1 Vases

Since each vase must have at least one flower and every vase must have a different number of flowers, the smallest number of flowers we can arrange is $1+2+3=6$. So, if $N < 6$, we should output `0 0 0` since it is impossible. If $N \geq 6$, we can put 1 flower in the first vase, 2 flowers in the second vase, and $N-3$ in the third vase.

This solution has been implemented below in C++ and Python.

```cpp
// C++ Solution to AIO 2019 Vases

#include <cstdio>
int main() {
    // Open vasesin.txt and make scanf read from it
    freopen("vasesin.txt", "r", stdin);

    // Open vasesout.txt and make printf write to it
    freopen("vasesout.txt", "w", stdout);

    // Read the value of n
    int n;
    scanf("%d", &n);

    // Write the answer to the output file, and \n for a newline
    if (n < 6) printf("0 0 0\n");
    else printf("1 2 %d\n", n-3);
}
```

```python
# Python3 Solution to AIO 2019 Vases

import sys

# Open the input and output files
input_file = open("vasesin.txt", "r")
output_file = open("vasesout.txt", "w")

# Read the value of n
n = int(input_file.readline())

# Write the answer to the output file, and \n for a newline
if n < 6:
    output_file.write("0 0 0\n")
else:
    output_file.write(f"1 2 {n-3}\n")
```

2 RPS

To maximise the number of points, we should try to win as many rounds as possible, and then draw as many of the remaining ones as we can.

When the opponent throws rock, we win if we throw paper. Hence, out of the opponents R_a rocks, we can win all of them if $R_a \leq P_b$ and we can win P_b of them if $R_a > P_b$. Hence, we can win $R_{win} = \min(R_a, P_b)$ of the rounds, where min gives the smallest of the two values. Similarly, we win $P_{win} = \min(P_a, S_b)$ of the rounds where the opponent throws paper, and $S_{win} = \min(S_a, R_b)$ of the rounds where the opponent throws scissors.

Now, consider draws. We draw when both players throw the same thing. How many times can both players throw heads? Well, there are $R_a - R_{win}$ rock throws remaining from our opponent, and $R_b - P_{win}$ rock throws remaining from us, and we can draw $R_{draw} = \min(R_a - R_{win}, R_b - S_{win})$ of these. Similarly, $P_{draw} = \min(P_a - P_{win}, P_b - R_{win})$ and $S_{draw} = \min(S_a - S_{win}, S_b - P_{win})$.

We lose the remaining throws, and so we lose $N - (R_{win} + P_{win} + S_{win} + R_{draw} + P_{draw} + S_{draw})$ throws.

This solution has been implemented below in C++ and Python.

```cpp
// C++ Solution to AIO 2019 RPS

#include <cstdio>
#include <algorithm>
using namespace std;
int main() {
    // Open the input/output files
    freopen("rpsin.txt", "r", stdin);
    freopen("rpsout.txt", "w", stdout);

    // Scan in the input
    int N, Ra, Pa, Sa, Rb, Pb, Sb;
    scanf("%d%d%d%d%d%d%d", &N, &Ra, &Pa, &Sa, &Rb, &Pb, &Sb);

    // Calculate the wins
    int Rwin = min(Ra, Pb);
    int Pwin = min(Pa, Sb);
    int Swin = min(Sa, Rb);

    // Calculate the draws
    int Rdraw = min(Ra-Rwin, Rb-Swin);
    int Pdraw = min(Pa-Pwin, Pb-Rwin);
    int Sdraw = min(Sa-Swin, Sb-Pwin);

    // Calculate the losses
    int losses = N-(Rwin+Pwin+Swin+Rdraw+Pdraw+Sdraw);

    // Print the answer
    printf("%d\n", Rwin+Pwin+Swin-losses);
}
```

```python
# Python3 Solution to AIO 2019 RPS

import sys

# Open the input/output files
input_file = open("rpsin.txt", "r")
output_file = open("rpsout.txt", "w")
```

```
# Scan in the input
N = int(input_file.readline().strip())
Ra, Pa, Sa = map(int, input_file.readline().split())
Rb, Pb, Sb = map(int, input_file.readline().split())

# Calculate the wins
Rwin = min(Ra, Pb);
Pwin = min(Pa, Sb);
Swin = min(Sa, Rb);

# Calculate the draws
Rdraw = min(Ra-Rwin, Rb-Swin);
Pdraw = min(Pa-Pwin, Pb-Rwin);
Sdraw = min(Sa-Swin, Sb-Pwin);

# Calculate the losses
losses = N-(Rwin+Pwin+Swin+Rdraw+Pdraw+Sdraw);

# Print the answer
output_file.write(str(Rwin+Pwin+Swin-losses));
```

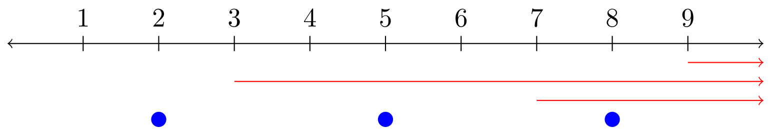
3 Hiring Monks

3.1 $S = 0$ Subtasks

We will first consider the case when $S = 0$, which means there are no student jobs. This represents subtask 3 and 4, which combined are worth 50 marks.

Since $S = 0$, all jobs are master jobs. We will motivate our solution using an example.

Consider the example below. The red arrows represent the jobs, with the arrows covering the skill levels that are allowed to do that job, and the blue dots represent the monks.



The monk with the most skill is the monk with $x_i = 8$. Which job should we assign them to? We can't assign them to the first job with $m_k = 9$, since the monk doesn't have the skill level required. What about the second job, with $m_k = 3$. We could assign the monk to this job, but if we did there would be no jobs left for other monks, as they have too little skill for either of the other jobs. If we assign the monk to the job with $m_k = 7$, then we can assign the job with $m_k = 3$ to the monk with skill 5, which allows 2 jobs to be assigned.

Generalising this, the monk with the highest skill level will always have the most jobs available to it. As such, we should assign them to the highest skilled job they can do, to maximise the opportunities we have to assign the other monks to jobs.

We can repeat this process with the second highest monk, giving them the highest remaining job possible, and then do the same with the third highest monk, etc. At some point there will be a monk which we cannot assign a job to. Since all remaining monks have a skill level lower (or the same) as this one, this means that we cannot assign jobs to any more monks, and so we are finished.

This gives us the following solution:

1. Consider the largest remaining monk, with skill level x_i.
2. Remove all jobs with $m_k > x_i$, as no monk will ever be able to do these jobs.
3. Give this monk the highest skilled remaining job, and remove this job from future consideration. If there is no available job, we are finished.
4. Repeat from step 1 if there is still a monk remaining.

This is known as a *greedy* algorithm because at each step we choose the best available option, and this creates an optimal solution.

In subtask 4, we need to do this solution efficiently or it will be too slow. You can read about how to measure the efficiency of a solution in this tutorial. To do the solution efficiently, we can sort the monks and jobs by descending skill level and keep track of which job and which monk we are up to. This solution is $O(N \log N)$, because the time complexity of sorting is $O(N \log N)$. See the code below for more details.

```
// C++ Solution to AIO 2019 Hiring Monks when S = 0

#include <cstdio>
#include <algorithm>
#include <vector>
#define MAXN 100010
using namespace std;
```

```
int monks[MAXN], master_jobs[MAXN];
int main() {
    // Open the input/output files
    freopen("hirein.txt", "r", stdin);
    freopen("hireout.txt", "w", stdout);

    // Scan in the input
    int N, S, M;
    scanf("%d", &N);
    for (int i = 0; i < N; i++) scanf("%d", &monks[i]);
    scanf("%d", &S); // We assume S = 0
    scanf("%d", &M);
    for (int i = 0; i < M; i++) scanf("%d", &master_jobs[i]);

    // Sort the monks and master jobs
    sort(monks, monks+N);
    sort(master_jobs, master_jobs+M);
    // We want descending order, but by default sort gives ascending, so we must
         reverse to get the correct order
    reverse(monks, monks+N);
    reverse(master_jobs, master_jobs+M);

    // Now, loop over the monks in descending order and give them the highest
        skilled available master job
    int job = 0; // the current highest skilled available job
    int master_answer = 0;
    for (int monk = 0; monk < N; monk++) {
        // First, remove any jobs which have too high skill
        while (job < M && master_jobs[job] > monks[monk]) {
            job++;
        }

        // Stop if there are no jobs left
        if (job == M) {
            break;
        }

        // Increase the answer and remove the job since we are giving it to this
             monk
        master_answer++;
        job++;
    }

    // Print the answer
    printf("%d\n", master_answer);
}
```

```
# Python3 Solution to AIO 2019 Hiring Monks when S = 0

import sys

# Open the input/output files
input_file = open("hirein.txt", "r")
output_file = open("hireout.txt", "w")

# Scan in the input
```

```python
N = int(input_file.readline().strip())
monks = []
for _ in range(N):
    monks.append(int(input_file.readline().strip()))
S = int(input_file.readline().strip()) # We assume S = 0
M = int(input_file.readline().strip())
master_jobs = []
for _ in range(M):
    master_jobs.append(int(input_file.readline().strip()))

# Sort the monks and jobs
monks.sort(reverse=True) # the reverse=True specifies descending order
master_jobs.sort(reverse=True)

# Now, loop over the monks in descending order and give them the highest skilled
     available job
job = 0; # the current highest skilled available job
master_answer = 0;
for monk in monks:
    # First, remove any jobs which have too high skill
    while job < M and master_jobs[job] > monk:
        job += 1

    # Stop if there are no jobs left
    if job == M:
        break

    # Increase the answer and remove the job since we are giving it to this monk
    master_answer += 1
    job += 1

# Print the answer
output_file.write(str(master_answer));
```

3.2 Full Solution

We will now consider the full solution. One approach is to first assign master jobs to monks, and then assign the student jobs to the monks who didn't get master jobs. What happens if we do this? First, we run the $S = 0$ solution described in subsection 3.1 to assign master jobs to monks. This will assign x jobs (for some x) to the x most skilled monks.

Now consider a modification of the $S = 0$ solution which instead assigns student jobs to monks. This would be the same, except we process monks and student jobs in ascending order instead of descending order. This will assign y jobs (for some y) to the y least skilled monks.

If $x + y \leq N$, there will be no overlap between the student and master jobs assigned (that is, no monk has been given a master and a student job), and so the answer is $x + y$.

If $x + y > N$, there will be overlap between the jobs assigned (that is, some monks are assigned two jobs), but every monk will be assigned at least one job. Hence, the answer is N.

In both cases, the answer is $\min(N, x + y)$.

Consider the time complexity of the solution. Because we are essentially running our old $O(N \log N)$ solution twice, the time complexity of this solution is also $O(N \log N)$, which is fast enough to solve the problem.

This solution has been implemented below in C++ and Python.

```
// C++ Solution to AIO 2019 Hiring Monks

#include <cstdio>
#include <algorithm>
#include <vector>
#define MAXN 100010
using namespace std;
int monks[MAXN], master_jobs[MAXN], student_jobs[MAXN];
int main() {
    // Open the input/output files
    freopen("hirein.txt", "r", stdin);
    freopen("hireout.txt", "w", stdout);

    // Scan in the input
    int N, S, M;
    scanf("%d", &N);
    for (int i = 0; i < N; i++) scanf("%d", &monks[i]);
    scanf("%d", &S);
    for (int i = 0; i < S; i++) scanf("%d", &student_jobs[i]);
    scanf("%d", &M);
    for (int i = 0; i < M; i++) scanf("%d", &master_jobs[i]);

    // Sort the monks and master jobs
    sort(monks, monks+N);
    sort(master_jobs, master_jobs+M);
    // We want descending order, but by default sort gives ascending, so we must
         reverse to get the correct order
    reverse(monks, monks+N);
    reverse(master_jobs, master_jobs+M);

    // Now, loop over the monks in descending order and give them the highest
        skilled available master job
    int job = 0; // the current highest skilled available job
    int master_answer = 0;
    for (int monk = 0; monk < N; monk++) {
        // First, remove any jobs which have too high skill
        while (job < M && master_jobs[job] > monks[monk]) {
            job++;
        }

        // Stop if there are no jobs left
        if (job == M) {
            break;
        }

        // Increase the answer and remove the job since we are giving it to this
             monk
        master_answer++;
        job++;
    }

    // Now, sort the monks and student jobs in ascending order, and do the same
        process again
    sort(monks, monks+N);
    sort(student_jobs, student_jobs+S);
    job = 0;
    int student_answer = 0;
    for (int monk = 0; monk < N; monk++) {
        // Remove any jobs which have too low skill
        while (job < S && student_jobs[job] < monks[monk]) {
            job++;
        }

        // Stop if there are no jobs left
        if (job == S) {
            break;
        }

        // Increase the answer and remove the job since we are giving it to this
             monk
        student_answer++;
        job++;
    }

    // Print the answer
    printf("%d\n", min(N, master_answer+student_answer));
}
```

```
# Python3 Solution to AIO 2019 Hiring Monks

import sys

# Open the input/output files
input_file = open("hirein.txt", "r")
output_file = open("hireout.txt", "w")

# Scan in the input
N = int(input_file.readline().strip())
monks = []
for _ in range(N):
    monks.append(int(input_file.readline().strip()))
S = int(input_file.readline().strip())
student_jobs = []
for _ in range(S):
    student_jobs.append(int(input_file.readline().strip()))
M = int(input_file.readline().strip())
master_jobs = []
for _ in range(M):
    master_jobs.append(int(input_file.readline().strip()))

# Sort the monks and jobs
monks.sort(reverse=True) # the reverse=True specifies descending order
master_jobs.sort(reverse=True)

# Now, loop over the monks in descending order and give them the highest skilled
     available job
job = 0; # the current highest skilled available job
master_answer = 0;
for monk in monks:
    # First, remove any jobs which have too high skill
    while job < M and master_jobs[job] > monk:
            job += 1

        # Stop if there are no jobs left
        if job == M:
            break

        # Increase the answer and remove the job since we are giving it to this monk
        master_answer += 1
        job += 1

# Now, sort the monks and student jobs in ascending order, and do the same
    process again
monks.sort()
student_jobs.sort()
job = 0;
student_answer = 0;
for monk in monks:
    # Remove any jobs which have too low skill
    while job < S and student_jobs[job] < monk:
        job += 1

    # Stop if there are no jobs left
    if job == S:
        break

    # Increase the answer and remove the job since we are giving it to this monk
    student_answer += 1;
    job += 1;

# Print the answer
output_file.write(str(min(N, master_answer+student_answer)));
```

4 Medusa's Snakes

4.1 Decision Problem

Let's first consider a slightly modified version of the problem: given some venom level x, does there exist a snake with this venom level? This is known as a decision problem because the answer is either yes or no.

For example, in the input KSEESNANNAAKNKESE there is a snake with venom level $x = 1$, KSEESNANNAAKNKESE, and one with venom level $x = 2$, KSEESNANNAAKNKESE. There is no snake with venom level 3.

To check if there is a snake with venom level x, we can try to find one. We construct our snake as follows

1. Find the first x S's in the string
2. Find the first x N's in the string which occur after the S's
3. Find the first x A's in the string which occur after the N's
4. Find the first x K's in the string which occur after the A's
5. Find the first x E's in the string which occur after the K's

This will always find a snake with venom level x if one exists. Why? Using the first x S's maximises the remaining characters to use in the rest of the snake. If we skipped over an S to use a different one, we might also skip over an N which is required to finish our snake, but the method we use guarantees we don't skip over letters unless we are required to. The same argument applies to the other steps.

This is implemented below in C++ and Python as a function which takes in a string and an integer x, and returns true if there is a snake with venom level x, and false otherwise.

```cpp
// C++ implementation of hasSnake

/*
 * Returns true if there is a snake with venom level x, false otherwise
 */
bool hasSnake(int N, char snake[], int x) {
    char letters[] = { 'S', 'N', 'A', 'K', 'E' };
    int current_letter = 0; // index of the letter we are looking for
    int num_of_letter = 0; // the amount of the current letter we have found
    for (int i = 0; i < N; i++) {
        if (snake[i] == letters[current_letter]) {
            num_of_letter++;
        }
        if (num_of_letter == x) {
            if (current_letter == 4) {
                return true; // Success - we found a snake with venon level x
            } else {
                // Move onto the next letter
                current_letter++;
                num_of_letter = 0;
            }
        }
    }
    return false;
}
```

```python
# Python3 implementation of hasSnake
```

```
"""
Returns true if there is a snake with venom level x, false otherwise
"""
def hasSnake(N, snake, x):
    letters = [ 'S', 'N', 'A', 'K', 'E' ]
    current_letter = 0; # index of the letter we are looking for
    num_of_letter = 0; # the amount of the current letter we have found
    for c in snake:
        if c == letters[current_letter]:
            num_of_letter+=1;

        if num_of_letter == x:
            if current_letter == 4:
                return True; # Success - we found a snake with venon level x
            else:
                # Move onto the next letter
                current_letter+=1;
                num_of_letter = 0;
    return False;
```

4.2 Slow Solution

We can use the hasSnake function to solve the problem. Let's say the highest venom of any snake in the string is x_{max}. For any $x \leq x_{max}$, hasSnake will return true, because we can always create a smaller snake by removing characters from a bigger snake. Also, hasSnake will return false for all $x > x_{max}$, because x_{max} represents the biggest snake.

We can turn this into a solution by calling hasSnake with increasing values of x (1, 2, 3, etc), until it returns false for the first time. Subtracting 1 from this gives us the answer.

We will now find the time complexity of this solution. You can read about time complexity here if you are not familiar with it. The time complexity of hasSnake is $O(N)$, because the function has a for-loop which runs N times. The maximum possible answer is $\frac{N}{5}$, so in the worst case we call hasSnake $O(N)$ times. Hence, the time complexity is $O(N^2)$. This is fast enough when $N \leq 1000$, but is not fast enough for the subtasks where $N \leq 100\,000$.

4.3 Full Solution

We can use a technique called *binary search* to speed up the solution. We will demonstrate it using an example.

Let's say we know the answer is somewhere from 0 to 1000. We call hasSnake in the middle of this range, $x = 500$. If this returns true, we know the answer is at least 500 and if it returns false we know the answer is less than 500. Using just one call to hasSnake, we have halved the range of possible answers. We repeat the process with our new, smaller range.

For example, if the answer is 400, our binary search would proceed like this:

1. The answer is somewhere from 0 to 1000. We call hasSnake with $X = \frac{1000+0}{2} = 500$ and get false. We know the answer is somewhere from 0 to 499.
2. We call hasSnake with $X = \frac{499+0}{2} = 250$ (rounding up) and get true. We know the answer is somewhere from 250 to 499.
3. We call hasSnake with $X = \frac{250+499}{2} = 375$ (rounding up) and get true. We know the answer is somewhere from 375 to 499.

4. We call **hasSnake** with $X = \frac{375+499}{2} = 437$ and get false. We know the answer is somewhere from 375 to 436.
5. We call **hasSnake** with $X = \frac{375+436}{2} = 406$ (rounding up) and get false. We know the answer is somewhere from 375 to 405.
6. We call **hasSnake** with $X = \frac{375+405}{2} = 390$ and get true. We know the answer is somewhere from 390 to 405.
7. We call **hasSnake** with $X = \frac{390+405}{2} = 398$ (rounding up) and get true. We know the answer is somewhere from 398 to 405.
8. We call **hasSnake** with $X = \frac{398+405}{2} = 402$ (rounding up) and get false. We know the answer is somewhere from 398 to 401.
9. We call **hasSnake** with $X = \frac{398+401}{2} = 400$ (rounding up) and get true. We know the answer is somewhere from 400 to 401.
10. We call **hasSnake** with $X = \frac{400+401}{2} = 401$ (rounding up) and get false. We know the answer is somewhere from 400 to 400, and so must be 400.

This finds the answer despite calling **hasSnake** only 10 times. In fact, because the binary search halves the range of possible answers each time it calls **hasSnake**, it only calls **hasSnake** $\log_2 N$ times, which is 17 when $N = 100\,000$. This gives a solution with a time complexity of $O(N \log N)$, which is fast enough to solve the problem.

This solution has been implemented below in C++ and Python. Note that binary search can be finicky to implement, and can often lead to off-by-1s or infinite loops if there is a mistake. For example, this binary search would infinite loop if we rounded down instead of rounding up.

```cpp
// C++ Solution to AIO 2019 Medusa's Snakes

#include <cstdio>
#include <algorithm>
using namespace std;

/*
 * Returns true if there is a snake with venom level x, false otherwise
 */
bool hasSnake(int N, char snake[], int x);

int N;
char snake[100010];
int main() {
    // Open the input/output files
    freopen("snakein.txt", "r", stdin);
    freopen("snakeout.txt", "w", stdout);
    scanf("%d %s", &N, snake);

    // lo and hi represent the range (inclusive) that the answer is in during
       our binary search
    int lo = 0, hi = N;
    while (lo != hi) {
        int mid = (lo+hi+1)/2; // rounds up
        if (hasSnake(N, snake, mid)) {
            lo = mid;
        } else {
            hi = mid-1;
        }
    }

    // Print the answer
    printf("%d\n", lo);
}
```

```
# Python3 Solution to AIO 2019 Medusa's Snakes

import sys

"""
Returns true if there is a snake with venom level x, false otherwise
"""
def hasSnake(N, snake, x)

input_file = open("snakein.txt", "r")
output_file = open("snakeout.txt", "w")

# Scan in the input
N = int(input_file.readline().strip())
snake = input_file.readline().strip()

# lo and hi represent the range (inclusive) that the answer is in during our
    binary search
lo = 0
hi = N
while lo != hi:
    mid = (lo+hi+1)//2; # rounds up
    if hasSnake(N, snake, mid):
        lo = mid;
    else:
        hi = mid-1;

# Print the answer
output_file.write(str(lo));
```