DAPC 2023 Training Sessions
Session 4

Verwoerd

September 21, 2003

Session 4

- Role of the coach on big contests
- Tips, tricks and common mistakes
- Dealing with randomization
- Solutions to the Interactive Problems and Dynamic Programming Problems
- Solutions the hardest problems

Role of the coach

What is a coach?

- The coach is the contact person for the contest organization.
- Usually a faculty member, local contest organizer or student
- The coach doesn’t participate in the contest

Coach preparations before a contest

- Registers the teams for the contest
- Requests Extension of Eligibility if needed
- Requests funding for travel cost reimbursement
- Gives updates about important rules, systems and sometimes travel to the teams

Coach during the contest

- Makes sure teams are registered
- Visits during the test session
- Give last minute tips before the contest
- During the contest attend meetings
- Is available as emergency contact
- Evaluates with team members how the contest went

Tips, tricks and common mistakes

General tips

- Read the output specification carefully!
- Don’t forget to remove debug prints!
- When integers get large, use 64-bit!
- Do not do string concatenation with $+$ in a loop!
- Calling functions is more expensive than you might think!
- For Java, `BufferedReader` is faster than `Scanner`!
- Don’t forget to eat and drink. Programming contest is a sport, and you need to be energized and focussed for 5 hours.

Training your self

- If you don’t make the World Finals, you can train for next year’s event
- Many online problem-solving websites:
    - December: Advent of code (`https://adventofcode.com/`)
    - September-Januari: Universal Cup (`https://ucup.ac`)
    - Year round: Kattis Problem Archive (`https://open.kattis.com/`)
    - Year round: Codeforces (`https://codeforces.com/`)
- Several books available, listed on `https://chipcie.wisv.ch/resources`

Dealing with randomization

Randomization in Programming contest

- Randomized Algorithms

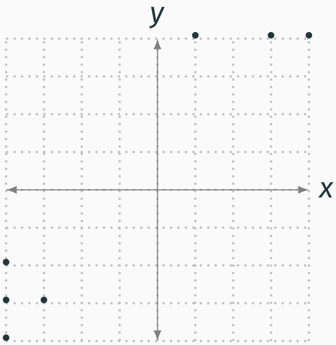
  Monte Carlo Algorithm The result might be incorrect (with low propability), with ranging time complexity.

  Las Vegas Algorithm The answer is always correct, but the time complexity may vary

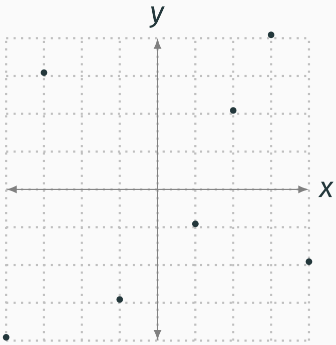
- Usually not used, but some very rare cases:
  - Prime Probability for large numbers (build in Java in `BigInteger.isProbablePrime()`)
  - Used in algorithms like Pollard Rho for integer factorization for large numbers over 10^{13}
- Randomized data

Randomized data

- Problems with random data have been appearing in the last years in the contest
- E.g.: all input independent uniformly random in a given range



Random none-uniform Distributed



Independent Uniformly Distributed

Properties of Uniform Random Points

- What is the average distance between two randomly chosen points inside a square with side length 1?

$$\frac{2+\sqrt{2}+5\ln\left(1+\sqrt{2}\right)}{15} \approx 0.5214$$

- This is referred to as the mean line segment length, several properties can be derived from this.
- This can be a subject to include in your Team Reference Document, but this might be too obscure.

Formulas for Uniform Random Points

Average distance between points on a line with length $d = \frac{1}{3}d$

Mininum distance beteen n **points on a line with length** $d = \frac{d}{n^2-1}$

Average distance between points of a quilateral triangle with side lenght a

$$\left(\frac{4+3\ln 3}{20}\right) \cdot a \approx 0.3647918 \cdot a$$

Average distance between points in a square with side lenght s

$$\left(\frac{2+\sqrt{2}+5\ln\left(1+\sqrt{2}\right)}{15}\right) \cdot s \approx 0.5214054 \cdot s$$

Average distance between points chosen on opposite sides

$$\left(\frac{2+\sqrt{2}+5\ln\left(1+\sqrt{2}\right)}{9}\right) \cdot s \approx 0.869009 \cdot s$$

Average chord length between two points on a circle with circumference r

$$\frac{4}{\pi}r \approx 1.2732395 \cdot r$$

Average distance between points in a cube with side length $s \approx 0.661707 \cdot s$

Lowest Latency

- Source BAPC 2022
- Time limit: 5s

Problem: Lowest Latency(1)

It is the year 2222. The whole universe has been explored, and settlements have been built on every single planet. You live in one of these settlements. While life is comfortable on almost all aspects, there is one dire problem: the latency on the internet connection with other planets is way too high.

Luckily, you have thought of a solution to solve this problem: you just need to put Bonded, Astronomically Paired Cables between all planets, and internet will be super fast! However, as you start developing this idea, you discover that constructing a cable between two planets is more difficult than expected. For this reason, you would like the first prototype of your cable to be between two planets which are as close as possible to each other.

Problem: Lowest Latency(2)

From your astronomy class, you know that the universe is best modelled as a large cube measuring 10^9 lightyears in each dimension. There are exactly 10^5 stationary planets, which are distributed completely randomly through the universe (more precisely: all the coordinates of the planets are independent uniformly random integers ranging from 0 to 10^9).

Given the random positions of the planets in the universe, your goal is to find the minimal Euclidean distance between any two planets.

Problem: Lowest Latency: Input and Output

Input

The input consists of:

- One line with an integer n, the number of planets.
- n lines, each with three integers x, y and z ($0 \leq x, y, z < 10^9$), the coordinates of one of the planets.

Your submissions will be run on exactly 100 test cases, all of which will have $n = 10^5$. The samples are smaller and for illustration only.

Each of your submissions will be run on new random test cases.

Output

Output the minimal Euclidean distance between any two of the planets.

Your answer should have an absolute or relative error of at most 10^{-6}.

Problem: Lowest Latency: Samples

Sample Input 1	Sample Output 1
5 10 5 1 8 2 0 4 7 5 1 0 9 0 10 7	3.7416573867739413

Sample Input 2	Sample Output 2
3 790726336 656087587 188785845 976472310 22830435 160538063 211966015 87530388 542618498	660540781.9387681

Problem: Lowest Latency: Observations

- The input size is 10^5, so we are looking for $\mathcal{O}(n \log^2 n)$ solution

---

[1] Or at least, almost always ;-)

Problem: Lowest Latency: Observations

- The input size is 10^5, so we are looking for $\mathcal{O}(n \log^2 n)$ solution
- The timelimit is high for high input IO

---

[1] Or at least, almost always ;-)

Problem: Lowest Latency: Observations

- The input size is 10^5, so we are looking for $\mathcal{O}(n \log^2 n)$ solution
- The timelimit is high for high input IO
- Since the input is Independent Uniform Random, the average line length will be $0.661707 \cdot 10^9 \approx 6.6 \cdot 10^8$ but the minimum will be lower.

---

[1] Or at least, almost always ;-)

Problem: Lowest Latency: Observations

- The input size is 10^5, so we are looking for $\mathcal{O}(n \log^2 n)$ solution
- The timelimit is high for high input IO
- Since the input is Independent Uniform Random, the average line length will be $0.661707 \cdot 10^9 \approx 6.6 \cdot 10^8$ but the minimum will be lower.
- Expand the minimum distance for n points on a line to three dimensions

$$\left(\frac{d}{n^2 - 1}\right)^3 => \left(\frac{10^9}{(10^5)^2 - 1}\right)^3 \approx 10^6$$

---

[1] Or at least, almost always ;-)

Problem: Lowest Latency: Observations

- The input size is 10^5, so we are looking for $\mathcal{O}(n \log^2 n)$ solution
- The timelimit is high for high input IO
- Since the input is Independent Uniform Random, the average line length will be $0.661707 \cdot 10^9 \approx 6.6 \cdot 10^8$ but the minimum will be lower.
- Expand the minimum distance for n points on a line to three dimensions

$$\left(\frac{d}{n^2 - 1}\right)^3 => \left(\frac{10^9}{(10^5)^2 - 1}\right)^3 \approx 10^6$$

- So the average length will be less then 10^6.[1]

---

[1] Or at least, almost always ;-)

Problem: Fastestest Function: Observations

- The Euclidean distance is calculated by

$$d(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2}$$

Problem: Fastestest Function: Observations

- The Euclidean distance is calculated by

$$d(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2}$$

- There are three common solutions to solve this problem

Problem: Fastestest Function: Observations

- The Euclidean distance is calculated by

$$d(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2}$$

- There are three common solutions to solve this problem
    1. Divide and conquer

Problem: Fastestest Function: Observations

- The Euclidean distance is calculated by

$$d(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2}$$

- There are three common solutions to solve this problem
    1. Divide and conquer
    2. Local brute force using the random property

Problem: Fastestest Function: Observations

- The Euclidean distance is calculated by

$$d(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2}$$

- There are three common solutions to solve this problem
  1. Divide and conquer
  2. Local brute force using the random property
  3. Sorted Bruteforce

Problem: Lowest Latency: Solution 1

- Sort the points by the x-value, use y and z for tiebreakers.
- Split the points in half and solve the halfs recursively
- Once only two points are in a group, calculate and return the distance
- Once both groups have their distances calculated, select the lowest distance
- Check with the points in the other groups within this distance if they create a shorter distance in the overlap and return the distance.
- The complexity of the $\mathcal{O}(n \log n)$

Problem: Lowest Latency: Solution 2

- Divide the space in $100 \times 100 \times 100$ spaces of size $10^7 \times 10^7 \times 10^7$
- Iterate over the pairs in each box
- The minimum distance can cross the space, so also include all pairs from adjacent boxes
- Time complexity is $\mathcal{O}\left(\frac{n^2}{k} + k\right)$, where k is the number of boxes

Problem: Lowest Latency: Solution 3

- Sort the points by the x-value, use y and z for tiebreakers
- The average x-distance is $\frac{10^9}{10^5} = 10^4$
- Points over 100 poistions apart are expected to have a distance over 10^6
- Consider all pairs (i, j) with $|i - j| \leq 100$
- This has a time complexity of $\mathcal{O}(100n + n \log n)$

Lowest Latency

```python
from math import sqrt

n = int(input())
ps = []
for _ in range(n):
  ps.append(list(map(int, input().split())))
ps.sort()
W = 100
ans = 3 * 10**9
for (i, (x1, y1, z1)) in enumerate(ps):
  for j in range(max(0, i - W), i):
    (x2, y2, z2) = ps[j]
    d = sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2 + (z1 - z2) ** 2)
    ans = min(ans, d)
print(ans)
```

Solutions to the Interactive Problems and Dynamic Programming Problems

Guessing Primes

- Source BAPC Preliminaries 2022
- Interactive Problem
- Time limit: 10s
- Guess the hidden 5-digit prime in at most 6 guesses, i.e., play Primel.

Guessing Primes

- There are 8363 primes between 10000 and 99999, can be generated within the time limit of 10s

Guessing Primes

- There are 8363 primes between 10000 and 99999, can be generated within the time limit of 10s
- Primality can be checked an odd number n cannot be divided by $(3..\lfloor\sqrt{n}\rfloor)$

Guessing Primes

- There are 8363 primes between 10000 and 99999, can be generated within the time limit of 10s
- Primality can be checked an odd number n cannot be divided by $(3..\lfloor\sqrt{n}\rfloor)$
- Selecting a random prime, and using the rules to generate the next guess will take on average seven guesses.

Guessing Primes

- There are 8363 primes between 10000 and 99999, can be generated within the time limit of 10s
- Primality can be checked an odd number n cannot be divided by $(3..\lfloor\sqrt{n}\rfloor)$
- Selecting a random prime, and using the rules to generate the next guess will take on average seven guesses.
- The reason is when only one digit is known, to many guesses needed for the other 4

Guessing Primes

- There are 8363 primes between 10000 and 99999, can be generated within the time limit of 10s
- Primality can be checked an odd number n cannot be divided by $(3..\lfloor\sqrt{n}\rfloor)$
- Selecting a random prime, and using the rules to generate the next guess will take on average seven guesses.
- The reason is when only one digit is known, to many guesses needed for the other 4
- So your first 2 guesses should all contain different digits, like 24683 and 10597

Guessing Primes

- There are 8363 primes between 10000 and 99999, can be generated within the time limit of 10s
- Primality can be checked an odd number n cannot be divided by $(3..\lfloor\sqrt{n}\rfloor)$
- Selecting a random prime, and using the rules to generate the next guess will take on average seven guesses.
- The reason is when only one digit is known, to many guesses needed for the other 4
- So your first 2 guesses should all contain different digits, like 24683 and 10597
- Use the digits to generate the next guesses

Guessing Primes

- There are 8363 primes between 10000 and 99999, can be generated within the time limit of 10s
- Primality can be checked an odd number n cannot be divided by $(3..\lfloor\sqrt{n}\rfloor)$
- Selecting a random prime, and using the rules to generate the next guess will take on average seven guesses.
- The reason is when only one digit is known, to many guesses needed for the other 4
- So your first 2 guesses should all contain different digits, like 24683 and 10597
- Use the digits to generate the next guesses
- This is guaranteed you can do this in 6 guesses

Guessing Primes(1)

```python
import random
from math import *

def is_prime(i):
  if i < 0: return False
  if i not in primes.keys(): primes[i] = not any(i % x == 0 for x in range(2, int(floor(sqrt(i))) + 1))
  return primes[i]

def is_valid(p, guess, res):
  gud = [c for c, r in zip(guess, res) if r != "w"]
  for i, (c, r) in enumerate(zip(guess, res)):
    if r == "w" and (c == p[i] if c in gud else c in p): return False
    if r == "y" and (c == p[i] or c not in p): return False
    if r == "g" and c != p[i]: return False
  return True
```

Guessing Primes(2)

```python
def perform_guess(guess_int):
  global left
  _, res = print(guess := str(guess_int)), input().strip()
  if res == "ggggg": return True
  left = [p for p in left if p != guess and is_valid(p, guess, res)]

n, primes = int(input()), {0: False, 1: False, 2: True, 3: True}
primes_list = [str(i) for i in range(100_000) if is_prime(i) and i > 10_000]
start_a, start_b = next((a, b) for a in primes_list for b in primes_list if sorted(f"{a}{b}") == list("0123456789"))

for _ in range(n):
  left = list(primes_list)
  if perform_guess(start_a) or perform_guess(start_b): continue
  while not perform_guess(random.choice(left)): pass
```

Dividing DNA

- Source BAPC 2022
- Interactive Problem
- Time limit: 2s
- Given a set of forbidden (present) intervals, partition $[0, n)$ into as many disjoint (absent) intervals as possible with at most $2n$ queries.

Dividing DNA

- If an interval is forbidden, then all shorter intervals are forbidden too.

Dividing DNA

- If an interval is forbidden, then all shorter intervals are forbidden too.
- An absent interval is just one longer then a forbidden interval.

Dividing DNA

- If an interval is forbidden, then all shorter intervals are forbidden too.
- An absent interval is just one longer then a forbidden interval.
- A greedy solution works here

Dividing DNA

- If an interval is forbidden, then all shorter intervals are forbidden too.
- An absent interval is just one longer then a forbidden interval.
- A greedy solution works here
- Start with $[0, 1)$ and keep growing until an absent interval is found

Dividing DNA

- If an interval is forbidden, then all shorter intervals are forbidden too.
- An absent interval is just one longer then a forbidden interval.
- A greedy solution works here
- Start with $[0, 1)$ and keep growing until an absent interval is found
- Then start at the last exclusive boundary a new boundary and continue until the end is reached.

Dividing DNA

- If an interval is forbidden, then all shorter intervals are forbidden too.
- An absent interval is just one longer then a forbidden interval.
- A greedy solution works here
- Start with $[0, 1)$ and keep growing until an absent interval is found
- Then start at the last exclusive boundary a new boundary and continue until the end is reached.
- 

Dividing DNA

- If an interval is forbidden, then all shorter intervals are forbidden too.
- An absent interval is just one longer then a forbidden interval.
- A greedy solution works here
- Start with $[0, 1)$ and keep growing until an absent interval is found
- Then start at the last exclusive boundary a new boundary and continue until the end is reached.
- 

- The result is the number of intervals with *n* queries

Dividing DNA

```
ans, i = 0, 0
for j in range(1, int(input()) + 1):
  print("?", i, j)
  if input() == "absent":
    ans += 1
    i = j
print("!", ans)
```

Jaged skylines

- Source BAPC 2022
- Interactive Problem
- Time limit: 4s
- Given $w \leq 10000$ integers $0 \leq h_i \leq 10^{18}$, find the maximum in at most 12000 queries: “Is integer h_i less than y?”

Jaged skylines

- For every column we can binary search the highest tile

Jaged skylines

- For every column we can binary search the highest tile
- This is 5000 queries, so too much

Jaged skylines

- For every column we can binary search the highest tile
- This is 5000 queries, so too much
- Rather than start in the middle, we can check if it is higher than the best found

Jaged skylines

- For every column we can binary search the highest tile
- This is 5000 queries, so too much
- Rather than start in the middle, we can check if it is higher than the best found
- and then binary search only found

Jaged skylines

- For every column we can binary search the highest tile
- This is 5000 queries, so too much
- Rather than start in the middle, we can check if it is higher than the best found
- and then binary search only found
- Worst case: the maximum increases with every column

Jaged skylines

- For every column we can binary search the highest tile
- This is 5000 queries, so too much
- Rather than start in the middle, we can check if it is higher than the best found
- and then binary search only found
- Worst case: the maximum increases with every column
- Randomize the order, the change that an item is higher is $\ln(w)$

Jaged skylines

- For every column we can binary search the highest tile
- This is 5000 queries, so too much
- Rather than start in the middle, we can check if it is higher than the best found
- and then binary search only found
- Worst case: the maximum increases with every column
- Randomize the order, the change that an item is higher is $\ln(w)$
- Resulting in number of queries of $w + \ln(w) \cdot \log(h)$

Jaged skylines

- For every column we can binary search the highest tile
- This is 5000 queries, so too much
- Rather than start in the middle, we can check if it is higher than the best found
- and then binary search only found
- Worst case: the maximum increases with every column
- Randomize the order, the change that an item is higher is $\ln(w)$
- Resulting in number of queries of $w + \ln(w) \cdot \log(h)$
- Note that this a Monte Carlo estimate, which was manually monitored by the jury

Dividing DNA

```python
import random

w, h = map(int, input().split())
xs, highest_x, highest_y = list(range(1, w + 1)), 1, 1
random.shuffle(xs)
for x in xs:
  print(f"? {x} {min(highest_y, h)}")
  if input() == "building":
    low, high = highest_y + 1, h + 1
    while low < high:
      mid = (low + high) // 2
      print(f"? {x} {mid}")
      if input() == "building": low = mid + 1
      else: high = mid
    highest_x, highest_y = x, high
print(f"! {highest_x} {highest_y - 1}")
```

Solving the Hardest Problems

Heavy Hauling

- Source BAPC Preliminaries 2022
- Time limit: 3s
- Given n boxes at given positions. Moving a box d positions costs d^2. What is the minimal cost to make all box positions distinct?

Heavy Hauling (1)

- We have $n \leq 10^6$ so we are looking for a $\mathcal{O}(n \log n)$ algorithm.
- The boxes will remain in their original order (they will never overtake each other).
- Groups of consecutive boxes map to an interval.
- The cost of moving a box from p to x can be modelled as $C_p(x) = (x - p)^2$. For example, moving box 3 to position x gives $C_3(x) = (x - 3)^2 = x^2 - 6x + 9$
- When two boxes overlap from the left group to the right group. For example, with 2 boxes, the left most box is at x: $C_{3,3} = C_3(x) + C_3(x + 1) = (x - 3)^2 + (x - 2)^2 = 2x^2 - 10x + 13$.

Heavy Hauling (2)

- When merging groups, they can touch or overlap with existing group, so merge them recursivly
- Now every group has a cost function $C(x) = ax^2 + bx + c$
- The minimal cost is $C\left(\lfloor \frac{-b}{2a} + \frac{1}{2} \rfloor\right)$
- The total runtime is $\mathcal{O}(n)$ for the $n-1$ merges

Heavy Hauling

```python
class S:
  def __init__(self, a, b, c): self.a, self.b, self.c = a, b, c

  def getStart(self): return (self.a - self.b) // (2 * self.a)

  def getScore(self):
    x = self.getStart()
    return self.a * x * x + self.b * x + self.c

  def intersect(self, s): return self.getStart() + self.a >= s.getStart()

  def merge(self, s):
    return S(self.a + s.a, self.b + 2 * self.a * s.a + s.b, self.c + self.a * self.a * s.a + self.a * s.b + s.c)

n, A, B = int(input()), [S(1, -2 * a, a * a) for a in map(int, input().split())], []
for a in A:
  while B and B[-1].intersect(a): a = B.pop().merge(a)
  B.append(a)
print(sum(s.getScore() for s in B))
```

Inked Inscriptions

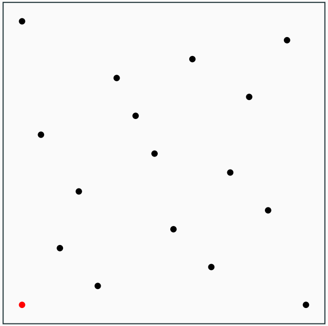
- Source BAPC Preliminaries 2022
- Time limit: 4s
- Copy n psalms in at most $2n\sqrt{n}$ pageflips

Inked Inscriptions (1)

- The order of the target algorith is given as $\mathcal{O}\left(n\sqrt{n}\right)$

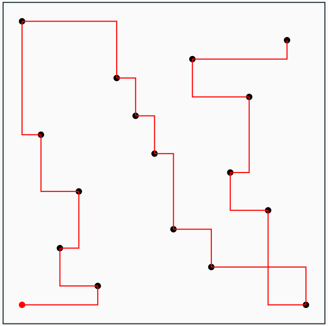
Inked Inscriptions (1)

- The order of the target algorith is given as $\mathcal{O}\left(n\sqrt{n}\right)$
- Each psalm can be represented in a plot as current page and the target page



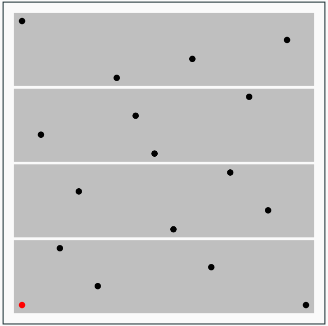
Inked Inscriptions (1)

- The order of the target algorith is given as $\mathcal{O}\left(n\sqrt{n}\right)$
- Each psalm can be represented in a plot as current page and the target page
- Create a path with the manhattan distance of max length $2n\sqrt{n}$



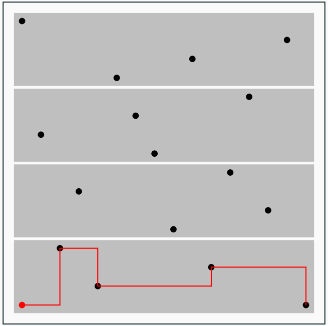
Inked Inscriptions (1)

- Divide the graph in $\sqrt{n}$ bands of height $\sqrt{n}$



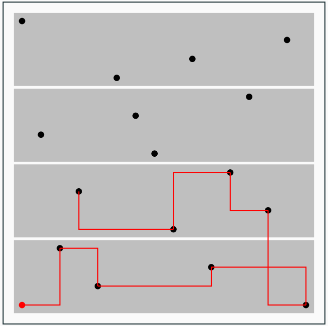
Inked Inscriptions (1)

- Divide the graph in $\sqrt{n}$ bands of height $\sqrt{n}$
- Move each band alternating from left to right and then right to left.



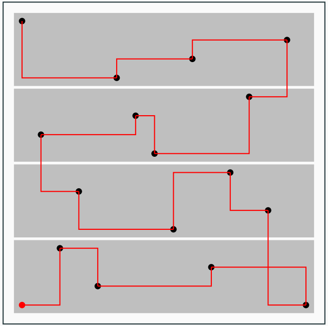
Inked Inscriptions (1)

- Divide the graph in $\sqrt{n}$ bands of height $\sqrt{n}$
- Move each band alternating from left to right and then right to left.



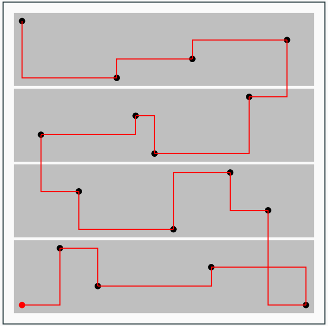
Inked Inscriptions (1)

- Divide the graph in $\sqrt{n}$ bands of height $\sqrt{n}$
- Move each band alternating from left to right and then right to left.
- This results in $1.5n\sqrt{n} + 2n$ page flips



Inked Inscriptions (1)

- Divide the graph in $\sqrt{n}$ bands of height $\sqrt{n}$
- Move each band alternating from left to right and then right to left.
- This results in $1.5n\sqrt{n} + 2n$ page flips
- Alternative: Go greedy to the nearest unvisited point
- Alternative: Visit the points in a spiral



Inked Inscriptions

```python
n = int(input())
points = [(int(c),i+1) for i,c in enumerate(input().split())]

x, y = 1,1
result = []

sqrtn = round(n**.5)
for i in range(sqrtn):
  row = points[(i*n)//sqrtn:((i+1)*n)//sqrtn]
  row.sort(reverse = i % 2)
  for j,i in row:
    result.append("%i %i" % (i,j))
    x,y = i,j

print(*result, sep='\n')
```

Adjusted Average

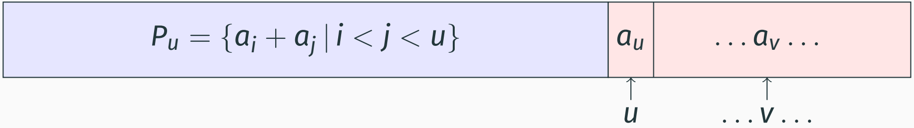
- Source BAPC 2022
- Time limit: 8s
- Given $n \leq 1500$ integers a_i, remove at most $k \leq 4$ of them to get an average as close as possible to the target x.

Adjusted Average

- Observation: The high time limit is for the IO
- Observation: The input $n \leq 1500$, so we are looking for a $\mathcal{O}(n^2 \log^2 n)$
- Calculate for each set of k-elements the average that is as close as possible to $S_k \sum_i a_i - k \cdot x$
- For cases where you remove 1 or 2 elements is doable to brute force in $\mathcal{O}\left(\frac{n^k}{k!}\right)$
- For cases where you remove 3 or 4 element this is too slow, so we use meet-in-the-middle approach

Adjusted Average: $k = 3$ and $k = 4$

- For the case $k = 3$:
  - For each $u \in [1, n]$ calculate the possible values $P_u = a_i + a_j$ where $i < j < u$
  - For each value take the closest value from P_u closest to $S_k - a_u$
  - By using an ordered set for P_u values, the time complexity is $\mathcal{O}(n^2 \log n)$
- For the case $k = 4$:
  - Reuse the values P_u, but now also consider v where $v \in (u, n]$
  - For each u, loop over v and pick P_u closes to $S_k - a_u - a_v$
  - This is still $\mathcal{O}(n^2 \log n)$



Adjusted Average

```python
from bisect import bisect

n, K, X = map(int, input().split())

xs = sorted(list(map(int, input().split())))
S = sum(xs)

pairs = []
for i in range(n):
  for j in range(i):
    pairs.append((xs[i] + xs[j], [i, j]))
pairs.sort()

# k = 0
best = abs(S / n - X)
if K >= 1:
  for s in xs:
    best = min(best, abs((S - s) / (n - 1) - X))
if K >= 2:
  for (s, ij) in pairs:
    best = min(best, abs((S - s) / (n - 2) - X))

```

Adjusted Average

```
if K >= 3:
  i = 0
  j = len(pairs) - 1
  while True:
    s1 = xs[i]
    (s2, [k, l]) = pairs[j]
    if k != i and l != i:
      best = min(best, abs((S - s1 - s2) / (n - 3) - X))
    if i == len(xs) - 1 and j == 0:
      break
    if j == 0 or (i < len(xs) - 1 and S - s1 - s2 > X * (n - 3)):
      i = i + 1
    else:
      j = j - 1
```

Adjusted Average

```python
if K >= 4:
  for (s1, [i, j]) in pairs:
    s2 = S - s1 - (n - 4) * X
    idx = bisect(pairs, (s2, []))
    # Find first position to the left and right disjoint with ij
    for idx2 in range(idx - 1, -1, -1):
      s2, kl = pairs[idx2]
      if i not in kl and j not in kl:
        best = min(best, abs((S - s1 - s2) / (n - 4) - X))
        break
    for idx2 in range(idx, len(pairs)):
      s2, kl = pairs[idx2]
      if i not in kl and j not in kl:
        best = min(best, abs((S - s1 - s2) / (n - 4) - X))
        break

print(best)
```

Grinding Gravel

- Source BAPC 2022
- Time limit: 4s
- Given $n \leq 100$ integers, split them into groups of size $k \leq 8$ making as few cuts as possible.

Grinding Gravel

- First for every number $x \geq k$ is replaced by $x mod k$.
- Now all integers are in $[0, k)$
- For every $x < \frac{k}{2}$, we can pair up x and $k - x$, where $x = 0$ is its own group
- This leaves 4 different values left: 1 or 7, 2 or 6, 3 or 5 and at most one 4
- Now do a DP on state $[c_1, \ldots, c_{k-1}]$, the counts for each remainder
    - For each subset with sum 0 mod k and recurse
    - merge the least-occuring element with one of the others

Grinding Gravel

```
n, k = map(int, input().split())
w = list(map(int, input().split()))
ans = 0
cnt = [0] * k
# modulo
for x in w:
  if x % k == 0:
    ans += 1
  else:
    cnt[x % k] += 1
# pairs
for i in range(1, k // 2):
  x = min(cnt[i], cnt[k - i])
  cnt[i] -= x
  cnt[k - i] -= x
  ans += x
if k % 2 == 0:
  x = cnt[k // 2] // 2
  cnt[k // 2] -= 2 * x
  ans += x
# Left with at most 3 non-empty values, and possibly k/2.
ans = {tuple([0] * k): ans}
print(sum(w) // k - calc(cnt))
```

Grinding Gravel

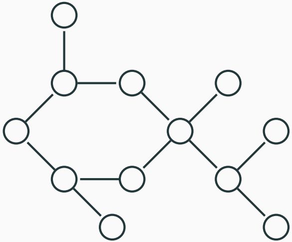
```python
def calc(cnts):
  if tuple(cnts) in ans:
    return ans[tuple(cnts)]
  best = (100, -1)
  for m in range(1, k):
    if cnts[m] > 0:
      best = min(best, (cnts[m], m))
  m = best[1]
  new_cnts = cnts[:]
  new_cnts[m] -= 1
  best = 0
  for i in range(1, k):
    if new_cnts[i] > 0:
      new_cnts[i] -= 1
      v = 0
      if m + i != k:
        new_cnts[(m + i) % k] += 1
      else:
        v += 1
      v += calc(new_cnts)
      if m + i != k:
        new_cnts[(m + i) % k] -= 1
      best = max(best, v)
      new_cnts[i] += 1
  ans[tuple(cnts)] = best
  return best
```

House Numbering

- Source BAPC 2022
- Time limit: 4s
- Given a graph with n nodes and edges, and h house numbers for an edge, determine whether house numbers can be assigned such that there is no intersection where two edges start with the same house number.

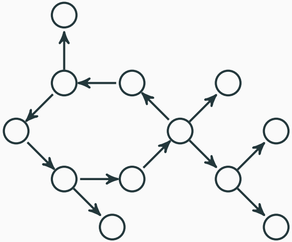
House Numbering

- Every node in the grap can at most have one edge with house number 1
- The number of nodes is equal to the number of edges, the grapsh contains exactly 1 cycle.
- The cycle the numbering of 1 has to be clockwise or counter clock wise
- The nodes in the cycle have threes attached in which the number 1 has to face outward



House Numbering

- Find the cycle in the graph
- Assign house numbers clockwise and check if it is valid, if so report it
- Assign house numbers counter-clockwise and check if it valid, if so report it
- print impossible



Grinding Gravel

```python
from collections import defaultdict, Counter as C

n, edges, stack, seen, todo = int(input()), defaultdict(dict), [], set(), [[1]]
for i in range(n):
  u, v, h = map(int, input().split())
  edges[u][v] = (h, i)
  edges[v][u] = (h, i)

if any(any(c > 2 for c in C(h for h, _ in edges[u].values()).values()) for u in edges): print("impossible"), exit()

while True:
  if (curr := todo[-1].pop()) in seen: break
  stack.append(curr), seen.add(curr)
  todo.append([x for x in edges[curr].keys() if len(stack) == 1 or stack[-2] != x])
  while todo and not todo[-1]:
    if not stack: print("impossible"), exit()
    todo.pop(), seen.remove(stack.pop())

cycle = [curr]
while stack:
  u = stack.pop()
  if u == curr: break
  cycle.append(u)
while stack: seen.remove(stack.pop())
```

Grinding Gravel

```python
def find_numbering(st):
  new_seen = set(seen)
  ans = []
  for u, v in zip(st, [*st[1:], st[0]]):
    h, i = edges[u][v]
    ans.append((i, u))
    todo = [(v, h)]
    while todo:
      curr, h2 = todo.pop()
      new_seen.add(curr), (ns := [(neigh, t) for neigh, t in edges[curr].items() if neigh not in new_seen])
      if any(c > 1 for c in C([*((h3 for _, (h3, _) in ns)), *([h] if curr == v else [])]).values()): return False
      for neigh, (h3, i3) in ns: ans.append((i3, neigh)), todo.append((neigh, h3))
  return " ".join(str(u) for i, u in sorted(ans))


print(find_numbering(cycle) or find_numbering(cycle[::-1]) or "impossible")
```

Conclussion

Thanks for comming

Contest on Saturday, Good luck all!

Any Questions?