NCPC 2023
Presentation of solutions

2023-10-07

Problems prepared by

- Arnar Bjarni Arnarson (Reykjavik University)
- Andreas Björklund (IT University of Copenhagen)
- Atli Fannar Franklín (University of Iceland)
- Nils Gustafsson (KTH Royal Institute of Technology)
- Oskar Haarklou Veileborg (Aarhus University)
- Asger Hautop Drewsen (Aarhus University)

Test solver

- Björn Martinsson

NCPC 2024

Presentation of solutions

October 5, 2024

Problems prepared by

- Hlíf Arnbjargardóttir (Atli's Mom)
- Arnar Bjarni Arnarson (Reykjavik University)
- Pål Grønås Drange (University of Bergen)
- Atli Fannar Franklín (University of Iceland)
- Nils Gustafsson (KTH Royal Institute of Technology)
- Björn Martinsson (KTH Royal Institute of Technology)
- Bergur Snorrason (University of Iceland)

A — Aperiodic Appointments

Problem

A string is generated by repeatedly appending a 1 if the last characters contain a pattern that repeats K times, and a 0 otherwise. Count the number of ones.

A — Aperiodic Appointments

Problem

A string is generated by repeatedly appending a 1 if the last characters contain a pattern that repeats K times, and a 0 otherwise. Count the number of ones.

Solution

1. A good idea is to write out the string for $K = 3$ or 4 on paper, and look for patterns.

A — Aperiodic Appointments

000100010001100010001000110001000100011111111111

A — Aperiodic Appointments

000100010001100010001000110001000100011111111111

A — Aperiodic Appointments

Solution

1. Guess: the string is generated by first letting $s = 0$, and then applying $s = s * k + 1$ k times, and then append infinitely many ones.

Statistics at 4-hour mark: 379 submissions, 68 accepted, first after HH:MM

A — Aperiodic Appointments

Solution

1. Guess: the string is generated by first letting $s = 0$, and then applying $s = s * k + 1$ k times, and then append infinitely many ones.
2. There are now some recursive ways of counting the number of ones in $\mathcal{O}(\log(N))$.

Statistics at 4-hour mark: 379 submissions, 68 accepted, first after HH:MM

B — Baseball Court

Problem

Given bounds a, b how many partitions ρ of n satisfy $|\rho| \leq b$, $\max \rho \leq a$ and $\rho_i + i$ is constant across all indices i such that $\rho_i \neq \rho_{i+1}$.

Solution

Problem

Given bounds a, b how many partitions ρ of n satisfy $|\rho| \leq b$, $\max \rho \leq a$ and $\rho_i + i$ is constant across all indices i such that $\rho_i \neq \rho_{i+1}$.

Solution

1. First consider the case without the bounds a, b.

B — Baseball Court

Problem

Given bounds a, b how many partitions ρ of n satisfy $|\rho| \leq b$, $\max \rho \leq a$ and $\rho_i + i$ is constant across all indices i such that $\rho_i \neq \rho_{i+1}$.

Solution

1. First consider the case without the bounds a, b.
2. We define $f(n, d)$ as the number of partitions of n with $\rho_i + i = d + 1$ for the aforementioned indices.

B — Baseball Court

Problem

Given bounds a, b how many partitions ρ of n satisfy $|\rho| \leq b$, $\max \rho \leq a$ and $\rho_i + i$ is constant across all indices i such that $\rho_i \neq \rho_{i+1}$.

Solution

1. First consider the case without the bounds a, b.
2. We define $f(n, d)$ as the number of partitions of n with $\rho_i + i = d + 1$ for the aforementioned indices.
3. The problem can now be solved using dynamic programming.

B — Baseball Court

Solution

1. Consider the maximum index k such that $\rho_1 = \rho_k$ and $\rho_k \neq \rho_{k+1}$. Then $\rho_k + k = d + 1$. Then by cutting off these front values we remove k values and a sum of $k(d + 1 - k)$. This gives us the recurrence

$$f(n, d) = \sum_{k=1}^{d} f(n - k(d + 1 - k), d - k)$$

Solution

1. Consider the maximum index k such that $\rho_1 = \rho_k$ and $\rho_k \neq \rho_{k+1}$. Then $\rho_k + k = d + 1$. Then by cutting off these front values we remove k values and a sum of $k(d+1-k)$. This gives us the recurrence

$$f(n, d) = \sum_{k=1}^{d} f(n - k(d+1-k), d-k)$$

2. This is $\mathcal{O}(n^3)$ which is not quite good enough. But $f(n - k(d+1-k), d-k)$ has the first argument < 0 for all but the first and last $\sqrt{n}$ terms. So we can skip those and get a time complexity of $\mathcal{O}(n^{2.5})$.

B — Baseball Court

Solution

1. Consider the maximum index k such that $\rho_1 = \rho_k$ and $\rho_k \neq \rho_{k+1}$. Then $\rho_k + k = d + 1$. Then by cutting off these front values we remove k values and a sum of $k(d+1-k)$. This gives us the recurrence

$$f(n, d) = \sum_{k=1}^{d} f(n - k(d+1-k), d-k)$$

2. This is $\mathcal{O}(n^3)$ which is not quite good enough. But $f(n - k(d+1-k), d-k)$ has the first argument < 0 for all but the first and last $\sqrt{n}$ terms. So we can skip those and get a time complexity of $\mathcal{O}(n^{2.5})$.
3. Then we simply subtract all partitions that have $|\rho| > a$ or $\max \rho > b$, then add back all the ones that violate both to get the right answer by inclusion-exclusion.

B — Baseball Court

Solution

1. Consider the maximum index k such that $\rho_1 = \rho_k$ and $\rho_k \neq \rho_{k+1}$. Then $\rho_k + k = d + 1$. Then by cutting off these front values we remove k values and a sum of $k(d+1-k)$. This gives us the recurrence

$$f(n,d) = \sum_{k=1}^{d} f(n - k(d+1-k), d-k)$$

2. This is $\mathcal{O}(n^3)$ which is not quite good enough. But $f(n - k(d+1-k), d-k)$ has the first argument < 0 for all but the first and last $\sqrt{n}$ terms. So we can skip those and get a time complexity of $\mathcal{O}(n^{2.5})$.
3. Then we simply subtract all partitions that have $|\rho| > a$ or $\max \rho > b$, then add back all the ones that violate both to get the right answer by inclusion-exclusion.

Statistics at 4-hour mark: 14 submissions, 0 accepted, first after ??:??

C — Converting Romans

Problem

Convert Roman numerals to regular Arabic numbers.

Solution

Statistics at 4-hour mark: 473 submissions, 154 accepted, first after HH:MM

C — Converting Romans

Problem

Convert Roman numerals to regular Arabic numbers.

Solution

1. Using a map/dictonary to map Roman digits to numbers is convenient.

Statistics at 4-hour mark: 473 submissions, 154 accepted, first after HH:MM

C — Converting Romans

Problem

Convert Roman numerals to regular Arabic numbers.

Solution

1. Using a map/dictonary to map Roman digits to numbers is convenient.
2. For each digit in a number, check if it is smaller than anything to the right. In that case, subtract it.

Statistics at 4-hour mark: 473 submissions, 154 accepted, first after HH:MM

C — Converting Romans

Problem

Convert Roman numerals to regular Arabic numbers.

Solution

1. Using a map/dictonary to map Roman digits to numbers is convenient.
2. For each digit in a number, check if it is smaller than anything to the right. In that case, subtract it.
3. Make sure this runs faster than $\mathcal{O}(L^2)$!

Statistics at 4-hour mark: 473 submissions, 154 accepted, first after HH:MM

C — Converting Romans

Problem

Convert Roman numerals to regular Arabic numbers.

Solution

1. Using a map/dictonary to map Roman digits to numbers is convenient.
2. For each digit in a number, check if it is smaller than anything to the right. In that case, subtract it.
3. Make sure this runs faster than $\mathcal{O}(L^2)$!
4. This can be done by looping from the end of the string, and keeping track of the largest digit seen so far.

Statistics at 4-hour mark: 473 submissions, 154 accepted, first after HH:MM

D — Double Deck

Problem

Solve the longest common subsequence problem when the number of occurrences of any specific value is ≤ 15.

Solution

D — Double Deck

Problem

Solve the longest common subsequence problem when the number of occurrences of any specific value is ≤ 15.

Solution

1. We consider the classic dynamic programming solution and the space optimization of that solution.

D — Double Deck

Problem

Solve the longest common subsequence problem when the number of occurrences of any specific value is ≤ 15.

Solution

1. We consider the classic dynamic programming solution and the space optimization of that solution.
2. Next we place this single row of the dynamic programming memoization table into a data structure like a segment tree or fenwick tree that can query maximum values and update points.

D — Double Deck

Problem

Solve the longest common subsequence problem when the number of occurrences of any specific value is ≤ 15.

Solution

1. We consider the classic dynamic programming solution and the space optimization of that solution.
2. Next we place this single row of the dynamic programming memoization table into a data structure like a segment tree or fenwick tree that can query maximum values and update points.
3. Now the number of updates we have to make in the tree is bounded due to the few occurrences of a given value.

D — Double Deck

Problem

Solve the longest common subsequence problem when the number of occurrences of any specific value is ≤ 15.

Solution

1. We consider the classic dynamic programming solution and the space optimization of that solution.
2. Next we place this single row of the dynamic programming memoization table into a data structure like a segment tree or fenwick tree that can query maximum values and update points.
3. Now the number of updates we have to make in the tree is bounded due to the few occurrences of a given value.

Statistics at 4-hour mark: 214 submissions, 23 accepted, first after 00:10

E — Electronic Components

Problem

There are N types of components, each with f_i copies and placement time t_i. In one move you can place two components of different types i and j in $\max(t_i, t_j)$ nanoseconds. Find the minimum total time to place all components.

Solution

E — Electronic Components

Problem

There are N types of components, each with f_i copies and placement time t_i. In one move you can place two components of different types i and j in $\max(t_i, t_j)$ nanoseconds. Find the minimum total time to place all components.

Solution

1. Greedily place two component types with maximum t_i? Doesn't work on second sample.
2. If t_i values are similar, then it is more important to match all components.

E — Electronic Components

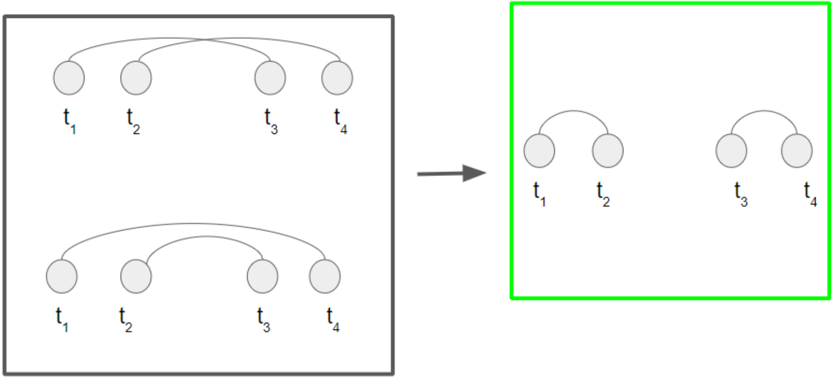
Solution

1. Let's try DP instead. Sort the types with respect to time.

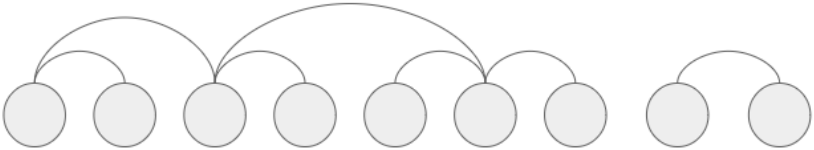
Solution

1. Let's try DP instead. Sort the types with respect to time.
2. $DP(i, u) =$ answer of first i components, if there are u unmatched components that we already "paid" for.
3. Number of states is N^2F, transitions take $\mathcal{O}(F)$. Running time $\mathcal{O}(N^2F^2)$.

E — Electronic Components



E — Electronic Components



E — Electronic Components

Solution

1. $DP(i, u) =$ answer of first i components, if there are u unmatched components that we already "paid" for.
2. Number of states is N^2F, transitions take $\mathcal{O}(F)$. Running time $\mathcal{O}(N^2F^2)$.

Statistics at 4-hour mark: 45 submissions, 2 accepted, first after HH:MM

E — Electronic Components

Solution

1. $DP(i, u) =$ answer of first i components, if there are u unmatched components that we already "paid" for.
2. Number of states is N^2F, transitions take $\mathcal{O}(F)$. Running time $\mathcal{O}(N^2F^2)$.
3. The value of u never exceeds F in an optimal solution.

Statistics at 4-hour mark: 45 submissions, 2 accepted, first after HH:MM

E — Electronic Components

Solution

1. $DP(i, u) =$ answer of first i components, if there are u unmatched components that we already "paid" for.
2. Number of states is N^2F, transitions take $\mathcal{O}(F)$. Running time $\mathcal{O}(N^2F^2)$.
3. The value of u never exceeds F in an optimal solution.
4. Write out the DP transitions carefully. They can be reduced to RMQ:s on a sliding window.

Statistics at 4-hour mark: 45 submissions, 2 accepted, first after HH:MM

E — Electronic Components

Solution

1. $DP(i, u) =$ answer of first i components, if there are u unmatched components that we already "paid" for.
2. Number of states is N^2F, transitions take $\mathcal{O}(F)$. Running time $\mathcal{O}(N^2F^2)$.
3. The value of u never exceeds F in an optimal solution.
4. Write out the DP transitions carefully. They can be reduced to RMQ:s on a sliding window.
5. Use segment tree, RMQ, or priority queue in the transition: $\mathcal{O}(NF\log(F))$.
6. Bonus: can you solve it in $\mathcal{O}(NF)$ (without fancy RMQ)?

Statistics at 4-hour mark: 45 submissions, 2 accepted, first after HH:MM

F — Fence Fee

Problem

Compute the faces of the provided planar graph, and find the areas (squared) of all faces (except the outer face).

Solution

F — Fence Fee

Problem

Compute the faces of the provided planar graph, and find the areas (squared) of all faces (except the outer face).

Solution

1. For each vertex, sort its neighborhood counter-clockwise (or cw).

F — Fence Fee

Problem

Compute the faces of the provided planar graph, and find the areas (squared) of all faces (except the outer face).

Solution

1. For each vertex, sort its neighborhood counter-clockwise (or cw).
2. For each edge uv, find the left-hand face of uv by going from u to v, and then go to the "next" neighbor (after u) of v, and repeat until you return to u.

F — Fence Fee

Problem

Compute the faces of the provided planar graph, and find the areas (squared) of all faces (except the outer face).

Solution

1. For each vertex, sort its neighborhood counter-clockwise (or cw).
2. For each edge uv, find the left-hand face of uv by going from u to v, and then go to the "next" neighbor (after u) of v, and repeat until you return to u.
3. Repeat the process for vu to find the right-hand face.

F — Fence Fee

Problem

Compute the faces of the provided planar graph, and find the areas (squared) of all faces (except the outer face).

Solution

1. For each vertex, sort its neighborhood counter-clockwise (or cw).
2. For each edge uv, find the left-hand face of uv by going from u to v, and then go to the "next" neighbor (after u) of v, and repeat until you return to u.
3. Repeat the process for vu to find the right-hand face.
4. Detect the outer face (the leftmost vertex (break even on y coordinate) and its neighbor with angle highest $\leq 90°$)

F — Fence Fee

Problem

Compute the faces of the provided planar graph, and find the areas (squared) of all faces (except the outer face).

Solution

1. For each vertex, sort its neighborhood counter-clockwise (or cw).
2. For each edge uv, find the left-hand face of uv by going from u to v, and then go to the "next" neighbor (after u) of v, and repeat until you return to u.
3. Repeat the process for vu to find the right-hand face.
4. Detect the outer face (the leftmost vertex (break even on y coordinate) and its neighbor with angle highest $\leq 90^\circ$)
5. Use the shoelace algorithm for computing the area of each face/polygon.

Statistics at 4-hour mark: 61 submissions, 16 accepted, first after 00:50

H — Heroes of Velmar

Problem

Given the description of a card game and a final state of the game, implement the rules for resolving which player won the match.

Solution

1. Process each player and location pair individually to apply abilities and add all the power levels of the cards together.
    1. If the card is Seraphina and k is the number of friendly cards at the location, add $k-1$ power to the location.
    2. If the card is Thunderheart and the location has 4 friendly cards, add 6 power to the location.
    3. If the card is Zenith, and the location is the center location, add 5 power to the location.
2. Count the number of locations where each player wins.
3. If one of the players wins more locations than the other, that player wins.

H — Hotfix

Problem

Given a program that prints every unique substring of a string along with its number of occurrences, find out how many copies of each non-whitespace character that program would print for a given string.

Solution

H — Hotfix

Problem

Given a program that prints every unique substring of a string along with its number of occurrences, find out how many copies of each non-whitespace character that program would print for a given string.

Solution

1. To solve this we need suffix arrays and segment trees. We start by constructing the suffix array and its longest common prefix array.

H — Hotfix

Problem

Given a program that prints every unique substring of a string along with its number of occurrences, find out how many copies of each non-whitespace character that program would print for a given string.

Solution

1. To solve this we need suffix arrays and segment trees. We start by constructing the suffix array and its longest common prefix array.
2. We now solve the characters and numbers separately.

H — Hotfix

Problem

Given a program that prints every unique substring of a string along with its number of occurrences, find out how many copies of each non-whitespace character that program would print for a given string.

Solution

1. To solve this we need suffix arrays and segment trees. We start by constructing the suffix array and its longest common prefix array.
2. We now solve the characters and numbers separately.
3. For the letters we use a lazy propagation segment tree that allows for a range update where the first element is incremented by b, the next by $a + b$, the third by $2a + b$ and so on.

Solution

1. The segment tree will keep track of the number of occurrences of each character in the input string as we iterate over the suffix array. Let S be the i-th element of the suffix array and L be the $(i-1)$-st element of the longest common prefix array.

Solution

1. The segment tree will keep track of the number of occurrences of each character in the input string as we iterate over the suffix array. Let S be the i-th element of the suffix array and L be the $(i-1)$-st element of the longest common prefix array.
2. Then $n - S - L$ new prefixes start at position S. The first L possible substrings starting at S are duplicates. So we increment the segment tree by $n - S - L$ at positions S through $S + L - 1$. Then we increment the segment tree by $n - S - L$, $n - S - L - 1$, $n - S - L - 2$, and so on at positions $S + L$ through $n - 1$. At the end of this all we can read off the number of occurrences of each character from the segment tree.

Solution

1. Finally we consider the digits. For this we use a lazy propagation segment tree that allows incrementing values on a range and then a collect operation that both counts the digits in the values on a range before then zeroing them out.

Solution

1. Finally we consider the digits. For this we use a lazy propagation segment tree that allows incrementing values on a range and then a collect operation that both counts the digits in the values on a range before then zeroing them out.
2. The collect operation might be slow worst case, but is amortized fast. We let the values of the segment tree count the number of occurrences of the substrings starting at our current position in the string as we iterate through it.

Solution

1. Finally we consider the digits. For this we use a lazy propagation segment tree that allows incrementing values on a range and then a collect operation that both counts the digits in the values on a range before then zeroing them out.
2. The collect operation might be slow worst case, but is amortized fast. We let the values of the segment tree count the number of occurrences of the substrings starting at our current position in the string as we iterate through it.
3. As we move from position $i - 1$ to i, let S and L be as before.

Solution

1. Finally we consider the digits. For this we use a lazy propagation segment tree that allows incrementing values on a range and then a collect operation that both counts the digits in the values on a range before then zeroing them out.
2. The collect operation might be slow worst case, but is amortized fast. We let the values of the segment tree count the number of occurrences of the substrings starting at our current position in the string as we iterate through it.
3. As we move from position $i - 1$ to i, let S and L be as before.
4. Everything but the first L substrings are no longer valid, so we count those and delete them using the collect operation on indices L to $n - 1$.

Solution

1. Finally we consider the digits. For this we use a lazy propagation segment tree that allows incrementing values on a range and then a collect operation that both counts the digits in the values on a range before then zeroing them out.
2. The collect operation might be slow worst case, but is amortized fast. We let the values of the segment tree count the number of occurrences of the substrings starting at our current position in the string as we iterate through it.
3. As we move from position $i - 1$ to i, let S and L be as before.
4. Everything but the first L substrings are no longer valid, so we count those and delete them using the collect operation on indices L to $n - 1$.
5. Next we add the substrings found at our current position, incrementing the values at indices 0 through $n - S - 1$.

Solution

1. Finally we consider the digits. For this we use a lazy propagation segment tree that allows incrementing values on a range and then a collect operation that both counts the digits in the values on a range before then zeroing them out.
2. The collect operation might be slow worst case, but is amortized fast. We let the values of the segment tree count the number of occurrences of the substrings starting at our current position in the string as we iterate through it.
3. As we move from position $i-1$ to i, let S and L be as before.
4. Everything but the first L substrings are no longer valid, so we count those and delete them using the collect operation on indices L to $n-1$.
5. Next we add the substrings found at our current position, incrementing the values at indices 0 through $n-S-1$.
6. This way we collect all digits in the output.

Solution

1. Finally we consider the digits. For this we use a lazy propagation segment tree that allows incrementing values on a range and then a collect operation that both counts the digits in the values on a range before then zeroing them out.
2. The collect operation might be slow worst case, but is amortized fast. We let the values of the segment tree count the number of occurrences of the substrings starting at our current position in the string as we iterate through it.
3. As we move from position $i-1$ to i, let S and L be as before.
4. Everything but the first L substrings are no longer valid, so we count those and delete them using the collect operation on indices L to $n-1$.
5. Next we add the substrings found at our current position, incrementing the values at indices 0 through $n-S-1$.
6. This way we collect all digits in the output.

Statistics at 4-hour mark: 42 submissions, 0 accepted, first after ??:??

I — Infinite Cash

Problem

Given the starting money, salary and salary frequency of a fiscally unsound man, find out how long he has before he goes broke (if ever).

Solution

I — Infinite Cash

Problem

Given the starting money, salary and salary frequency of a fiscally unsound man, find out how long he has before he goes broke (if ever).

Solution

1. There are two ways to go about this. The first is to notice that the number of days Svalur lasts can not be so high, and simply simulate $\approx 10^6$ days and return an answer, answering infinite if he has not gone broke yet.

I — Infinite Cash

Problem

Given the starting money, salary and salary frequency of a fiscally unsound man, find out how long he has before he goes broke (if ever).

Solution

1. There are two ways to go about this. The first is to notice that the number of days Svalur lasts can not be so high, and simply simulate $\approx 10^6$ days and return an answer, answering infinite if he has not gone broke yet.
2. The other is to do things more by hand. Let $b(x)$ be the number of binary digits in x. Then to check if he goes broke before his first payment, check whether $b(m) < d$. To check whether he never goes broke, check whether $d \leq b(s)$. Then if $b(d) > 12$ just print $b(m)$. Finally simulate the salary periods one by one and print when he runs out of money.

I — Infinite Cash

Problem

Given the starting money, salary and salary frequency of a fiscally unsound man, find out how long he has before he goes broke (if ever).

Solution

1. There are two ways to go about this. The first is to notice that the number of days Svalur lasts can not be so high, and simply simulate $\approx 10^6$ days and return an answer, answering infinite if he has not gone broke yet.
2. The other is to do things more by hand. Let $b(x)$ be the number of binary digits in x. Then to check if he goes broke before his first payment, check whether $b(m) < d$. To check whether he never goes broke, check whether $d \leq b(s)$. Then if $b(d) > 12$ just print $b(m)$. Finally simulate the salary periods one by one and print when he runs out of money.

Statistics at 4-hour mark: 575 submissions, 81 accepted, first after 00:12

J — Jamboree

Problem

Partition N items among M scouts so that no one has more than two items and the maximum total size is minimised.

Solution

Statistics at 4-hour mark: 442 submissions, 163 accepted, first after HH:MM

J — Jamboree

Problem

Partition N items among M scouts so that no one has more than two items and the maximum total size is minimised.

Solution

1. Sort the N items after decreasing size.

Statistics at 4-hour mark: 442 submissions, 163 accepted, first after HH:MM

J — Jamboree

Problem

Partition N items among M scouts so that no one has more than two items and the maximum total size is minimised.

Solution

1. Sort the N items after decreasing size.
2. If $N \leq M$, the answer is the size of largest item.

Statistics at 4-hour mark: 442 submissions, 163 accepted, first after HH:MM

J — Jamboree

Problem

Partition N items among M scouts so that no one has more than two items and the maximum total size is minimised.

Solution

1. Sort the N items after decreasing size.
2. If $N \leq M$, the answer is the size of largest item.
3. If $N > M$, pair up the $M+1$:th largest item with the M:th largest, the $M+2$:th largest item with the $M-1$:th largest, and so on until we have M groups of pairs and single elements. Report the maximum size.

Statistics at 4-hour mark: 442 submissions, 163 accepted, first after HH:MM

J — Jamboree

Problem

Partition N items among M scouts so that no one has more than two items and the maximum total size is minimised.

Solution

1. Sort the N items after decreasing size.
2. If $N \leq M$, the answer is the size of largest item.
3. If $N > M$, pair up the $M + 1$:th largest item with the M:th largest, the $M + 2$:th largest item with the $M - 1$:th largest, and so on until we have M groups of pairs and single elements. Report the maximum size.
4. This greedy strategy works since for any four sizes $a \leq b \leq c \leq d$, the best way to divide them in two pairs minimising the maximum is $a + d$ and $b + c$.

Statistics at 4-hour mark: 442 submissions, 163 accepted, first after HH:MM

K — Knitting Pattern

Problem

Figure out how to centre a knitting pattern on a sweater.

Solutiom

K — Knitting Pattern

Problem

Figure out how to centre a knitting pattern on a sweater.

Solutiom

1. We place the pattern in the centre, reaching $p/2$ away from the middle in either direction. Then we have $(n-p)/2$ left on either side.

K — Knitting Pattern

Problem

Figure out how to centre a knitting pattern on a sweater.

Solutiom

1. We place the pattern in the centre, reaching $p/2$ away from the middle in either direction. Then we have $(n-p)/2$ left on either side.
2. We take this modulo p to get the leftover space on one side, so multiply by two to get our answer.

K — Knitting Pattern

Problem

Figure out how to centre a knitting pattern on a sweater.

Solutiom

1. We place the pattern in the centre, reaching $p/2$ away from the middle in either direction. Then we have $(n - p)/2$ left on either side.
2. We take this modulo p to get the leftover space on one side, so multiply by two to get our answer.
3. The only exception is if we fit one more pattern exactly in the middle on the back, so when the leftover space is exactly p or equivalently when p divides n. In this case the answer is zero instead.

K — Knitting Pattern

Problem

Figure out how to centre a knitting pattern on a sweater.

Solutiom

1. We place the pattern in the centre, reaching $p/2$ away from the middle in either direction. Then we have $(n - p)/2$ left on either side.
2. We take this modulo p to get the leftover space on one side, so multiply by two to get our answer.
3. The only exception is if we fit one more pattern exactly in the middle on the back, so when the leftover space is exactly p or equivalently when p divides n. In this case the answer is zero instead.

Statistics at 4-hour mark: 687 submissions, 152 accepted, first after 00:04