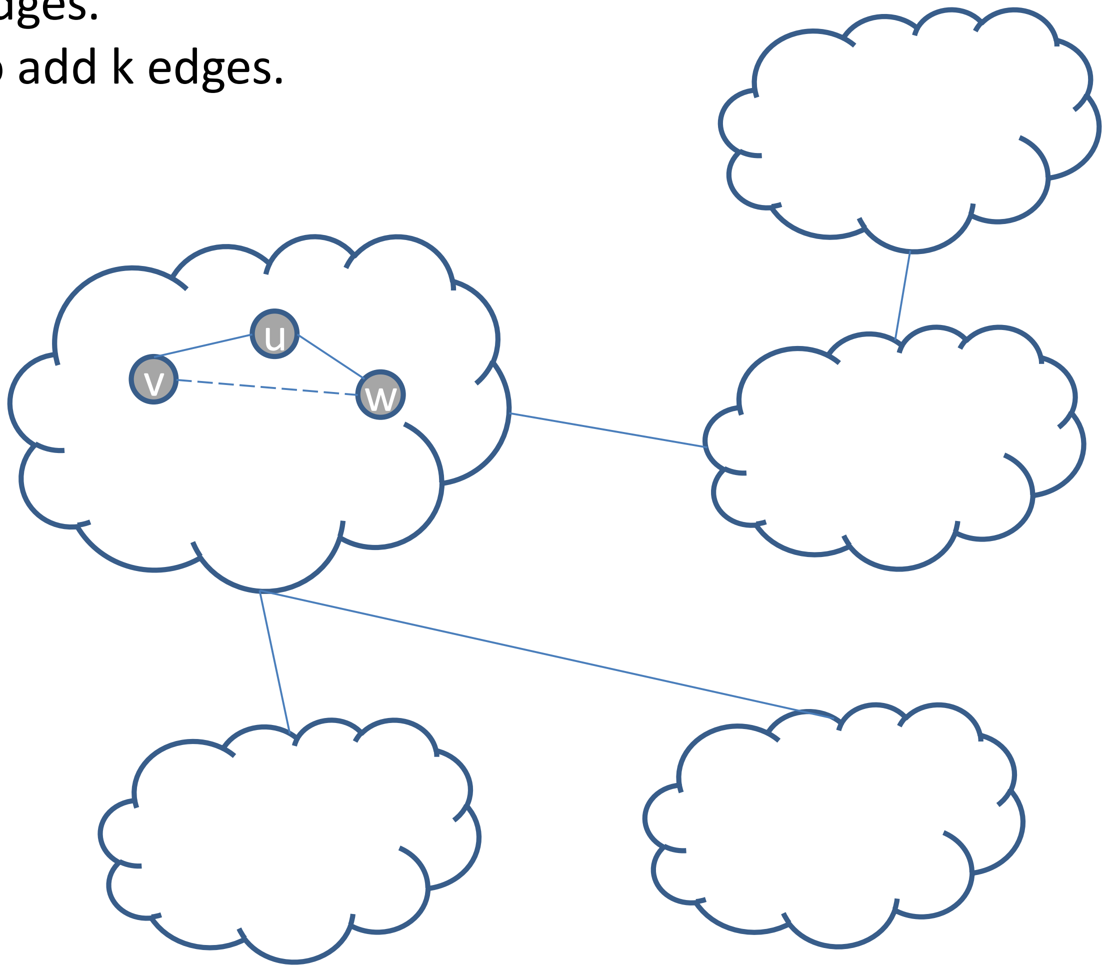
- Proof for case 1:
- We add k-1 edges to link all k components.
- If we infect u, u has an length-even path to all nodes in G.
- All nodes will be infected.



Theorem

- Lemma: Suppose the network G has k connected component.
  - Case 1: If G has an odd cycle, we need to add k-1 edges.
  - Case 2: If G does not have an odd cycle, we need to add k edges.

- Proof for case 2:
- We add k-1 edges to link all k components.
- There is no odd cycle.
- So, the network is still unsafe.
- We add a link (v, w).
- u-v-w is a triangle, odd-length cycle.
- All nodes will be infected.



Solution

- 1. Let k be the number of connected components
- 2. By DFS (or BFS), detect if there is an odd cycle.
- 3. If there is an odd cycle,
  - Report k-1
  - Otherwise, report k.

- This algorithm runs in O(N+M) time.

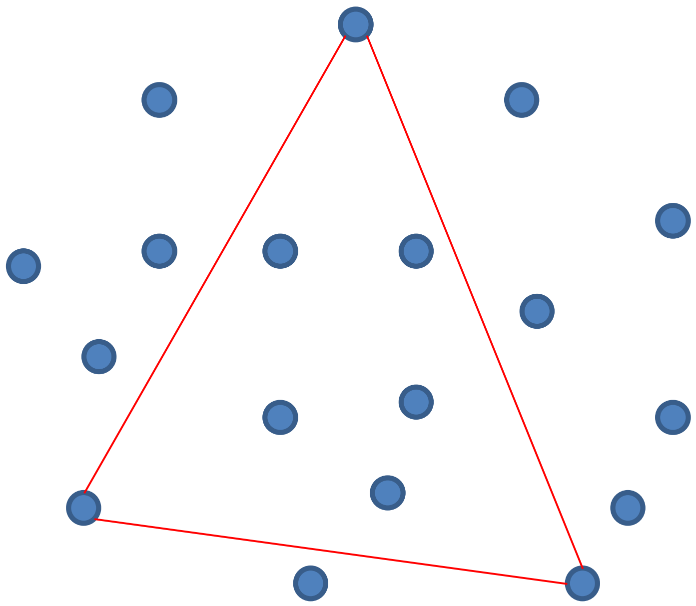
Largest Triangle

Problem A

Author: Dr. Steven Halim (NUS)
Tester: Dr. Felix Halim (Google), Dr. Suhendry Effendy (NUS)

Problem

- Input: A set of points.
- Output: The area of the largest triangle.



Naïve solution

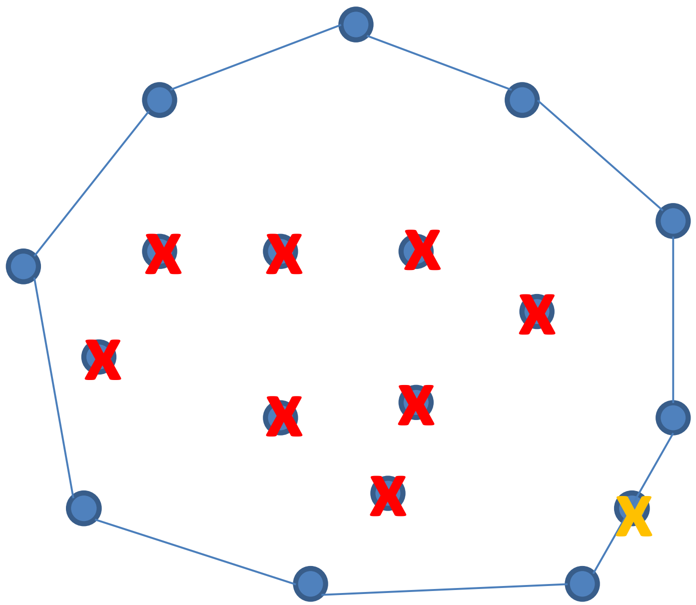
- Enumerate all 3 points.
- Find the one with the biggest area.

- This solution takes $O(N^3)$ time.
- It rendered Time-Limit-Exceeded (TLE)

A better solution

- We can reduce the number of points by filter out:
  - Duplicate points
  - Points not in convex hull
  - Points that are collinear

- However, it is still not fast enough.

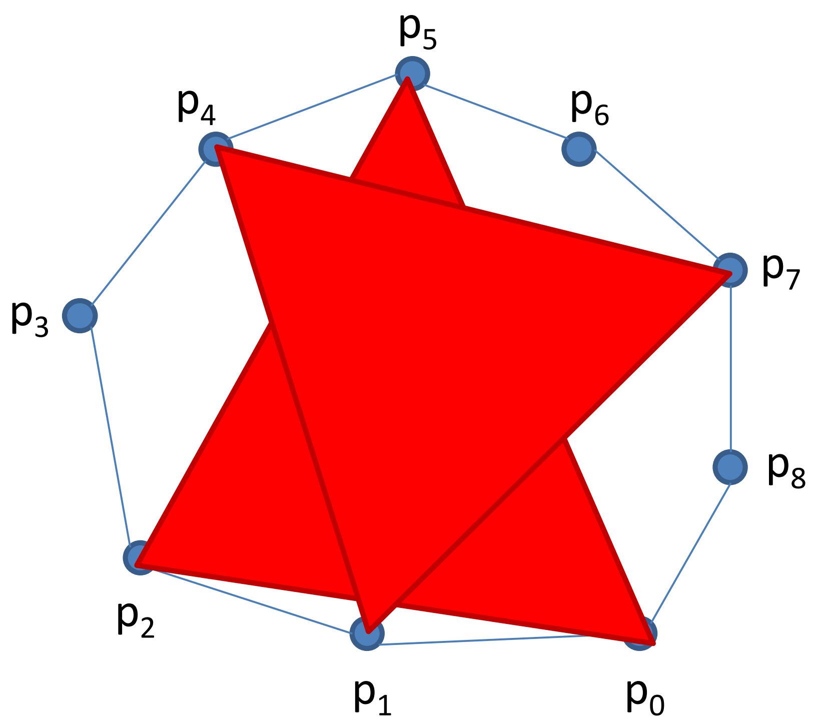


Idea of the solution

- A triangle is said rooted at a if one of its endpoint is a.
- Let the convex hull be P = p_0, p_1, …, p_n.

- Area = 0
- For i = 0 to n
  - Set A_i = area of the largest triangle rooted at p_i.
  - If (A_i > Area) then Area = A_i
- Report Area

- Below, we show that "area of the largest triangle rooted at p_i" can be computed in O(n) time.
- So, we give an $O(n^2)$ time algorithm.



Find the largest triangle rooted at a

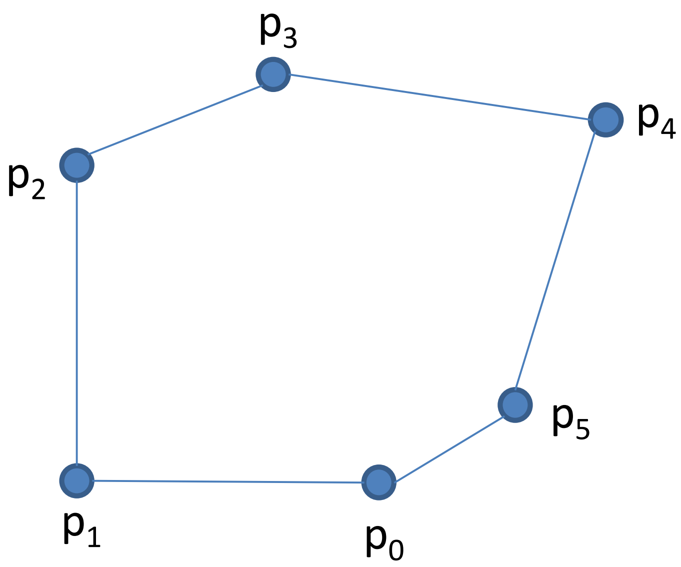
- Area of the largest triangle rooted at ‘a’ can be found using an idea similar to the rotating caliper algorithm

Find the largest triangle rooted at a

- Let the convex hull be $p_0, p_1, \ldots, p_N$.

- Set $a=p_0$, $b=p_1$, $c=p_2$
- Area = Δabc
- While ($c \neq p_N$)
  - c'=next(c)
  - While ($\Delta abc' \geq \Delta abc$)
    - If ($\Delta abc' >$ Area) then Area = $\Delta abc'$
    - $c = c'$
  - b = next(b)
- Return Area

- This algorithm runs in O(N) time.



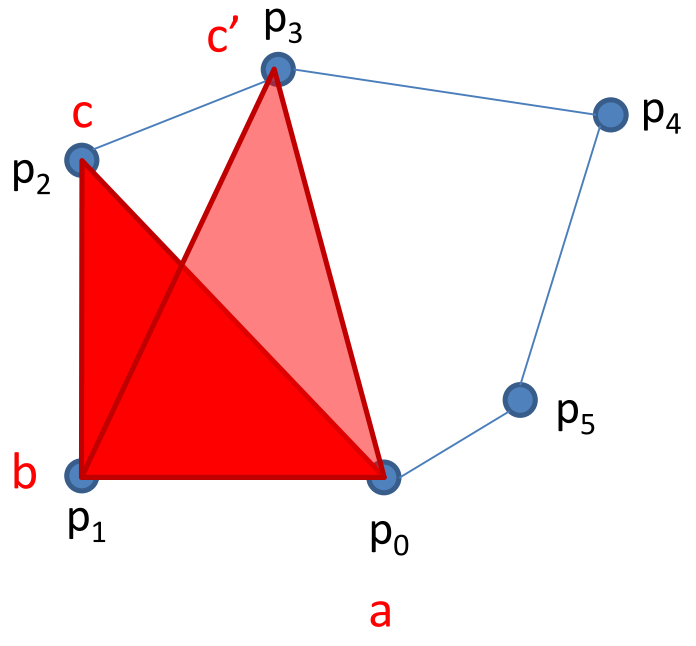
Keikha et al. Maximum-Area Triangle in a Convex Polygon, Revisited. 2017.

Find the largest triangle rooted at a

- Let the convex hull be p_0, p_1, …, p_N.

- Set $a=p_0$, $b=p_1$, $c=p_2$ ←
- Area = Δabc ←
- While ($c \neq p_N$)
  - c'=next(c) ←
  - While ($\Delta abc' \geq \Delta abc$)
    - If ($\Delta abc'$ > Area) then Area = $\Delta abc'$ ←
    - c = c'
  - b = next(b)
- Return Area

- This algorithm runs in O(N) time.



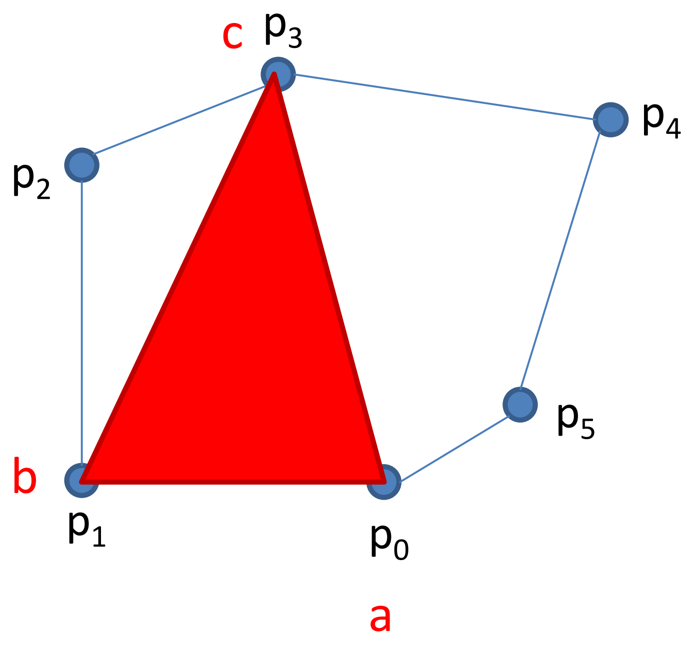
Area = $\Delta p_0p_1p_3$

Find the largest triangle rooted at a

- Let the convex hull be $p_0, p_1, \ldots, p_N$.

- Set $a=p_0$, $b=p_1$, $c=p_2$
- Area = Δabc
- While ($c \neq p_N$)
  - $c'=\text{next}(c)$
  - While ($\Delta abc' \geq \Delta abc$)
    - If ($\Delta abc' >$ Area) then Area = $\Delta abc'$
    - $c = c'$ ←
  - $b = \text{next}(b)$
- Return Area

- This algorithm runs in O(N) time.



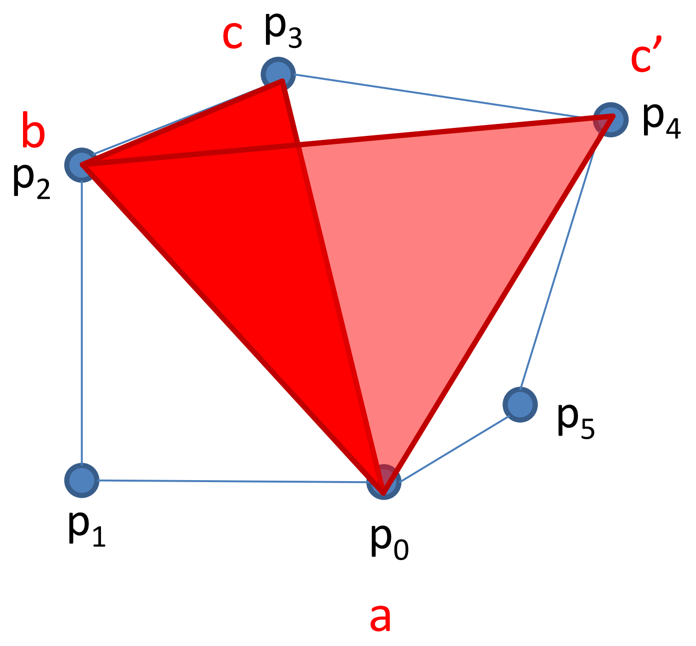
Area = $\Delta p_0p_1p_3$

Find the largest triangle rooted at a

- Let the convex hull be $p_0, p_1, \ldots, p_N$.

- Set $a=p_0$, $b=p_1$, $c=p_2$
- Area = Δabc
- While ($c \neq p_N$)
  - c'=next(c) ←
  - While ($\Delta abc' \geq \Delta abc$)
    - If ($\Delta abc'$ > Area) then Area = $\Delta abc'$ ←
    - $c = c'$
  - b = next(b) ←
- Return Area

- This algorithm runs in O(N) time.



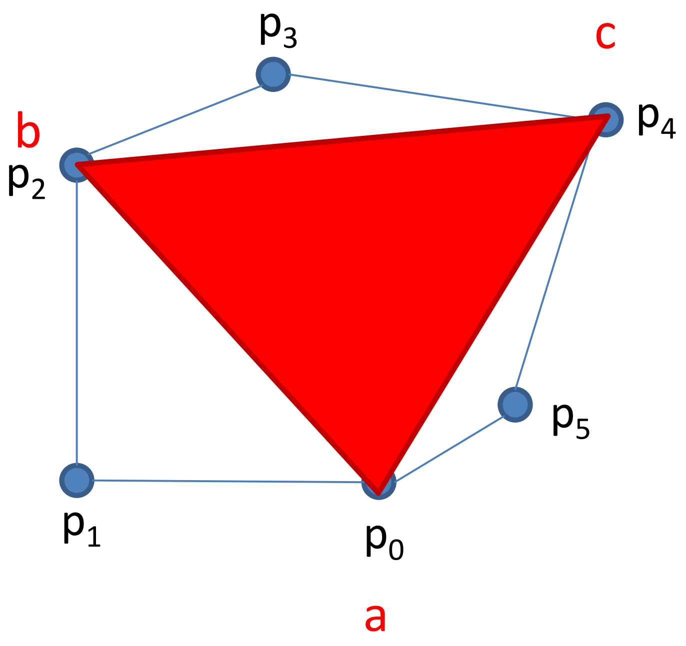
Area = $\Delta p_0p_2p_4$

Find the largest triangle rooted at a

- Let the convex hull be $p_0, p_1, \ldots, p_N$.

- Set $a=p_0$, $b=p_1$, $c=p_2$
- Area = Δabc
- While ($c \neq p_N$)
  - c'=next(c)
  - While ($\Delta abc' \geq \Delta abc$)
    - If ($\Delta abc'$ > Area) then Area = $\Delta abc'$
    - c = c' ←
  - b = next(b)
- Return Area

- This algorithm runs in O(N) time.



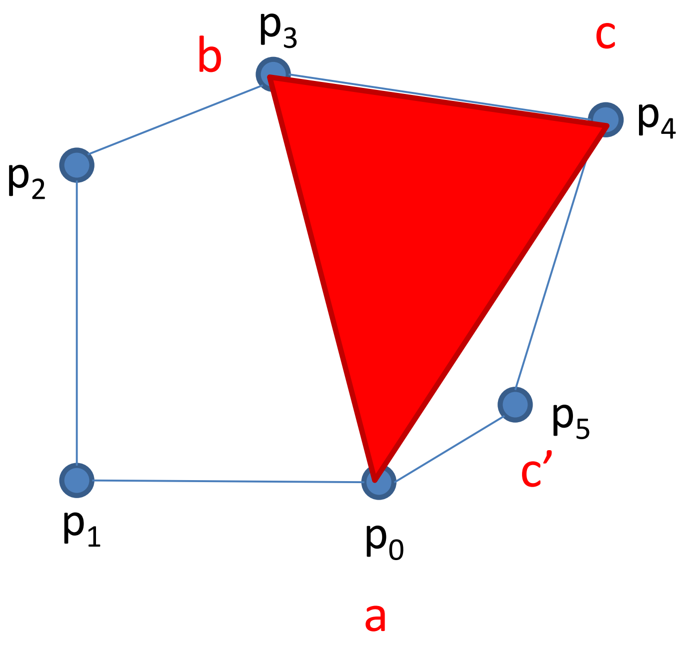
Area = $\Delta p_0 p_2 p_4$

Find the largest triangle rooted at a

- Let the convex hull be $p_0, p_1, \ldots, p_N$.

- Set $a=p_0$, $b=p_1$, $c=p_2$
- Area = Δabc
- While ($c \neq p_N$)
  - c'=next(c) ←
  - While ($\Delta abc' \geq \Delta abc$)
    - If ($\Delta abc'$ > Area) then Area = $\Delta abc'$
    - $c = c'$
  - b = next(b) ←
- Return Area

- This algorithm runs in O(N) time.



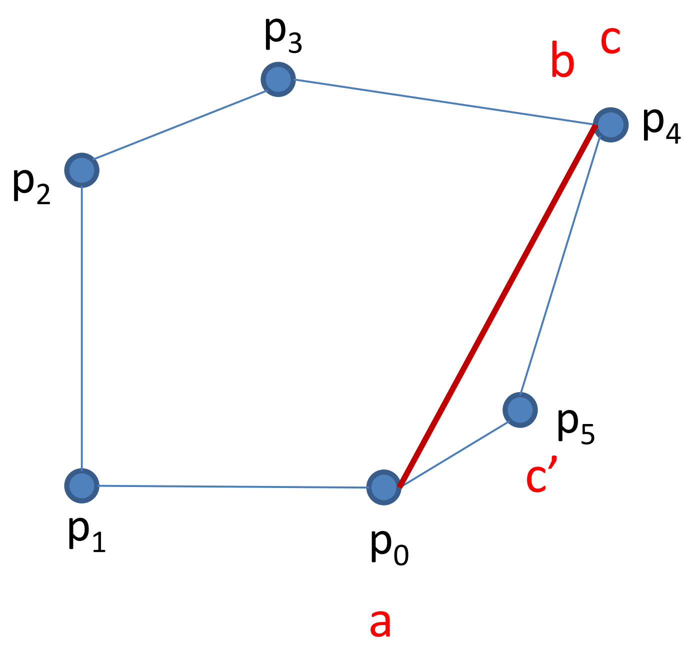
Area = $\Delta p_0 p_2 p_4$

Find the largest triangle rooted at a

- Let the convex hull be $p_0, p_1, \ldots, p_N$.

- Set $a=p_0$, $b=p_1$, $c=p_2$
- Area = Δabc
- While ($c \neq p_N$)
  - c'=next(c) ←
  - While ($\Delta abc' \geq \Delta abc$)
    - If ($\Delta abc'$ > Area) then Area = $\Delta abc'$
    - c = c'
  - b = next(b) ←
- Return Area

- This algorithm runs in O(N) time.



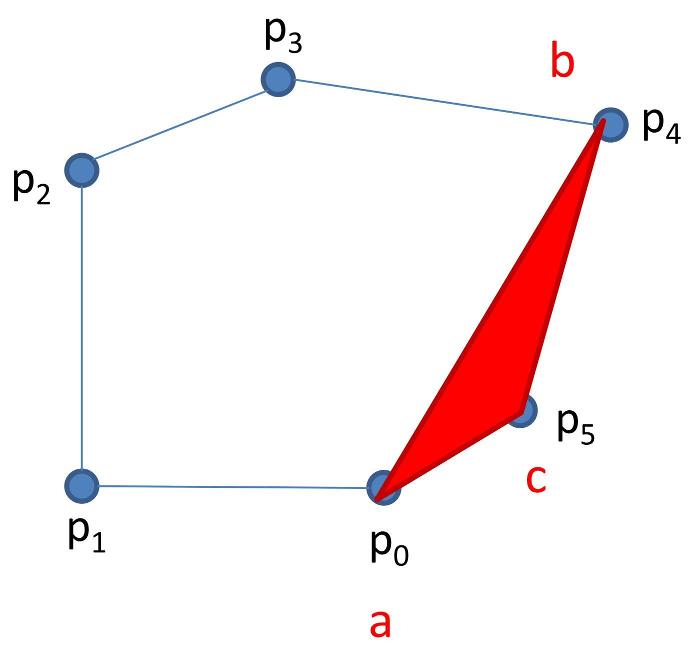
Area = $\Delta p_0p_2p_4$

Find the largest triangle rooted at a

- Let the convex hull be $p_0, p_1, \ldots, p_N$.

- Set $a=p_0$, $b=p_1$, $c=p_2$
- Area = Δabc
- While ($c \neq p_N$)
  - c'=next(c)
  - While ($\Delta abc' \geq \Delta abc$)
    - If ($\Delta abc'$ > Area) then Area = $\Delta abc'$
    - c = c' ←
  - b = next(b)
- Return Area

- This algorithm runs in O(N) time.



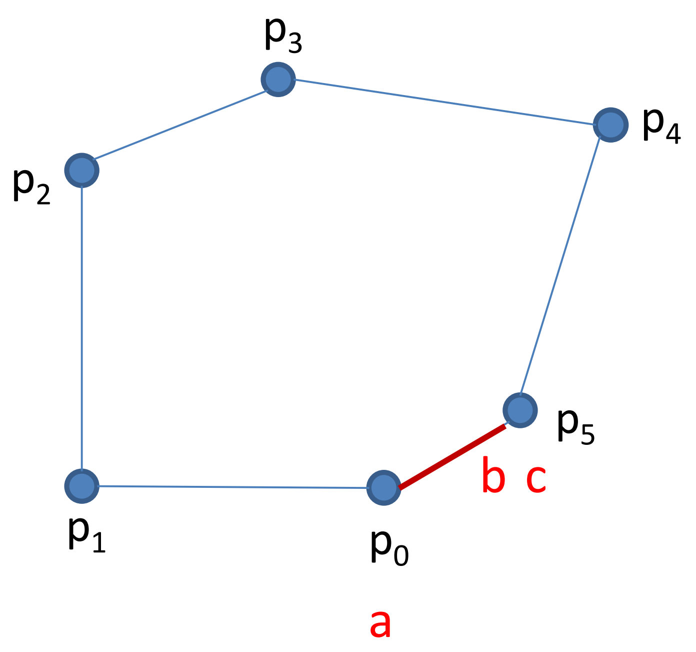
Area = $\Delta p_0 p_2 p_4$

Find the largest triangle rooted at a

- Let the convex hull be $p_0, p_1, \ldots, p_N$.

- Set $a=p_0$, $b=p_1$, $c=p_2$
- Area = Δabc
- While ($c \neq p_N$)
  - c'=next(c)
  - While ($\Delta abc' \geq \Delta abc$)
    - If ($\Delta abc'$ > Area) then Area = $\Delta abc'$
    - $c = c'$
  - b = next(b) ←
- Return Area ←

- This algorithm runs in O(N) time.



Area = $\Delta p_0p_2p_4$

Even faster solution

- $O(n^2)$ solution can pass all test cases.

- This problem actually can be solved in $O(n \log n)$ time.
  - Keikha et al. Maximum-Area Triangle in a Convex Polygon, Revisited. 2017.
  - https://arxiv.org/pdf/1705.11035.pdf

- The above paper also showed that idea based on the "modified rotating caliper algorithm" cannot give an $O(n)$ time.

Remark

- 1. This is the only geometry problem in the set, added to diversify the problem types.

- 2. For large test cases in this problem, it requires to generate many points in a convex hull.
  - We actually use the solution in ICPC.SG.2015 to generate the large test cases.
  - https://open.kattis.com/problems/convex

Acknowledgement (related to Scientific committee)

- Problem setters
  - **Sean Pek Yu Xuan** (NUS, SGP, prelim tasks only),
  - **Shafaet Ashraf** (Traveloka, SGP),
  - **Dr Felix Halim** (Google, USA; ICPC World Finalist 2007),
  - **Raymond Kang Seng Ing** (Google, USA; 2x ICPC World Finalist 2015, 2018),
  - **Hubert Teo Hua Kian** (Stanford University, USA; 2x ICPC World Finalist 2016, 2018),
  - **Chin Zhan Xiong** (Nuro, USA; 2x ICPC World Finalist 2017, 2018),
  - **Nguyen Tan Sy Nguyen** (Anduin Transactions, VNM; 2x ICPC World Finalist 2013, 2016),
  - **Kyle See** (Augmented Intelligence Pros, Philippines, World Finalist 2016),
  - **Irvan Jahja** (NUS, SGP, 2x ICPC World Finalist 2012, 2013),
  - **Dr Suhendry Effendy** (NUS, SGP, ICPC Asia Jakarta Head Judge 2010-2016),
  - **Dr Steven Halim** (NUS, SGP, 6x ICPC World Finalist (Coach 3x/RCD 2x/Attendee 1x) 2010, 2012, 2014, 2015, 2016, 2018).

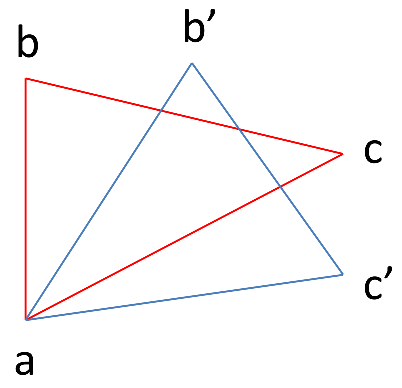
- Tester
  - **Dr Suhendry Effendy** (NUS, SGP, ICPC Asia Jakarta Head Judge 2010-2016),
  - **Dr Felix Halim** (Google, USA; ICPC World Finalist 2007),
  - **Dr Steven Halim** (NUS, SGP, 6x ICPC World Finalist (Coach) 2010, 2012, 2014-16, 2018),
  - **Shen Chuanqi** (Google, USA; ICPC World Finalist 2015),
  - **Nguyen Thanh Trung** (Google, SGP; ICPC World Finalist 2012, 2016).

- Honorary Judges from Kattis team
  - **Dr Fredrik Niemelä**,
  - **Associate Professor Greg Hamerly**.

2-stable triangle rooted at a

- Let the convex hull be $P = p_0, p_1, \ldots, p_n$. Fixed $a=p_0$.
- A triangle is said rooted at a if one of its endpoint is a.
- A triangle Δabc rooted at a is said to be 2-stable if
  - $\Delta ab'c, \Delta abc' \leq \Delta abc$ for all b' and c'.

- Lemma: Suppose Δabc and $\Delta ab'c'$ are 2-stable. We have:
  - $b \leq b' \leq c \leq c'$ or $b' \leq b \leq c \leq c'$ in P



Keikha et al. Maximum-Area Triangle in a Convex Polygon, Revisited. 2017.

Bitwise

Bitwise

Given:

- Sequence of **N** integers: $\mathbf{A_1}, \mathbf{A_2}, \ldots, \mathbf{A_n}$
- The integers is forming a circle
- The sequence is divided (partitioned) into **K** sections
- power(section) = the **bitwise OR** of all integers in that section

Determine:

- The **maximum** **bitwise AND** of the powers of the sections in an optimal partition of the circle of integers

$1 <= \mathbf{K} <= \mathbf{N} <= 5*10^5$, $0 <= \mathbf{A_i} <= 10^9$

Bitwise

Reverse the thinking:

- Given an integer **X**, can you divide the sequence so that the **bitwise AND** of the powers of the sections is at least **X**?
- Imagine there is a function **can(X)** that can answer the previous question
- Then we can "**greedy the answer**":

```
int ans = 0;
for (int i = 30; i >= 0; i--) {
  int bit = 1 << i;
  if (can(ans | bit)) {
    ans |= bit;
  }
}
printf("%d\n", ans);
```

Bitwise

can(X): How to divide the sequence so that the **bitwise AND** of the powers of the sections is at least **X**?

- Simulation:
    - Pick a starting point in the sequence and start performing **bitwise OR** onwards until the accumulator exceeds **X**, then you found a section.
    - From the last point, continue the process to find the next sections until you go back to the starting point.
    - See if you managed to find at least **K** sections?
- How many starting points are there?
    - There are at most $\log(10^9) = 31$ different starting points

Total complexity $O(N * 31 * 31) = O(N)$

Conveyor Belts

Conveyor Belts

Given:

- **N** junctions connected by **M** conveyor belts
- **K** producers located at the first **K** junctions
- Producer **j** produces a product each minute ($\mathbf{x} \cdot \mathbf{K}+\mathbf{j}$) where $\mathbf{x} \geq 0$ and $j=1,2,\ldots,K$.
- There is ***a deterministic route*** from a producer to the warehouse (junction N)
- Each conveyor belt only transports at most one product at any time
- No limit on the number of products at the junctions

Determine:

- Find the maximum number of producers which can be left running such that all the produced products can be delivered to the warehouse

1 <= **K** <= **N** <= 300, 0 <= **M** <= 1000

Conveyor Belts

Observation:

- This is a graph problem (junction -> node, conveyor belt -> edge)
- How do we encode this constraint in our graph:
    - Each conveyor belt only transports at most one product **at any time**
- We can encode the **“time” dimension** by blowing up a junction into **K** nodes
    - Junction **A** is represented as **K** nodes in the graph (node **A** at time 0, 1, … **K**-1)
        - The time wraps around. That is, time **K** is equivalent to time 0
    - A conveyor belt connecting from junction **A** to junction **B** is represented as
        - **K** edges: one edge from node **A** at time **i** to node **B** at time **(i + 1) % K**

Conveyor Belts

Maximum flow solution:

- Add two new nodes (a **source** node and a **sink** node)
- Connect the source node to all K producers
  - Add an edge from the **source** to **Producer i at time i** with **capacity 1**
- Connect the warehouse at all time periods to a sink with **infinite capacity**
  - Add an edge from Junction **N** at time **i** (for all i = 0..K-1) to the **sink**
- Run **maximum flow** from the source to the sink
  - The maxflow value is the number of producers that can be left running
  - Use **Dinic’s algorithm** to avoid getting time limit exceeded
    - The runtime is proportional to the maxflow value (max = **K**)

Prolonged Password

Prolonged Password

Given:

- A string **S** of alphabet characters.
- A function **f(S,T)** which transforms each character S_i into a string T_{Si}.
- An integer **K** denoting how many times **f(S,T)** is performed, i.e. $f^K(S,T)$.
- An integer **M** denoting the number of queries.
  - Each query contains an integer m_i.

Determine:

❖ For each query, the m_i^{th} character of $f^K(S,T)$

$1 \le |S| \le 10^6$; $2 \le |T_x| \le 50$; $1 \le K \le 10^{15}$; $1 \le M \le 1000$; $1 \le m_i \le 10^{15}$.

Prolonged Password

Example:

S = bccabac

T_a = ab

T_b = bac

T_c = ac

T_d .. T_z are not important in this example.

a → ab
b → bac
c → ac

$f^0(S,T)$ = bccabac

$K = 1 \rightarrow f^1(S,T)$ = bacacacabbacabac

$K = 2 \rightarrow f^2(S,T)$ = bacabacabacabacabbacbacabacabbacabac

Prolonged Password

- How to generate $f^K(S,T)$ for large K?
  - K can be very large, i.e. 10^{15} → a hint for $O(\log K)$ solution

- How to store $f^K(S,T)$?
  - Recall the constraints: $1 \le |\mathbf{S}| \le 10^6$ and $2 \le |\mathbf{T_x}| \le 50$
  - The complete $f^K(S,T)$ can be $\mathbf{10^6 \cdot 50^{10^{15}}}$
  - Each query falls within the first 10^{15} characters → we cannot store 10^{15} characters
  - We need to output only ONE character per query → we have to exploit this.

Prolonged Password

- We don't need to generate the whole $f^K(S,T)$.
  - Define $= |f^K(S,T)|$
  - Iterate through the string S to find out which character we should recurse down into.
  - E.g.,

a	b	a	a	c
30	20	30	30	50

Then, the 85th character can be obtained by expanding 'a' at index-3.

- $O\left(MK \max_i |T_i| + M|S|\right)$

Prolonged Password

To handle large K: Matrix Exponentiation

N_{aa} = count of character 'a' in T_a.

N_{ab} = count of character 'b' in T_a.

…

N_{za} = count of character 'a' in T_z.

N_{zb} = count of character 'b' in T_z.

r_a = count of character 'a'.

r_b = count of character 'b'.

…

r_z = count of character 'z'.

$$(r_a \quad \cdots \quad r_z)\begin{pmatrix} N_{aa} & \cdots & N_{za} \\ \vdots & \ddots & \vdots \\ N_{az} & \cdots & N_{zz} \end{pmatrix}$$

$$l^0(c,T) = r$$
$$l^1(c,T) = r \cdot N$$
$$l^2(c,T) = r \cdot N \cdot N$$
…
$$l^K(c,T) = r \cdot N^K$$

$$len^K(c,T) = \|l^K(c,T)\|_1$$

Prolonged Password

Another problem: K is too large, $len^K(S,T)$ will be overflow.

Observation:

- 2 ≤ $|T_i|$ → it means the string length doubles at each iteration.
- $2^{10^{15}}$ is way too large, but $m_i \leq 10^{15}$
- $10^{15} \leq 2^{50}$
- We can cut down K by exploiting **cycle** in the transformation function.

a → bda

b → cdc　　　　a → b → c → a

c → ab

Prolonged Password

Summary:

- Cut down K to ≤ 50.
- Solve by recursing and using matrix exponentiation.

Prolonged Password

Summary:

- Cut down K to ≤ 50.
- Solve by recursing and using matrix exponentiation.

However, if you solve each query independently, you will get **TLE** as M ≤ 1000.

→ You need to solve all queries at once (in one pass).

Magical String

Magical String

Given:

- A string S which has no substring containing 3 or more identical characters.
- An integer K, the number of maximum operations.

An operation on S: Convert S_i into another character (non-asterisk) s.t. S contains a substring of 3 or more identical characters. Turn such (maximal) substring into an asterisk.

Determine:

❖ The maximum number of characters in S which can be turned into asterisks with at most K operations.

$1 \leq K, |S| \leq 1000$

Magical String

Example:

S = abacaac

If K = 1

abacaac → abaaaac : ab*c

ANS: 4

If K = 2

abacaac → aaacaac : *caac → *caaa : *c*

ANS: 6

Magical String

Example:

S = abacaac

If K = 1

abacaac → abaaaac : ab*c

ANS: 4

If K = 2

abacaac → aaacaac : *caac → *caaa : *c*

ANS: 6

This example suggests that the solution is **not** incremental, i.e. the solution for (S,K) does not necessarily use the solution for (S,< K)

Magical String

Example:

S = abacaac

If K = 1

abacaac → abaaaac : ab*c

ANS: 4

If K = 2

abacaac → aaacaac : *caac → *caaa : *c*

ANS: 6

This example suggests that
the solution is **not** incremental,
i.e. the solution for (S,K) does not
necessarily use the solution for (S,< K)

↓

Greedy does not work!

Also, the operations order does matter.

Magical String

first attempt … dynamic programming

f(S, K) → The maximum number of characters in S which can be turned into asterisks with at most K operations (i.e. the answer we want).

$$f(S,K) = \max_{\substack{i \in valid(S,i) \\ j=[0,K)}} \left(f(A,j) + f(B, K-j-1) \right)$$

abacaacc*b*aabacbba

abacaa   aabacbba

Time complexity: $O(|S|^3 \cdot K^2)$

Definitely **TLE**

Magical String

... we need a muse and see the problem from a different perspective

Consider the **Weighted Interval Scheduling Problem**.

→ Given N intervals each with its weight, find a subset of intervals (at most of size K) s.t. there are no overlapping intervals and the total weight is maximized.

It's a similar problem!

```
abacaaccbaabacbba
aba
  acaa
    aac
     acc
        baa
         aaba
             cbb
              bba
```

Magical String

… we need a muse and see the problem from a different perspective

Consider the **Weighted Interval Scheduling Problem**.

→ Given N intervals each with its weight, find a subset of intervals (at most of size K) s.t. there are no overlapping intervals and the total weight is maximized.

It's a similar problem!

```
abacaaccbaabacbba
aba
  acaa
    aac
     acc
        baa
         aaba
             cbb
              bba
```

… but different

```
abacaa
aba
  acaa
```

Magical String

In Weighted Interval Scheduling Problem, we can only take one interval.

In Magical String, we can take “both” intervals.

Magical String

- Let SINGLE be the set of all intervals obtained individually from S.
- Let EXTEND be the set of all intervals obtained by extending SINGLE
  - [a, b] is in EXTEND iff its size is ≥ 3 and there is an interval [L, R] in SINGLE which can be **cut** into [a, b] by other intervals in SINGLE.
  - By definition, all intervals in SINGLE are in EXTEND.

→ The solution for Weighted Interval Scheduling Problem with EXTEND as the intervals is the solution for Magical String.

```
abacaa
aba         [1,3]
  acaa      [3,6]
   caa      [4,6]
```

→ [4, 6] is obtained by cutting [3, 6] with [1, 3].

Magical String

- Generate SINGLE $O(|S|)$
- Generate EXTEND $O(|S|^2)$

Size of EXTEND = $O(|S|)$

- Solve WISP with K: N intervals $O(NK)$

Magical String

- Generate SINGLE $O(|S|)$
- Generate EXTEND $O(|S|^2)$

Size of EXTEND = $O(|S|)$

- Solve WISP with K: N intervals $O(NK)$