ECS 122B: Programming Assignment #2

Instructor: Aaron Kaloti

Summer Session #2 2020

Contents

1 Changelog

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

- v.1: Initial version.
- v.2: When mentioning the dimensions of the DP table for parts #1 and #2, replaced any mention of n and W with $(n + 1)$ and $(W + 1)$, respectively.
- v.3: In red, added clarification about vertices' indices in part #4.
- v.4: Clarified that you will submit `report.md` to Gradescope and that `prog2.cpp` should not include a `main()` when you submit it.

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 Thursday, September 3. Gradescope will say 12:30 AM on Friday, September 4, due to the "grace period" (as described in the syllabus). *Rely on the grace period for extra time at your own risk.*

3 Grading Breakdown

Here is an approximate breakdown of the weight of this assignment. There will be no manual review of your code, but you will submit a small "report" (really a description of where to find certain key aspects of your code so that we don't have to dig through your code to find them). As stated in the updated syllabus, the plan is for this assignment to be worth 14% of your final grade, so I will make the assignment out of 140 points.

- Part #1: 35 points.
- Part #2: 35 points.
- Part #3: 35 points.
- Part #4: 35 points.

4 Autograder Details

Once the autograder is released, I will update this document to include important details here. Note that you should be able to verify the correctness of your functions without depending on the autograder.

The autograder will use `g++` on a Linux command line to compile your code. (Refer to the example Linux command line interaction for part #2 to see the flags used.) On your end, you can likely get away with not using `g++` or a Linux command line. However, just make sure that whatever you use is pretty good about telling you compiler error messages *and* warnings. You should make sure to avoid sources of undefined behavior, such as uninitialized variables.

Below is a list of files that you will submit to Gradescope. Do not compress or archive the files.

- `prog2.cpp`
  - This file should not contain a `main()` function when you submit it to Gradescope.
- `prog2.hpp`
- `report.md`

As explained below, you will not submit a makefile.

4.1 Reminder about Gradescope Active Submission

Once the deadline occurs, whatever submission is your **active submission** will be the one that dictates your final score. By default, your active submission will be your latest submission. However, you can change which submission is your active submission; I will talk about how to do so during one of the lectures, after the autograder is up.

5 Tasks

On Canvas, you are provided an initial version of `prog2.hpp`. You must create `prog2.cpp`.

You should be using C++11. If you are compiling on a Linux command line with `g++`, then make sure to use the `-std=c++11` flag.

5.1 Small "Report"

You must submit a small report containing certain details about some of the parts. These details are explicitly specified in blue in each part of the directions below, where applicable. This report is not worth points (at least, not directly). Rather, it is to help us identify certain parts of your code that show that you obeyed certain constraints of the problems, where applicable. If you do not submit this report, we will still look to see if you obeyed said constraints, but we will decrease the score that the autograder gave you. Although the report is not meant to be a true, detailed, exhaustive report, you should still make sure that the report's content can be understood; if it cannot be, we will ignore its content and penalize you as if you did not submit the report.

The name of the report should be `report.md`, meaning that it should be a Markdown file and not some Word document or MP3 file renamed with the wrong file extension.

Either in comments in your code or in your report, you should cite any resources used in the creation of your code. While it is acceptable, but not encouraged, to look up certain details about small portions of your code, if it seems as if you copied significant portions of your code from other sources instead of coming up with that code yourself, you may be given a zero on the assignment and reported to the OSSJA, even if you cited the source that you excessively copied from. For certain tasks such as subset enumeration or permutation enumeration, it may be more forgivable for your code to look similar to code that can be found online, given how formulaic such approaches are; cite your sources to be safe. Do not hesitate to ask for clarification on this, as the last thing that I want is for students to be reported to the OSSJA for cheating that they did not even know they did.

5.2 Coding Problems

5.2.1 Part #1: DP Optimization and Subset Sum

Function: `findOptSubsetSum()`.

Using the dynamic programming approach that we discussed in class, implement `findOptSubsetSum()` so that it determines the optimal *weight* in the given Subset Sum Problem instance. Recall that we define the Subset Sum Problem as follows: Given a set of n items, each with a nonnegative weight $w_i \forall 1 \leq i \leq n$, and given a nonnegative weight bound W, find $\sum\limits_{i \in S} w_i$ such that S is a subset of the items, $\sum\limits_{i \in S} w_i \leq W$, and $\sum\limits_{i \in S} w_i$ is maximized.

Below is an example of how your function should behave.

```
$ cat demo_part1.cpp
#include "prog2.hpp"
#include <cstdio>

int main()
{
    std::vector<unsigned> weights{5, 9, 1, 3, 6};
    printf("%u\n", findOptSubsetSum(8, weights));
}
$ make
g++ -Wall -Werror -std=c++11 -g -c prog2.cpp -o prog2.o
g++ -Wall -Werror -std=c++11 -g demo_part1.cpp prog2.o -o demo_part1
$ ./demo_part1
8
$
```

Constraint: For your DP array / DP table / memoization table (whatever term you want to use), you must use an $(n+1)$-by-$(W+1)$ (or $(W+1)$-by-$(n+1)$) table, where n is the number of items and W is the weight bound. In your report, you should briefly mention which variable corresponds to this table.

5.2.2 Part #2: Improving Auxiliary Space of Subset Sum

Function: `findOptSubsetSum2()`.

The auxiliary space of our approach in the previous part is $\Theta(nW)$, where n is the number of items and W is the weight bound. We can improve on this and achieve $\Theta(W)$ auxiliary space by taking advantage of the following observation about our recurrence $OPT(i, w)$ shown in the lecture slides: $\forall w$ such that $0 \leq w \leq W$, $OPT(i, w)$ does not depend on any $OPT(i-k, w\prime) \forall k \geq 2, 0 \leq w\prime \leq w$. In other words, we do not need to store all n rows (or n columns, depending on how you oriented it) of the table at once, *just the last row we calculated and the row we are currently calculating*, so long as we are careful about it.

Constraint: Your DP table must never grow bigger than 2 or 3 rows (or columns, depending on how you view it). You only need the row/column corresponding to the most recent item in order to calculate the next row/column; the other, unconstrained axis of the table should still go from indices 0 to W, where W is the weight bound. In your report, you should briefly describe how you ensure the table never gets larger than 2 or 3 rows/columns.

The third parameter, x, is meant to assess if you truly understand our space optimization here. x is always in the range $1 \leq x \leq n$. `findOptSubsetSum2()` should return the last *two* rows of the DP table, *after the first x items have been processed as per the recurrence.* For example, if $x = 3$, then your function should process the first three items with the DP algorithm and then return the last two rows (which should be the only two rows stored at that point) that were computed, i.e. the rows corresponding to the calculations for $i = 3$ and $i = 2$. The return value of `findOptSubsetSum2()` is thus a 2-by-$(W+1)$ array/`vector` of unsigned integers (unsigned because $OPT(i, w) \geq 0 \forall 1 \leq i \leq n, 1 \leq w \leq W$).

Below are examples of how your function should behave.

```
$ cat demo_part2.cpp
#include "prog2.hpp"
#include <cstdio>
#include <iostream>

int main()
{
    std::vector<unsigned> weights{5, 9, 1, 3, 6};
    printf("=== Example 1 ===\n");
    auto lastTwoRows = findOptSubsetSum2(8, weights, 3);
    for (const auto& row : lastTwoRows)
    {
        for (unsigned val : row)
            printf("%u ", val);
        std::cout << std::endl;  // Mixing printf() and std::cout just
                                 // for your reading enjoyment.
    }
    printf("=== Example 2 ===\n");
    lastTwoRows = findOptSubsetSum2(8, weights, 5);
    for (const auto& row : lastTwoRows)
    {
        for (unsigned val : row)
            printf("%u ", val);
        std::cout << std::endl;
    }
}
$ make
g++ -Wall -Werror -std=c++11 -g demo_part2.cpp prog2.o -o demo_part2
$ ./demo_part2
=== Example 1 ===
0 0 0 0 0 5 5 5 5
0 1 1 1 1 5 6 6 6
=== Example 2 ===
0 1 1 3 4 5 6 6 8
0 1 1 3 4 5 6 7 8
$
```

5.2.3 Part #3: Brute-Force Subset Sum

Function: `findOptSubsetSumBF()`.

For this problem, you will write a brute-force solver for the Subset Sum Problem. However, this time, we will use the following, different definition of the Subset Sum Problem: Given set $X = \{x_1, x_2, ..., x_n\}$ of integers, determine if there exists $S \subseteq X$ such that $|S| \neq 0$ and $\sum_{x_i \in S} x_i = 0$. If the answer to the given Subset Sum Problem instance is "yes", then your function return an instance of `std::tuple<bool, std::vector<unsigned>>` in which the first element is set to `true` and the second element is set to a vector of the *zero-based* indices of the items chosen in *a* optimal solution, in ascending order. If the answer is "no", then the returned instance of `std::tuple<bool, std::vector<unsigned>>` should have the first element set to `false` and the second element set to an empty vector.

For the report: You must explain how you implemented your brute-force approach. This might take around four to six sentences, and you should make references to variables in your code. The goal here is to convince us that you did solve this with a brute-force algorithm (and not, for example, with some approximation algorithm that you looked up online that just happens to get most of the test cases correct), preferably in a manner similar to the way in which we implemented brute-force algorithms during lecture.

Hint: You may find it helpful to review (or, if you have not taken ECS 50, briefly self-study) the following concepts. Note that I am not saying that you will need to use all of these concepts.

- Binary numbers.
- The bitwise OR operator.
- The bitwise AND operator (masks).
- The bitwise XOR operator.
- Bit shifting (left and right shift).

Below is an example of how your program should behave.

```
$ cat demo_part3.cpp
#include "prog2.hpp"

#include <iostream>
#include <tuple>

int main()
{
    std::vector<int> weights{-3, 8, -2, 5};
    auto output = findOptSubsetSumBF(weights);
    std::cout << "First: " << std::boolalpha << std::get<0>(output) << '\n';
    for (unsigned index : std::get<1>(output))
        std::cout << index << ' ';
    std::cout << std::endl;
}
$ make
g++ -Wall -Werror -std=c++11 -g demo_part3.cpp prog2.o -o demo_part3
$ ./demo_part3
```

```
First: true
0 2 3
$
```

5.2.4 Part #4: Brute-Force Independent Set

Function: `findMaxIndSetBF()`.

Write a brute-force solver for the max independent set problem. The function `findMaxIndSetBF()` should return *an* optimal independent set represented as a `vector` of unsigned integers in ascending order, where each unsigned integer is the index of a vertex chosen in that particular independent set. The input graph's information is stored in a file whose name is passed to `findMaxIndSetBf()`. The file, which will always be properly formatted, starts with a line containing the number of vertices. After that, the file contains a list of edges, with one edge per line and where each edge is indicated by two nonnegative integers (the indices of the vertices that are the endpoints of that edge). The graph is always undirected (and unweighted, although the independent set problem doesn't care about that), which means that an edge from v_2 to v_5 implies an edge from v_5 to v_2. To clarify, if the number of vertices is n, then the vertices' indices will always be in the range from 0 to $n-1$, e.g. you won't have a situation where $n = 4$ and the vertices' indices are 5, 7, 11, and 53.

For the report: You must explain how you implemented your brute-force approach. This might take around four to six sentences, and you should make references to variables in your code. The goal here is to convince us that you did solve this with a brute-force algorithm, preferably in a manner similar to the way in which we implemented brute-force algorithms during lecture.

Below is an example of how your program should behave.

```
$ cat graph_files/graph1.txt
4
0 1
1 2
1 3
$ cat demo_part4.cpp
#include "prog2.hpp"
#include <iostream>

int main()
{
    auto indices = findMaxIndSetBF("graph_files/graph1.txt");
    for (unsigned index : indices)
        std::cout << index << ' ';
    std::cout << std::endl;
$ make
g++ -Wall -Werror -std=c++11 -g demo_part4.cpp prog2.o -o demo_part4
$ ./demo_part4
0 2 3
$
```