Calorie Counting

Problem Description

The full description can be found in this site's repository.

The data is formatted as a series of integers, with each Cadet's data separated by a blank line. The data is stored in the file `input.txt`. A sample input file is provided below:

```
1000
2000
3000

4000

5000
6000

7000
8000
9000

10000
```

> Given the input file, `input.txt`, figure out which 3 Cadets have the most number of Calories in their rucksack, and find the total number of Calories that each of them have.
>
> Submit your answer as an integer wrapped in the `flag{}` braces (for example, `flag{24000}`).

Walkthrough

The biggest challenge here is doing this in C. We need to read in the file, parse the data, and then sort the data. When we do this problem, we need to consider the following:

1. **How do we read in the file?** In this case, the file is a series of integers, where each is separated by a newline.
2. **What data structure will we use?** In this case, we can use a static array because it's easy to use. A linked list is also a good choice for this challenge.
3. **How are we processing the data?** We will need the top three Cadets, which requires us to sort the data and read the top three. Another solution allows us to keep track of the top three as we go, but this is a less expandable solution.

Let's discuss each of these in turn.

Reading the File

From the binex sections, we know we read files in C using `fopen()`. In case you don't remember the arguments, it takes two:

```
FILE *fopen(const char *filename, const char *mode);
```

Our filename is *input.txt*, and we only want to open the file for reading (i.e., `r`). Since this is not binary data, we don't use the `b` flag.

When we open a file, we should always check that the file opened correctly. If it didn't, we should exit the program. We can do this with the following code:

```
FILE* fp = fopen("input.txt", "r");
if (fp == NULL)
    exit(EXIT_FAILURE);
```

Now that we have the file open, we need to read the data. We can use `getline()` to read data line-by-line. This function takes three arguments:

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

Let's break this down a little more:

- `lineptr` is a pointer to a character array (i.e., a string). This is where the data will be stored.
- `n` is the size of the character array. If the data is larger than this, the function will reallocate the array to be larger.
- `stream` is the file we are reading from.
- This function returns an `ssize_t`, which is a signed integer. This is the number of characters read, or `-1` if there was an error (such as EOF).

We can use this function to read the data from the file. We will need to loop through the file until we reach the end. We can do this with the following code:

```
char* line = NULL;
size_t len = 0;
ssize_t read;

while ((read = getline(&line, &len, fp)) != -1) {
    // logic goes here
}
```

This code will read the file line-by-line, storing the data in `line`. We can then process the data in the `while` loop.

Processing the Data

Now that we have the data, we need to process it. We need to store the data in a data structure, and then sort the data. We can use a static array to handle this because this is easier to sort than a linked list.

Let's define a variable called `total` to store each Cadet's total number of Calories. Remember that static arrays cannot have their size changed, so we choose something that's large enough to hold the data. There are around `2500` lines, which means there can be a max of around `1250` Cadets (since each one must be separated by a newline).

> *This is where Linked Lists become a lot more robust than static arrays.* Linked Lists are not limited to a static size, so they can grow as needed.

We're also going to need a variable to keep track of the current Cadet, as well as a variable to track that Cadet's sum. We'll use the following initializations:

```
int totals[1250] = {0};
uint16_t i = 0;
uint32_t sum = 0;
```

Now, we need to decide for our logic as we process each line. The best way to do this is as follows:

- If there's data to collect, we convert the line into an integer and add it to the total.
- Otherwise, if we're at a newline, we add this total to the array and reset the sum.

We can do this with the following code:

```
if (read == 1) { // if blank line, store total
    totals[i] = sum;
    ++i;
    sum = 0;
} else { // else, add to total
    sum += atoi(line);
}
```

Why did we use `read == 1`? Remember that `getline()` returns the number of characters read. If we read a blank line, we only read one character (the newline). If we read a number, we read more than one character. This is why we check for `read == 1`.

> *Note: We use `atoi()` to convert the string to an integer. This function is not safe, but it's good enough for this challenge. Since we are allowed to assume that the file only contains integers, we can use this function.*

This code successfully reads the data into the array. Now, it's time for processing.

Sorting the Data

We need to sort the data in the array. We can do this using the `qsort()` function. This function takes four arguments:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

This one's quite a complicated function, so let's break it down:

- `base` is the array we are sorting.
- `nmemb` is the number of elements in the array.
- `size` is the size of each element in the array. We most often use `sizeof` and pass in the data type.
- `compar` is a functor that compares two elements. We'll most often define this ourselves because it's easier.
- This function has no return because it does an in-place sort. If we wanted to keep the original array, we would need to make a copy.

Let's write the `compar` function. See it takes two `const void*` arguments. These are pointers to the elements in the array. We need to cast these to the correct data type, and then compare them. We can do this with the following code:

```
int compare(const void* a, const void* b)
{
    return (*(int*)b - *(int*)a);
}
```

This code will sort the array from largest to smallest. *How do we know this?* We are told this by the `man` pages:

```
The comparison function must return an integer less than, equal to, or
greater than zero if the first argument is considered to be respectively
less than, equal to, or greater than the second.
```

Therefore, if `a` is greater than `b`, we return a negative number. If `a` is less than `b`, we return a positive number. If they are equal, we return `0`. This is why we return `b - a`.

> If we want to sort the array in ascending order, we would return `a - b`.

Now that we have the `compar` function, we can sort the array. We can do this with the following code:

```
qsort(totals, i, sizeof(int), compare);
```

This code will sort the array in descending order. Now, we need to print the top three Cadets. We can do this with the following code:

```
printf("flag{%d}\n", totals[0] + totals[1] + totals[2]);
```

Finally, we need to free the memory we allocated. We can do this with the following code:

```
fclose(fp);
if (line)
    free(line);
exit(EXIT_SUCCESS);
```

If we compile and run this code, we get the following output:

```
$ gcc -o solve solve.c && ./solve
flag{209914}
```

Full Solution

The full solution is below:

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

int compare(const void* a, const void* b)
{
    return (*(int*)b - *(int*)a);
}

int main(void)
{
    // open the file
    FILE* fp = fopen("input.txt", "r");
    if (fp == NULL)
        exit(EXIT_FAILURE);

    char* line = NULL;
    size_t len = 0;
    ssize_t read;

    // initialize array
    int totals[1250] = {0};
    uint16_t i = 0;
    uint32_t sum = 0;

    // read the file till no more lines
    while ((read = getline(&line, &len, fp)) != -1) {
        if (read == 1) { // if blank line, store total
            totals[i] = sum;
            ++i;
            sum = 0;
        } else { // else, add to total
```

```
            sum += atoi(line);
        }
    }

    // sort the array
    qsort(totals, i, sizeof(int), compare);

    // print the sum of the top 3
    printf("flag{%d}\n", totals[0] + totals[1] + totals[2]);

    fclose(fp);
    if (line)
        free(line);
    exit(EXIT_SUCCESS);
}
```