Task 1: **Area**

Authored by: Ling Yan Hao

Prepared by: Tan Yi Kai

Editorial written by: Tan Yi Kai

Subtask 1

Additional constraints: $n = 1$

As $n = 1$, the first rectangle must have the largest area. Hence, calculate and output the area of the first rectangle by multiplying $h[1]$ and $w[1]$. In Python, the code is as follows:

```python
n = int(input())
h,w = input().split()
area = int(h) * int(w)

print(area)
```

In C++, the code is as follows:

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
   int n;
   cin>>n;
   int h,w;
   cin>>h>>w;

   cout<<h*w;
}
```

Subtask 2

Additional constraints: None

Maintain a current largest area. Suppose we name this variable max_area. As every rectangle has some area, max_area can be initialised to 0.

Use a for loop to repeatedly read $h[i]$ and $w[i]$ from standard input. At each step after reading $h[i]$ and $w[i]$, compute the area $h[i] \cdot w[i]$. Using an if statement, check if this area is greater than max_area, and update v to $h[i] \cdot w[i]$ should that be the case.

In Python, the code is as follows:

```python
n = int(input())

max_area = 0

for i in range(0,n):
   h,w = input().split()
   area = int(h) * int(w)
   if area > max_area:
      max_area = area

print(max_area)
```

In C++, the code is as follows:

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
   int n;
   cin>>n;
   int max_area = 0;

   for(int i = 0;i < n;i++){
      int h,w;
      cin>>h>>w;
      int area = h*w;
      if(area>max_area)max_area=area;
   }
   cout<<max_area;
}
```

Task 2: `Swords`

Authored by: Ling Yan Hao

Prepared by: Tan Yi Kai

Editorial written by: Tan Yi Kai

Subtask 1

Additional constraints: $n \leq 500$

Sword i is useless if there is another sword j such that $a[j] \geq a[i]$ and $b[j] \geq b[i]$.

Hence, for every sword i, we iterate all possible options for sword j. If there exists a sword j that makes sword i useless, we mark sword i as useless. Else, we mark it as useful.

Then, we count the number of swords that is marked as useful.

Time complexity: $O(n^2)$

Subtask 2

Additional constraints: $a[i] \leq 500, b[i] \leq 500$

Observe that if two swords have the same attack, one of them is useless. That is, if there exists swords i and j such that $a[i] = a[j]$, then $b[i] \neq b[j]$. If $b[i] > b[j]$, then sword j is useless. If $b[i] < b[j]$, then sword i is useless. Evidently, if two swords have the same attack, the one with lower defence is useless.

As such, for all swords with the same attack value, we should only pick 1 sword with the greatest defence value as all other swords will be useless. Given that there are only 500 possible attack values from 1 to 500, this leaves us with at most 500 swords with unique attack values left.

With 500 swords left, we can then use the $O(n^2)$ solution from subtask 1.

Time complexity: $O(k + k^2)$ where k is the limit for $a[i]$

Subtask 3

Additional constraints: $a[i] = i$

This subtask is meant to hint contestants towards the full solution.

Sword i is useless if there is another sword j such that $a[i] \leq a[j]$ and $b[i] \leq b[j]$. Since $a[i] = i$, the condition $a[i] \leq a[j]$ is equivalent to $i < j$.

Hence, sword i is useless if there is another sword j such that $j > i$ and $b[j] \geq b[i]$.

Hence, we can iterate from sword n to sword 1, maintaining a maximum defence D. When we are processing sword i, we check if any of the previously processed swords have defence greater than $b[i]$.

To do this, we can simply check if $D \geq b[i]$ as D is the maximum defence of the swords processed. If $D \geq b[i]$, then there exists a previously processed sword j where $j > i$ and $b[j] = D \geq b[i]$, so sword i is useless. Otherwise, sword i should be marked as useful, and the value of D should be updated to $b[i]$ before processing $b[i-1]$.

Time complexity: $O(n)$

Subtask 4

Additional constraints: $a[i] \neq a[j]$ for every $1 \leq i < j \leq n$.

This subtask is meant for contestants going for the full solution but have issues with the edge case where $a[i] = a[j]$.

Firstly, sort the swords non-decreasingly by the value of $a[i]$.

Under the new ordering of the swords, $i < j$ will imply $a[i] < a[j]$. The same solution in subtask 3 can then be used, processing the swords from highest to lowest $a[i]$ value.

Time complexity: $O(n \log n)$ due to sorting

Subtask 5

Additional constraints: None

The full solution is similar to the solution to subtask 4.

Firstly, sort the swords non-decreasingly first by the value of $a[i]$, then by $b[i]$.

The solution in subtask 3 assumes that if $i < j$, then $a[i] < a[j]$ and j only makes i useless if $b[i] < b[j]$.

Due to the constraints, it is possible that $a[i] = a[j]$ in this subtask. Due to the way we sort, $a[i] = a[j]$ guarantees that $b[i] < b[j]$, so it is still sufficient to check that $b[i] < b[j]$ to check if j makes i useless.

Hence, the solution in subtask 3 can be used.

Time complexity: $O(n \log n)$ due to sorting

Task 3: `Dolls`

Authored by: Ling Yan Hao

Prepared by: Marc Phua Hsiao Meng

Editorial written by: Tan Yi Kai

Subtask 1

Additional constraints: $n \leq 200$

Suppose you are given some dolls and are tasked to compute the maximum size of a doll stack. One way to do it is to first sort the dolls by size. Then, iterating from the smallest to largest doll, we maintain the number of dolls taken ans and size of the last doll taken L to be in the doll stack.

Let the current doll be of size s. If $L + 1 < s$, then we should take the current doll: update ans to $ans + 1$ and update L to s. Else, we shouldn't take the current doll.

The described procedure would run in $O(nlogn)$ due to sorting. For the i^{th} day, we can run the above solution with dolls 1 to i in $O(ilogi)$.

Time complexity: $O(n^2 \log n)$. Alternate slower solutions $O(n^3)$ should pass too.

Subtask 2

Additional constraints: $a[i]$ is not a multiple of 2

As $a[i]$ is not a multiple of 2, all $a[i]$ are odd numbers. If $a[i] \neq a[j]$, then their difference must be at least 2. As such, if we can pick a subset of dolls with unique sizes, they will definitely form a valid doll stack after their sizes are sorted.

Therefore, we only need to maintain the number of unique doll sizes for the first i days as i increases. This can be done with a set data structure in both Python and C++.

Time complexity: $O(n \log n)$

Alternatively, since $a[i] \leq 500000$, one can store a boolean array of whether each doll size has previously appeared. If it hasn't appeared, then it should be updated in the boolean array and the count of unique doll sizes should be increased.

Time complexity: $O(n)$

Subtask 3

Additional constraints: $a[i]$ is not a multiple of 4

This subtask is meant to hint at the full solution.

As $a[i] \not\equiv 0 \pmod 4$, $a[i] \equiv 1, 2$ or $3 \pmod 4$. There will be at most 3 dolls of consecutive sizes, of which 2 can be taken.

Iterate from doll 1 to n, maintaining a current answer ans. When we process a doll of size $a[i]$, we find it's "group" by finding $\lfloor \frac{a[i]}{4} \rfloor$.

If there are previously other dolls of the same size, we ignore the current doll.

If there are previously other dolls of the same group, we check if that group now has a doll of size $a[i] \equiv 1 \pmod 4$ and another doll of size $a[i] \equiv 3 \pmod 4$, given that the $a[i]$ being updated is one of them. If it does, we increment ans.

The data about dolls in the same group can be quickly checked and updated using an array of size $500000 \div 4 = 125000$.

Time complexity: $O(n)$

Subtask 4

Additional constraints: None

Now, we need to be able to handle longer ranges of consecutive doll sizes.

Maintain processed doll sizes as ranges using union find disjoint set (UFDS).

The UFDS starts with 500000 "unactivated" nodes. Maintain the size of the connected component in the union find disjoint set. The answer at any point will always be the sum of $\lceil \frac{size}{2} \rceil$ for all connected components. As such, when 2 components of sizes $size_i$ and $size_j$ merge, the answer should be incremented by $\lceil \frac{size_i + size_j}{2} \rceil - \lceil \frac{size_i}{2} \rceil - \lceil \frac{size_j}{2} \rceil$. When nodes are "activated", they are taken to form a new connected component of size 1, therefore incrementing the answer by 1.

Iterate from doll 1 to n. If there was previously a doll of size $a[i]$, the size of the doll stack definitely remains the same.

Otherwise, we activate node $a[i]$ in the UFDS. If node $(a[i]-1)$ is already activated, merge $a[i]$ and $(a[i]-1)$ in the UFDS. If node $(a[i]+1)$ is already activated, merge $a[i]$ and $(a[i]+1)$ in the UFDS. Then, output the answer maintained in the modified UFDS.

Alternate ways to maintain the merging of ranges will work as long as they run in $O(n \log n)$. One example would be to use an ordered set of lower and upper bounds.

Task 4: Burgers

Authored by: Lim Rui Yuan, Benson Lin, Ling Yan Hao

Prepared by: Lim Rui Yuan

Editorial written by: Lim Rui Yuan

Subtask 1

Additional constraints: $a[i] = b[i]$

Since both recipes are the same, only one of the recipes matter. For each ingredient i, $\lfloor \frac{x[i]}{a[i]} \rfloor$ is a hard limit on the number of burgers we can make based on that ingredient. As such, the answer is the minimum of $\lfloor \frac{x[i]}{a[i]} \rfloor$ across all ingredients i.

Time complexity: $\mathcal{O}(n)$

Subtask 2

Additional constraints: $n, x[i] \leq 100$

For this editorial, let A burger be the burger using the first recipe and B burger be the burger using the second recipe.

Since n and $x[i]$ are small here, we can test all combinations of how many A burgers and how many B burgers we want to make. Since $x[i] \leq 100$, the number of A burgers and B burgers we can make ranges between 0 and 100.

Time complexity: $\mathcal{O}(n(x_{max})^2)$

Subtask 3

Additional constraints: $n, x[i] \leq 1500$

Suppose we fix the number of A burgers we want to make, this number ranges from 0 to 1500. We will have some leftover ingredients. Using these leftover ingredients, we can determine the maximum of B burgers we can make, by iterating through all n ingredients and taking the minimum of $\lfloor \frac{x[i]}{b[i]} \rfloor$, just like in subtask 1.

Time complexity: $\mathcal{O}(nx_{max})$

Subtask 4

Additional constraints: None

Firstly, we can observe that if it is possible to make $k + 1$ burgers, we can make k burgers. Using this, we can binary search the total number of burgers we want to make.

Let's suppose we want to make k burgers in total, we will now describe an algorithm for determining if it's possible to make k burgers in total. Consider each ingredient i, if $a[i] < b[i]$, then this gives us a lower bound on the number of A burgers we need to make to achieve k burgers, since making too many B burgers will require too many ingredients. Reversely, if $a[i] > b[i]$, then this gives us an upper bound on the number of A burgers we can make. With a lower and an upper bound, we can check if there exists a good integer value within those bounds, and if such value exists, then it is possible to make k burgers.

More specifically, there are three cases to consider:

1. $a[i] = b[i]$, then we have to check if $a[i] * k \leq x[i]$
2. $a[i] < b[i]$, then we must make minimally $k - \lfloor \frac{x[i]-a[i]\times k}{b[i]-a[i]} \rfloor$ A burgers
3. $a[i] > b[i]$, the we must make maximally $\lfloor \frac{x[i]-b[i]\times k}{a[i]-b[i]} \rfloor$ A burgers

Finally, we check if the lower bound is not larger than the upper bound.

Task 5: `Network`

Authored by: Ashley Aragorn Khoo

Prepared by: Stuart Lim Yi Xiong

Editorial written by: Stuart Lim Yi Xiong, Ling Yan Hao

Subtask 1

Additional constraints: $a[j] = b[j]$

Each application has both its databases in the same server. The condition given in the problem statement reduces to the following: application j is working if and only if server $a[j]$ is active.

To ensure that all applications are not working, all servers $a[j]$ must be chosen to be deactivated. Deactivating the remaining servers will not affect whether the applications are working, but we want to minimise the number of deactivated servers, so they must not be chosen. The required set of deactivated servers is therefore exactly the set of servers $a[j]$ that appear in the input. We find all of the **unique** values that appear in the array a, count them as well as output them according to the output format.

Time complexity: $O(n + m)$

Subtask 2

Additional constraints: $u[i] = i, v[i] = i + 1$

The conditions for an application j to be working do not change when $a[j]$ and $b[j]$ are swapped. Hence, for this subtask only, assume $a[j] \leq b[j]$. If this is not the case in the input, swap $a[j]$ and $b[j]$.

Algorithm: Sort the applications by increasing $a[j]$, breaking ties arbitrarily. Maintain a set S, which is initially empty. For each application j:

- Let x be the smallest number in S. If $a[j] > x$, deactivate x and empty S. (If S is empty, skip this step.)
- Add $b[j]$ into S.

At the end, deactivate the smallest number in S. (We can speed this up further by simply storing the smallest number in S instead of the whole set S.)

First, we prove that this algorithm deactivates all the applications. Observe that when we deactivate x, for each j' corresponding to items in S, we must have $b[j'] \geq x$ (since x is the smallest). Also, we have $a[j'] \leq x$, or else S would have been emptied earlier.

To show that this is the minimum, observe that everytime we empty S, x is generated by some application $(a[j], b[j])$. It can be shown that the collection of these $\{(a_j, b_j)\}$ correspond to disjoint intervals, and therefore each of them must be deactivated by a different server. Therefore this is the minimum number required.

Time complexity: $O(n + m \log m)$

Subtask 3

Additional constraints: $n, m \leq 15$

This subtask is for solutions with exponential time and/or space complexity. One such solution will be discussed below.

Let $S[j]$ be the set of all vertices lying on the path from $a[j]$ to $b[j]$ for each j. Notice that if the network tree is rooted at vertex $a[j]$, then a vertex v lies along the path if and only if v is an ancestor of $b[j]$. Hence, we can perform depth-first search (DFS) from $a[j]$ to find the immediate parent of each vertex, then repeatedly move up from vertex $b[j]$ to find all vertices along the path. This procedure runs in $O(n)$ time per application, giving a total time complexity of $O(nm)$. Storing all $S[j]$ also gives a space complexity of $O(nm)$.

We iterate over all 2^n possible subsets of vertices to choose, C, and check if each C can cause all applications to not work. Application j does not work if and only if there is a vertex that is in both C and $S[j]$. This check can be done in $O(|C| \cdot |S[j]|) = O(nm)$ time per subset.

Time complexity: $O(2^n nm)$

Space complexity: $O(nm)$

Subtask 4

Additional constraints: $n, m \leq 2000$

We begin by rooting the tree at an arbitrary vertex and performing DFS to find the immediate parent and depth of each vertex. For each application j, first compute the lowest common ancestor of $a[j]$ and $b[j]$, which we will refer to as $\ell[j]$. For this subtask, an algorithm finding the lowest common ancestor in $O(n)$ time is sufficient.

Claim: Suppose there exists a value of j such that for all $j' \neq j$, $\ell[j']$ is not a proper descendant of $\ell[j]$, then there exists an optimal solution that deactivates $\ell[j]$.

Proof of claim: Consider any solution, and suppose that application j is deactivated by node x (if there are multiple such x, choose any). We undo the deactivation of x, and deactivate $\ell[j]$ instead. It remains to show that this new configuration deactivates all servers. Suppose by contradiction, there exists some j' which is deactivated by x, but is not deactivated by $\ell[j]$. Since x deactivates j', one of $a[j']$ and $b[j']$ is a descendant of x, WLOG assume $a[j']$ is a descendant of x (which is also a descendant of $\ell[j]$). If $b[j']$ is also a descendant of ℓ, then $\ell[j']$ is a common ancestor of both $a[j']$ and $b[j']$. On the other hand, we assumed that $\ell[j']$ is not a proper descendant of $\ell[j]$, so $\ell[j'] = \ell[j]$, which means that the unique path from $a[j']$ to $b[j']$ passes through $\ell[j]$. If $b[j']$ is not a descendant of ℓ, then the only path from $a[j']$ to $b[j']$ passes through $\ell[j]$. In any case, $\ell[j]$ deactivates j'.

This suggests the following algorithm: first compute $\ell[j]$ for all j, and keep a set S of activated applications, which is initialized to $\{1, 2, \ldots, m\}$. While S is not empty, extract some j such that $\ell[j]$ has the maximum distance from the root, and deactivate $\ell[j]$. Once this happens, find all j' which is deactivated by $\ell[j]$, and remove them from S.

Note that we can delay the deactivation of applications j'. Rather, we process applications by decreasing depth of $\ell[j]$, and check if the application has been deactivated previously. To do so, iterate over all vertices between $a[j]$ and $b[j]$ and see if any has been deactivated. Sorting the applications takes $O(m \log m)$ time, whereas checking if an application is already deactivated takes $O(n)$ time per application.

Time complexity: $O(n^2 + nm + m \log m)$

Subtask 5

Additional constraints: None

We speed up the solution from Subtask 4.

Now, we must compute lowest common ancestors in $O(\log n)$ time or better.

We also need to speed up the process of checking if an application has been deactivated. Notice that a path from $a[j]$ to $b[j]$ can be split into two paths from $\ell[j]$ to $a[j]$ and $b[j]$. Let $p[i]$ be the number of deactivated vertices on the path from the root to vertex i, and let $p[parent[root]] = 0$. Since $\ell[j]$ is an ancestor of $a[j]$, there exists a deactivated vertex on the path from $\ell[j]$ to $a[j]$ if and only if $p[a[j]] > p[parent[\ell[j]]]$. Similarly, we can check if there is a deactivated vertex on the path from $\ell[j]$ to $b[j]$.

We are left with quickly updating and querying values of $p[i]$. Notice that when a vertex x is deactivated, $p[x']$ increases by 1, for all x' that are in the subtree rooted at x. Since a rooted subtree of vertices has consecutive labels in the pre-ordering of a tree, by converting all vertex labels to their pre-order labels, increasing $p[x']$ by 1 becomes a range update. Using a Fenwick tree, range updates to $p[x']$ and point queries of $p[i]$ can both be done in $O(\log n)$ time.

Time complexity: $O(n \log n + m \log n + m \log m)$