ECS 50: Programming Assignment #1

Instructor: Aaron Kaloti

Fall Quarter 2021

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: Explicitly stated the upper limit of the value (that you'd have to convert to/from) for part #1. Fixed a minor typo in part #1 as well.
- v.3: Pushed deadline back.
- v.4: Autograder details.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Friday, 10/15. Gradescope will say 12:30 AM on Saturday, 10/16, due to the “grace period” (as described in the syllabus). *Be careful about relying on the grace period for extra time; this could be risky.*

3 Some Relevant Concepts

3.1 The Specific C++ Standard

For all five parts of this assignment, you will write C++ code. Specifically, you will use the C++14 standard. If you are familiar with the C++11 standard, then you should barely notice a difference. If you are not even familiar with the C++11 standard, then that should not be a big deal either, since we’re not using too many of the features that C++11 has that older standards of C++ lack. The main reason that we are using C++14 instead of an older C++ standard is that some of the parts of this assignment use certain STL containers that did not exist in the older standards (i.e. before C++11).

3.2 STL Tuple

For parts #3 and #4, you will have to use the `std::tuple` container from the STL. I did not go over this container in ECS 34 during FQ20 (which some of you may have taken), and I would imagine that not all ECS 36B classes go over this container either, so I talk about it below, starting with an example that illustrates much of what you need to know and that should be straightforward.

```
$ cat tuple_example.cpp
#include <iostream>
#include <tuple>

int main()
{
    std::tuple<int, bool, int, float> t1{5, true, 18, -89.34};
    std::cout << std::get<0>(t1) << '\n';
    std::cout << std::get<3>(t1) << '\n';

    // Can use parentheses instead of braced initialization when invoking the
    // constructor, if that's what you prefer.
    std::tuple<int, std::string> t2(5, "abc");
    std::cout << std::get<1>(t2) << '\n';
    // If the tuple only contains unique types, then you can access by type
    // instead of by element.
    std::cout << std::get<int>(t2) << '\n';
    std::cout << std::get<std::string>(t2) << '\n';

    // Can assign or manipulate referencnes to specific elements.
    std::get<int>(t2) = 18;
    std::string& str = std::get<std::string>(t2);
    str[2] = 'X';
    std::cout << std::get<int>(t2) << '\n';
    std::cout << std::get<std::string>(t2) << '\n';
}
$ g++ -Wall -Werror -std=c++14 tuple_example.cpp
$ ./a.out
5
-89.34
abc
5
abc
18
abX
$
```

Notice that we compiled with the `-std=c++14` flag instead of the `-std=c++11` flag. We had to do this because `std::get` does not support access by type (as opposed to by index) until C++14.

The syntax for accessing an element may seem awkward, but by using a template parameter for the index, an out-of-range index can be checked at compile time. (The STL `std::array` type uses a similar technique.) Unfortunately, as is the case with anything in C++ where templates are involved, the error messages are awful, and even if you scroll to the top, it’s at first difficult to understand what the messages are telling you. See the below.

```
$ cat tuple_example.cpp
#include <iostream>
#include <tuple>

int main()
{
    std::tuple<int, bool, int, float> t1{5, true, 18, -89.34};
    std::cout << std::get<5>(t1) << '\n';
}
$ g++ -Wall -Werror -std=c++14 tuple_example.cpp
In file included from tuple_example.cpp:2:0:
/usr/include/c++/7/tuple: In instantiation of 'class std::tuple_element<1, std::tuple<> >':
/usr/include/c++/7/tuple:1279:12:   recursively required from 'class std::tuple_element<4, std::tuple<bool, int, float> >'
/usr/include/c++/7/tuple:1279:12:   required from 'class std::tuple_element<5, std::tuple<int, bool, int, float> >'
/usr/include/c++/7/utility:133:69:   required by substitution of 'template<long unsigned int __i, class _Tp> using __tuple_element_t = typename std::tuple_element::type [with long unsigned int __i = 5; _Tp = std::tuple<int, bool, int, float>]'
/usr/include/c++/7/tuple:1326:5:   required by substitution of 'template<long unsigned int __i, class ... _Elements> constexpr std::__tuple_element_t<__i, std::tuple<_Elements ...> >&& std::get(std::tuple<_Elements ...>&&) [with long unsigned int __i = 5; _Elements = {int, bool, int, float}]'
tuple_example.cpp:7:32:   required from here
/usr/include/c++/7/tuple:1297:7: error: static assertion failed: tuple index is in range
       static_assert(__i < tuple_size<tuple<>>::value,
       ^~~~~~~~~~~~~
tuple_example.cpp: In function 'int main()':
tuple_example.cpp:7:32: error: no matching function for call to 'get<5>(std::tuple<int, bool, int, float>&)'
     std::cout << std::get<5>(t1) << '\n';
                                ^
In file included from /usr/include/c++/7/tuple:38:0,
                 from tuple_example.cpp:2:
/usr/include/c++/7/utility:225:5: note: candidate: template<long unsigned int _Int, class _Tp1, class _Tp2> constexpr typename std::tuple_element<_Int, std::pair<_Tp1, _Tp2> >::type& std::get(std::pair<_Tp1, _Tp2>&)
     get(std::pair<_Tp1, _Tp2>& __in) noexcept
     ^~~



... (around 100 lines -- not joking -- removed here for readability) ...


/usr/include/c++/7/tuple:1349:5: note:   template argument deduction/substitution failed:
/usr/include/c++/7/tuple:1355:5: note: candidate: template<class _Tp, class ... _Types> constexpr _Tp&& std::get(std::tuple<_Elements ...>&&)
     get(tuple<_Types...>&& __t) noexcept
     ^~~
/usr/include/c++/7/tuple:1355:5: note:   template argument deduction/substitution failed:
/usr/include/c++/7/tuple:1361:5: note: candidate: template<class _Tp, class ... _Types> constexpr const _Tp& std::get(const std::tuple<_Elements ...>&)
     get(const tuple<_Types...>& __t) noexcept
     ^~~
/usr/include/c++/7/tuple:1361:5: note:   template argument deduction/substitution failed:
```

What the message at the very top is saying is that `std::tuple` uses `static_assert` to check if an index is out-of-range. Whereas `assert()` (from `<cassert>`) allows you to crash if a condition is false during runtime, `static_assert` allows you to automatically cause *compilation* to fail of a condition is false. `static_assert` only works with conditions that use values known during compile time, which applies to template parameters.

Note that you can use a combination of piping and the `head` command to see the top lines of the compiler error messages and throw out the rest, but this requires that we use `2>&1` because piping only redirects standard output, not standard error. This unfortunately removes the coloration in the above (which you can't see in this document but which you can see if you do the above in a terminal that supports coloring).

```
$ g++ -Wall -Werror -std=c++14 tuple_example.cpp 2>&1 | head -n 15
In file included from tuple_example.cpp:2:0:
/usr/include/c++/7/tuple: In instantiation of 'class std::tuple_element<1, std::tuple<> >':
/usr/include/c++/7/tuple:1279:12:   recursively required from 'class std::tuple_element<4, std::tuple<bool, int, float> >'
/usr/include/c++/7/tuple:1279:12:   required from 'class std::tuple_element<5, std::tuple<int, bool, int, float> >'
/usr/include/c++/7/utility:133:69:   required by substitution of 'template<long unsigned int __i, class _Tp> using __tuple_element_t = typename std::tuple_element::type [with long unsigned int __i = 5; _Tp
```

```
    = std::tuple<int, bool, int, float>]'
/usr/include/c++/7/tuple:1326:5:   required by substitution of 'template<long unsigned int __i, class ...
    _Elements> constexpr std::__tuple_element_t<__i, std::tuple<_Elements ...> >&& std::get(std::tuple<
    _Elements ...>&&) [with long unsigned int __i = 5; _Elements = {int, bool, int, float}]'
tuple_example.cpp:7:32:   required from here
/usr/include/c++/7/tuple:1297:7: error: static assertion failed: tuple index is in range
       static_assert(__i < tuple_size<tuple<>>::value,
       ^~~~~~~~~~~~~
tuple_example.cpp: In function 'int main()':
tuple_example.cpp:7:32: error: no matching function for call to 'get<5>(std::tuple<int, bool, int, float
    >&)'
     std::cout << std::get<5>(t1) << '\n';
                                ^
In file included from /usr/include/c++/7/tuple:38:0,
$
```

3.3 Brute-Force Algorithms

A brute-force algorithm is an algorithm that finds the *optimal solution* to a problem by checking *all* – or in some cases, *potentially* checking all – possible solutions and then picking the best solution (if any), or *a* best solution if there are multiple optimal solutions (i.e. a tie for the best solution). This may sound horribly inefficient, and that's because it *is* horribly inefficient. A brute-force algorithm should not be your first resort. However, there are situations in which a brute-force approach is useful, and it is thus worthwhile to know how to implement a brute-force approach.

Suppose that you had three items – A, B, and C – and that you wanted to iterate over all possible subsets of these items. Below is a list of those subsets, where – for example – $\emptyset$ means nothing is chosen, $\{C\}$ means that only C is chosen, $\{B, C\}$ means that B and C are chosen, etc. Notice that each subset of the three items corresponds to a bit string. For example, 011_2 corresponds to $\{B, C\}$ because we choose to let bit #0, the least significant bit, dictate the selection of C and bit #1 dictate the selection of B. Note that there is nothing significant about which bit corresponds to which item. We could have let bit #2, the most significant bit, dictate the selection of C instead of that of A.

Subset	Bitstring	Base 10 Value
$\emptyset$	000_2	0_{10}
$\{C\}$	001_2	1_{10}
$\{B\}$	010_2	2_{10}
$\{B, C\}$	011_2	3_{10}
$\{A\}$	100_2	4_{10}
$\{A, C\}$	101_2	5_{10}
$\{A, B\}$	110_2	6_{10}
$\{A, B, C\}$	111_2	7_{10}

With what we have established above, we can iterate over all possible subsets using a loop with a counter variable x that takes on the values 0 through $2^n - 1$, where n is the number of items. Within each iteration of the loop, we determine which bits of x are set in order to determine the corresponding subset.

3.3.1 Connection to ECS 122AB / Algorithms Analysis

As mentioned above, brute-force algorithms are inefficient. The runtime of a brute-force algorithm that uses the technique described above runs in $\Theta(2^n)$ time in the worst case, where n is the input size[1]. Brute-force algorithms are an acceptable approach to problems where no efficient approach is known to exist. An efficient approach is one that runs in, say, $\Theta(n^3)$ time or better, although the definition of "efficient" varies depending on details such as the expected size of n in the real world. Some may say that an efficient approach is one that runs in polynomial time, but I wouldn't personally consider an algorithm that runs in $\Theta(n^{100})$ time to be efficient...

In ECS 122B, depending on who teaches it, you may learn more about brute-force algorithms. The approach to brute-force algorithms that is described above only works for problems in which the solution can be found through *subset enumeration*, i.e. iterating over the power set / all possible subsets of a given set. Other kinds of problems may demand a brute-force approach that instead uses *permutation enumeration*. Permutation enumeration is not done through bit operations, so we do not discuss this further here.

[1]You may not have encountered big-Theta notation yet if you are taking the ECS 36 track and have not taken ECS 36C. Understanding big-Theta notation is not necessary for you to do this assignment.

4 Programming Problems

If you haven't already, you should make sure that you can comfortably access the CSIF computers (ideally both remotely and physically, just in case). A comparable Linux environment may be fine too, but ultimately, the CSIF is the reference environment. You should avoid causes of undefined behavior in your code (i.e. you should avoid things that make your code behave differently in one environment/context vs. another), such as uninitialized variables.

Hopefully, in at least one of your prerequisite courses, the instructor encouraged you to learn how to use a debugger, regardless of the programming language. There are many debuggers that you can use for C++. One that is already on the CSIF (and should be on or easily installable on any other Linux environment) is GDB (`gdb`). GDB is a command-line debugger, and some may not like that, so there is also DDD, a GUI version of GDB. Learning to use a debugger takes some time (just like learning to use the Linux command line), but once you are sufficiently experienced with a debugger, it becomes much easier to find many kinds of bugs.

4.1 File Organization

There are four files involved in this assignment:

- `base_converter.cpp`
- `print_signed_rep.cpp`
- `brute_force.cpp`
- `decode_float.cpp`

You are to create these C++ files.

`base_converter.cpp`, `print_signed_rep.cpp`, and `decode_float.cpp` – the files for parts #1, #2, and #5 – are C++ *programs*, so each of these must have a `main()` implementation. On the other hand, for `brute_force.cpp` – the file for parts #3 and #4 – you will be asked to write two functions. Your `brute_force.cpp` file *must not* have a `main()` implementation.

4.2 General Restrictions

You are NOT allowed to use `std::bitset` (static bitset) or `std::vector<bool>` (dynamic bitset) in this assignment.

4.3 Part #1: Base Converter

File: `base_converter.cpp`

In this part, you will write a *program* that converts an **unsigned** value from one base to another base. The program should prompt the user for the initial base and then the representation of the value in that base, and then the program should ask for the desired base and print the new representation (in that desired base) of the value that was given earlier.

Assume that both bases entered by the user will be between 2 and 16 (inclusive).

For bases above 10[2] (i.e. bases 11 through 16), digits for 10 and above will be needed. In this case, like with hexadecimal, the uppercase letters A through F will be used.

Update: You may assume that the maximum base-10 value that you will ever have to deal with is $2^{32} - 1$. For example, you will never be asked to convert $FFFFFFFFF_{16}$ (notice there are 9 F's, not 8) from base 16 to, say, base 13, since $FFFFFFFFF_{16}$ is $2^{36} - 1$, which is greater than $2^{32} - 1$.

You do not need to perform input validation.

Below are examples of how your program should behave.

```
$ g++ -Wall -Werror -std=c++14  base_converter.cpp -o base_converter
$ ./base_converter
Enter initial base: 2
Enter base-2 representation: 10110
Enter desired base: 10
Base-10 representation: 22
$ ./base_converter
Enter initial base: 10
Enter base-10 representation: 22
Enter desired base: 2
Base-2 representation: 10110
$ ./base_converter
Enter initial base: 16
Enter base-16 representation: 59
Enter desired base: 5
Base-5 representation: 324
```

[2] In the initial version of this document, I erroneously said 9 instead of 10 here.

```
$ ./base_converter
Enter initial base: 13
Enter base-13 representation: BB
Enter desired base: 10
Base-10 representation: 154
$ ./base_converter
Enter initial base: 10
Enter base-10 representation: 12090
Enter desired base: 16
Base-16 representation: 2F3A
$
```

4.4 Part #2: Signed Integer Representations

File: `print_signed_rep.cpp`

In this part, you will write a *program* that takes an integer as a command-line argument and prints the two's complement and signed magnitude representations of the integer. Assume that an integer is represented with 32 bits (as is the case for `int` on the CSIF, the reference environment) and that the given integer can be represented (with both representations) with 32 bits.

The program should print an error message *to standard error* (not standard output) *and return* 1 if the user provides the wrong number of command-line arguments.

Here are examples of how your program should behave on a Linux command line.

```
$ g++ -Wall -Werror -std=c++14  print_signed_rep.cpp -o print_signed_rep
$ ./print_signed_rep
Wrong number of command-line arguments.
$ ./print_signed_rep blah blah
Wrong number of command-line arguments.
$ echo $?
1
$ ./print_signed_rep 5
2's complement: 00000000000000000000000000000101
Signed magnitude: 00000000000000000000000000000101
$ echo $?
0
$ ./print_signed_rep -5
2's complement: 11111111111111111111111111111011
Signed magnitude: 10000000000000000000000000000101
$ ./print_signed_rep 255
2's complement: 00000000000000000000000011111111
Signed magnitude: 00000000000000000000000011111111
$ ./print_signed_rep 2147483647
2's complement: 01111111111111111111111111111111
Signed magnitude: 01111111111111111111111111111111
$ ./print_signed_rep -2147483647
2's complement: 10000000000000000000000000000001
Signed magnitude: 11111111111111111111111111111111
$ ./print_signed_rep 0
2's complement: 00000000000000000000000000000000
Signed magnitude: 00000000000000000000000000000000
$
```

4.5 Part #3: Brute Force Subset Sum

File: `brute_force.cpp`

In the subset sum problem, the inputs are the following:

- A set of items and their values.
- A target value.

In this problem, the goal is to find an optimal solution, where an optimal solution is defined as a selection of items whose total value equals the target value.

To solve this problem, you will implement `isSubsetSum()`, which is declared in `brute_force.hpp`. The first element of the return value (which is of type `std::tuple<bool, std::vector<unsigned>>`) should be `false` if there is no optimal solution, i.e. no selection of items whose combined weight equals the target value. In this case, the second element should just be an empty vector. If there *is* an optimal solution, then the first element of the return value should be `true`, and the second element should contain the indices of the items that make up the optimal solution. (The items' values are given in a vector, so by "indices", I mean the indices in relation to that vector.) Those indices should be in ascending order, i.e. the lowest index should come first.

Restrictions: **Your implementation must use a brute-force algorithm and must use the approach described above (in the earlier section) that involves bitstrings. Expect a *heavy* penalty if you violate this requirement.** I will manually verify fulfillment of this requirement after the deadline. I will design the autograder such that so long as you do a reasonable job in your brute-force implementations, the autograder won't time out. However, if you do a sloppy, needlessly inefficient job such that the autograder times out, then you'll have to improve your approach.

At least one item must be chosen. That is, the empty set (i.e. a selection of *no* items) can never be considered an optimal solution (i.e. when the target value is 0).

For any input used by the autograder, there will be at most one optimal solution.

You may assume that there will never be more than 16 items.

Below are examples of how your program should behave.

```
$ cat test_subsetSum.cpp
#include "brute_force.hpp"

#include <iostream>

int main()
{
    std::cout << std::boolalpha;
    std::vector<int> vals1{5, 4, 12, -3, 10};
    auto result = isSubsetSum(vals1, 6);
    std::cout << std::get<bool>(result) << '\n';
    auto& indices1 = std::get<std::vector<unsigned>>(result);
    for (auto index : indices1) std::cout << index << ' ';
    std::cout << '\n';
    std::cout << "===\n";
    result = isSubsetSum(vals1, -5);
    std::cout << std::get<0>(result) << '\n';
    std::cout << "===\n";
    std::vector<int> vals2{20, 15, 10};
    result = isSubsetSum(vals2, 45);
    auto& indices2 = std::get<1>(result);
    for (auto index : indices2) std::cout << index << ' ';
    std::cout << std::endl;
    std::cout << "===\n";
    std::vector<int> vals3{8, 17, 2, 9, 3, 4, 5, 20, 15, -5, -8, 6,
                           101, 200, 300, 400};
    result = isSubsetSum(vals3, 87);
    std::cout << std::get<0>(result) << '\n';
    auto& indices3 = std::get<1>(result);
    for (auto index : indices3) std::cout << index << ' ';
    std::cout << std::endl;
}
$ g++ -Wall -Werror -std=c++14  test_subsetSum.cpp brute_force.cpp -o test_subsetSum
$ ./test_subsetSum
true
0 1 3
===
false
===
0 1 2
===
true
0 1 3 4 5 6 7 8 11
$
```

4.6 Part #4: Brute Force Set Partition

File: `brute_force.cpp` (Yes, this is the same file as part #3. I made it this way because you will likely have some common helper functions that both parts #3 and #4 would benefit from.)

In the set partition problem, the input is a set of integer values. An optimal partition is a way of spitting the set into two sets such that the total value of the first set equals the total value of the other set. For example, if the set of values is $\{5, 9, -6, -3, 1\}$, then an optimal partition would be $\{5, -3, 1\}$ and $\{9, -6\}$, as the total value of each of these two sets is 3. Note that you can't throw away any elements while partitioning, i.e. $\{5, 1\}$ and $\{9, -6\}$ would not be a valid partition, because you can't throw away -3.

To solve this problem, you will implement `isSetPartionable`, which is declared in `brute_force.hpp`. The first element of the return value (which is of type `std::tuple<bool, std::vector<unsigned>, std::vector<unsigned>>`) should be `false` if there is no optimal partition. In this case, the second and third elements should be empty vectors. If there *is* an optimal solution, then the first element of the return value should be `true`, and the second and third elements should contain the indices of the first and second sets (that result from the partitioning) respectively. Within each vector, the indices should be in ascending order. The first vector should be the one that has 0 as an index.

Restrictions: Same restrictions as part #3 (see above).

The optimal partition must split the set into two sets. That is, you *cannot* say that the optimal partition is to have no partition at all, resulting in one of the two sets being the empty set.

For any input used by the autograder, there will be at most one optimal partition.

You may assume that there will never be more than 16 values.

Here are examples of how your program should behave.

```
$ cat test_setPartition.cpp
#include "brute_force.hpp"

#include <iostream>

int main()
{
    std::cout << std::boolalpha;
    std::vector<int> vals{-8, 1000, -988, -6, 2, -24};
    auto result = isSetPartionable(vals);
    auto& indices1 = std::get<1>(result);
    for (auto index : indices1) std::cout << index << ' ';
    std::cout << std::endl;
    auto& indices2 = std::get<2>(result);
    for (auto index : indices2) std::cout << index << ' ';
    std::cout << std::endl;
}
$ g++ -Wall -Werror -std=c++14  test_setPartition.cpp brute_force.cpp -o test_setPartition
$ ./test_setPartition
0 3 4
1 2 5
$
```

4.7 Part #5: Floating-Point Representations

File: `decode_float.cpp`

In this part, you will write a *program* that parses a bit string to determine the corresponding floating-point value, according to a given format. Write a program that takes as its only command-line argument the name of a file whose contents describe the floating-point format. This file will specify the order of the sign, exponent, and mantissa fields. This file will also specify the sizes (in bits) of the exponent and mantissa fields. Below are examples of such files (all of which are provided on Canvas). Just so that the order of the fields is clear: in the first example, the most significant bit is the sign, and the least significant 23 bits are the mantissa, and in the last example (`test_format2.txt`), the most significant 3 bits form the exponent (with the bias applied, of course), and the least significant bit is the sign.

```
$ cat format_single_precision.txt
sign
exp: 8
man: 23
$ cat test_format1.txt
man: 2
sign
exp: 4
$ cat test_format2.txt
exp: 3
man: 4
sign
$
```

Your program should prompt the user to enter the bit string. If the user enters a bit string with the wrong number of bits, then the program should print an error message (to standard output) and keep prompting the user until they enter an appropriate bit string. (This is the only input validation that needs to be done.) Once the user enters an acceptable bit string, the program should interpret the bit string as a floating-point value (according to the format outlined in the given file) and print out that value.

As will be the case with the floating-point formats we will discuss during the 01/12 lecture, the exponent is stored with a bias. That bias is *always* $2^{n-1} - 1$, where n is the number of bits used for the exponent. For example, if the exponent is *stored* as 00111_2 and if $n = 5$ bits are used for the exponent, then the bias is $2^{5-1} - 1 = 2^4 - 1 = 15$, so the exponent will be $00111_2 - 15_{10} = 7_{10} - 15_{10} = -8_{10}$; **the bias should be *subtracted* when you are converting from the bit pattern to the actual base 10 exponent**. Moreover, there will always be an implicit leading 1 with the mantissa.

Unless you mess with the precision used by `printf()` or `std::cout` (whichever you prefer) when printing, floating-point error should not be an issue. I will avoid autograder test cases that would cause issues with the default precision used in printing.

None of the autograder test cases use a floating-point representation that:

- In total, takes up more than 32 bits.
- Has more than 8 bits for the exponent field.

Below are examples of how your program should behave on a Linux command line. I have added explanations for the first two examples.

4.7.1 Example #1

```
$ cat test_format1.txt
man: 2
sign
exp: 4
$ g++ -Wall -Werror -std=c++14 decode_float.cpp -o decode_float
$ ./decode_float test_format1.txt
Enter bit string: 101010
Wrong number of bits.
Enter bit string: 10101010
Wrong number of bits.
Enter bit string: 1111000
Value: -3.5
```

Decoding 1111000_2: The mantissa is 11_2, the sign is negative, and the exponent is stored as 1000_2. Since the exponent is represented with $n = 4$ bits, the bias is $2^{n-1} - 1 = 2^{4-1} - 1 = 2^3 - 1 = 7_{10}$. Thus, the exponent is $1000_2 - 7_{10} = 8_{10} - 7_{10} = 1_{10}$. We then have $1.11_2 \cdot 2^1 = 11.1_2 = 2 + 1 + \frac{1}{2} = 3.5_{10}$.

4.7.2 Example #2

```
$ ./decode_float test_format1.txt
Enter bit string: 0000100
Value: 0.125
```

Decoding 0000100_2: The mantissa is 00_2, the sign is positive, and the exponent is stored as 0100_2. The bias is 7_{10} since we are still using `test_format1.txt`. Thus, the exponent is $0100_2 - 7_{10} = 4_{10} - 7_{10} = -3_{10}$. We then have $1.00_2 \cdot 2^{-3} = 0.00100_2 = \frac{1}{8} = 1.25_{10}$.

4.7.3 Other Examples

```
$ ./decode_float test_format2.txt
Enter bit string: 11111110
Value: 31
$ ./decode_float format_single_precision.txt
Enter bit string: 01000000011000000000000000000000
Value: 3.5
$
```

5 Grading Breakdown

As stated in the updated syllabus, this assignment is worth 6% of your final grade. Below is the approximate worth of each individual part of this assignment.

- Part #1: 1%
- Part #2: 1%
- Part #3: 1%
- Part #4: 1%
- Part #5: 2%

6 Autograder Details

You will submit the below four files to the autograder on Gradescope. **The names of the files that you submit MUST be correct.** Do not submit any other files. If you don't submit all of the files, then you'll only be able to pass some of the test cases. (For example, if you only submit `base_converter.cpp` and `print_signed_rep.cpp`, then you'll only be able to pass the test cases for parts #1 and #2.) This could be useful if you have not finished all five parts of the assignment. ***However, once you have finished all five parts of the assignment, you should be submitting all four files EACH TIME that you submit to the autograder,*** if you want them all to be graded.

- `base_converter.cpp`
- `print_signed_rep.cpp`
- `brute_force.cpp`
- `decode_float.cpp`

Once the deadline occurs, whatever score the autograder has for your active submission (your last submission, unless you manually change it to be a different previous submission[3]) is your *final* score (unless you are penalized for violating any restrictions mentioned in this document).

The reference environment is the CSIF. That means that you should check if your code compiles and behaves properly on the CSIF if you want an indication as to how it will do with the autograder. The autograder will not show you compiler error messages, but you will be able to see such messages if you compile on the CSIF.

The autograder will compile your code with the commands used in the examples for each part, so you should use those same compilation commands on a Linux command line, esp. on the CSIF.

Your output must match mine *exactly*.

6.1 Released Test Cases vs. Hidden Test Cases

You will not see the results of certain test cases until after the deadline; these are the hidden test cases, and Gradescope will not tell you whether you have gotten them correct or not until after the deadline. One purpose of this is to promote the idea that you should test if your own code works and not depend solely on the autograder to do it. Unfortunately, Gradescope will display a dash as your score until after the deadline, but you can still tell if you have passed all of the visible test cases if you see no test cases that are marked red for your submission.

6.1.1 Test Cases for Parts #1, #2, and #5

There are 10 cases for each of parts #1, #2, and #5. Approximately half of the test cases are hidden, meaning that you cannot see if you got those cases right or wrong until after the deadline. You can find the inputs for the visible test cases for those parts on Canvas.

6.1.2 Test Cases for Parts #3 and #4

You can find the inputs used in the visible test cases on Canvas. You can compile these files and run the test cases directly on your end. Each of these files takes the case number as a command-line argument, which I demonstrate below. You may especially find it useful to run a test case on your end if the autograder gives you an unhelpful error message[4].

```
$ g++ -Wall -Werror -std=c++14 part3_visible.cpp brute_force.cpp -o part3_visible
$ ./part3_visible 1
true
3
===
true
0 3 4
$
```

There are 8 cases for each of parts #3 and #4. For part #3, 4 are visible; for part #4, only 3 are visible.

UCDAVIS
COMPUTER SCIENCE

[3] If you have any questions about how to do this on Gradescope, don't hesitate to let me know. I think it should be pretty straightforward though.

[4] There are some Gradescope autograder messages that I don't get much control over, unfortunately.