CS211 Spring 2021
Programming Assignment II

David Menendez

Due: March 12, 2021, 10:00 PM
Hand in by March 13, 2021, 4:00 AM

1 Background: the knapsack problem

We have a set of items, each of which has a *weight* and a *value*. We want to pack these items into a knapsack, but there is a maximum amount of weight that the knapsack can carry. Our goal is to find the most valuable subset of the items whose total weight does not exceed the knapsack's weight limit.

For simplicity, we assume that the weights and values for each item are positive integers. We also assume that the knapsack can carry any combination of items, as long as the weight limit is not exceeded.

For example, we might have five light items with weight 1 and value 10, and one heavy item with weight 4 and value 39. If our knapsack's weight limit is 6, what is the maximum total value of the items we can place in the knapsack?

1.1 Formal definition

An instance of the knapsack problem involves a weight limit L and N items, numbered from 0 to $N - 1$. Each item i has a weight w_i and value v_i.

A solution to the knapsack problem is a set if items S that maximizes the total value

$$\sum_{i \in S} v_i \tag{1}$$

such that

$$\sum_{i \in S} w_i \leq L. \tag{2}$$

1.2 Non-optimal approaches

A naïve strategy is to test every combination of items and keep the combination with the highest total value of all combinations that do not exceed the weight limit. This strategy requires checking 2^N subsets o items, leaving it impractical for all but the smallest problems.

A greedy approach might sort the items by some metric, such as value or ratio of value to weight, and repeatedly choose the best item that does not exceed the remaining weight limit. Unfortunately, this approach is not guaranteed to find an optimal subset.

Going back to our example, if our weight limit is 5 and we choose the most valuable items first, we will select the heavy item and one light item, for a total value of 49. The optimal subset with total weight 5 is to take five light items, for a total value of 50.

If we choose items to maximize value per weight, we will find the optimal set for weight limit 5. If the weight limit is 6, we will chose five light items again, for a total value of 50. Here, the optimal solution is to choose two light items and the heavy item, for a total value of 59.

1.3 Method: Dynamic programming

We will use a *dynamic programming* approach to find optimal solutions. Dynamic programming finds a solution to a problem by first finding solutions to smaller sub-problems.

For our knapsack problem, we will write V_ℓ^n to indicate the maximum total value obtainable with items 0 through n subject to weight limit ℓ. Thus, V_L^{N-1} is the maximum value obtainable using all N items with weight limit L.

We will construct a table containing V_ℓ^n for all subproblems from V_1^0 to V_L^{N-1}. To simplify the definition, we define $V_\ell^n = 0$ if $n < 0$ or $\ell < 1$.

To compute V_ℓ^n, we must decide whether or not to include item n in the set. The maximum total value if we do not include item n is V_ℓ^{n-1}. If we do choose n, then whatever we chose from the preceding items must fit within the limit $\ell - w_n$. Thus, the maximum total value if we do include item n is $v_n + V_{\ell-w_n}^{n-1}$. We choose whichever is larger.

$$V_{n,\ell} = \begin{cases} 0 & \text{if } n < 0 \text{ or } \ell < 1 \\ \max\{V_\ell^{n-1}, v_n + V_{\ell-w_n}^{n-1}\} & \text{otherwise} \end{cases} \tag{3}$$

Note that we can fill this table by columns, from left to right, or by rows, from top to bottom. To compute cell V_ℓ^n, we only need to know the values of its neighbor to the left and to another value higher in the same column.

Table 1 shows the table we would compute for our running example with one heavy item and five light items. We see that $V_4^4 = 40$. This was computed by comparing V_4^3 and $10 + V_3^3$, since $v_4 = 10$ and $w_4 = 1$. Items 0 through 3 include the heavy item and three light items. If the weight limit is 3, the best solution is to take all three light items, thus $V_3^3 = 30$. If the weight limit is 4, the best solution is to take the heavy item, thus $V_4^3 = 39$. A solution for items 0 through 4 will either include item 4 or not. If it does not include item 4, then its maximum value will be the same as the maximum value for items 0 through 3 with the same weight limit. If it does include item 4, then the maximum value will be the value of item 4 plus the maximum value obtainable with the remaining weight. That is, the weight limit minus $w_4 = 1$. Hence the comparison of $V_4^3 = 39$ and $10 + V_3^3 = 40$. The larger is the best possible solution with items 0 through 4 and weight limit 4.

It is worth working through a few examples yourself until you are convinced that this method works. Table 2 shows a table that would be computed for a less uniform set of items.

1.4 Determining the optimum set

The table V_ℓ^n gives the maximum value obtainable with items 0 through n and weight limit ℓ, but does not directly indicate which items were selected. This information could be kept in a separate table indicating whether the optimum solution for a particular subproblem includes that item, based on which choice was made when computing V_ℓ^n.

Table 1: Data and computed maximum values for the running example

(a) Item weights and values

i	w_i	v_i
0	4	39
1	1	10
2	1	10
3	1	10
4	1	10
5	1	10

(b) Maximum total values

ℓ	V_ℓ^0	V_ℓ^1	V_ℓ^2	V_ℓ^3	V_ℓ^4	V_ℓ^5
1	0	10	10	10	10	10
2	0	10	20	20	20	20
3	0	10	20	30	30	30
4	**39**	39	39	39	40	40
5	39	**49**	49	49	49	50
6	39	49	**59**	**59**	**59**	**59**

Table 2: Data and computed maximum values for another example

(a) Item weights and values

i	w_i	v_i
0	5	20
1	2	10
2	3	15
3	1	6
4	4	17

(b) Maximum total values

ℓ	V_ℓ^0	V_ℓ^1	V_ℓ^2	V_ℓ^3	V_ℓ^4
0	**0**	0	0	0	0
1	0	0	0	6	6
2	0	**10**	10	10	10
3	0	10	15	16	16
4	0	10	15	21	21
5	20	20	**25**	25	25
6	20	20	25	**31**	31
7	20	30	30	31	33
8	20	30	35	36	38
9	20	30	35	41	42
10	20	30	45	45	**48**

Alternatively, the set of items can be found by examining the table. In general, if $V_\ell^n = V_\ell^{n-1}$, we can assume that item n was not included in the solution. We consider the items in reverse order, from $N-1$ to 0, with an initial weight limit of L. For each item n, we compare V_ℓ^n and V_ℓ^{n-1}. If they are the same, then item n is not included and we continue. If they are not the same, we reduce the remaining weight by w_n and continue.

Applying this method to table 1b, we conclude that the optimum subset of items 0 through 5 with weight limit 6 includes items 0, 1, and 2. Entries considered are shown in **bold**.

Similarly, for table 2, the optimum subset for weight limit 10 is items 1–4. Exercise: what are the optimum subsets with weight limits 9 and 11?[1]

Ambiguous cases If $V_\ell^{n-1} = v_n + V_{\ell-w_n}^{n-1}$, this means there are (at least) two subsets that have the maximum total value. In our running example, when $\ell = 1$, any of the five light items can be chosen to obtain the maximum total value 10. For this assignment, we resolve ambiguity by choosing the solution that does not include the current item. Thus, the set corresponding to V_1^5 would include item 1.

2 Program

You will write a program `knapsack` which chooses items from a manifest. `knapsack` takes two arguments: the weight limit and a manifest file. It chooses a selection of items from the manifest that maximizes the total value without exceeding the weight limit. Once the selection is made, `knapsack` will print the selected items, one per line, in the same order they appear in the manifest file. After the list, `knapsack` will print the total value and total weight of the selected items.

The weight limit is a positive integer written in decimal. (You may use `atoi` or some other function to covert the string to an integer.)

The manifest file is specified using a path that should be passed to `open` without interpretation.

Your implementation SHOULD check for the correct number of arguments and report an error if first argument is not a positive integer or if the file specified by the second argument cannot be opened. You will not be tested for these cases.

Usage

```
$ cat manifest.01.txt
6
Heavy  4 39
Light1 1 10
Light2 1 10
Light3 1 10
Light4 1 10
Light5 1 10
$ ./knapsack 5 manifest.01.txt
Light1
Light2
Light3
```

[1] For limit 9, it is items 1, 2, and 4. For limit 11, it is 0, 1, 2, and 3.

```
Light4
Light5
50 / 5
~/D/R/2/2/p/src $
$ ./knapsack 6 manifest.01.txt
Heavy
Light1
Light2
59 / 6
```

2.1 Manifest file format

The manifest file begins with an integer indicating the number of items in the manifest. This is followed by the item definitions. Each item is specified by giving a *name*, a *weight*, and a *value*. The name is a string of at most 31 non-whitespace characters. The weight and value are positive integers.

All integers will be given in decimal notation.

The format specifies only that names and numbers are separated by whitespace. It is not advised to assume that items will be given one per line.

Your program will only be given manifests with uniquely named items. It is not necessary to check the uniqueness of item names. Your program SHOULD confirm that all specified weights and values are positive.

There is no specified maximum number of items, but you may assume that the number of items will fit in a regular `int`, signed or unsigned.

Note that the `%s` format specifier does not allocate space, and may only be safely used when an upper limit is given: `%31s`.

2.2 Output format

`knapsack` prints the names of the selected items, one per line, in the same order they are given in the manifest file. Item names must be given exactly as in the file.

Immediately after the list of items, `knapsack` must print the total value and total weight of the selected items, separated by a slash (`/`). The numbers must be printed in decimal, without leading zeros, and must be separated from the slash by a single space character.

3 Grading

Your submission will be awarded up to 100 points, based on how many test cases your program completes successfully.

The auto-grader provided for students includes half of the test cases that will be used during grading. Thus, it will award up to 50 points.

Make sure that your program meets the specifications given, even if no test case explicitly checks it. It is advisable to perform additional tests of your own devising.

3.1 Academic integrity

You must submit your own work. You should not copy or even see code for this project written by anyone else, nor should you look at code written for other classes. We will be using state of the art plagiarism detectors. Projects which the detectors deem similar will be reported to the Office of Student Conduct.

Do not post your code on-line or anywhere publically readable. If another student copies your code and submits it, both of you will be reported.

Do not look for solutions to the knapsack problem on-line. Do not look at sample code. You must submit your own original work.

4 Submission

Your solution to the assignment will be submitted through Sakai. You will submit a Tar archive file containing the source code and makefiles for your project. Your archive should not include any compiled code or object files.

The remainder of this section describes the directory structure (section 4.1), the requirements for your makefiles (section 4.2), how to create the archive (section 4.3), and how to use the provided auto-grader (section 4.4).

4.1 Directory structure

Your project should be stored in a directory named `src`. Typically, you will provide a single C file named for the program. That is, the source code for the program `knapsack` would be a file `knapsack.c`.

This diagram shows the layout of a typical project:

```
src
 +- Makefile
 +- knapsack.c
```

If you are providing your own tests, as described in section 4.5, they will be in a directory named `tests` inside the program directory. For example,

```
src
 +- Makefile
 +- factor.c
 +- tests
     +- manifest.01.txt
     +- test.01.20.txt
...
```

4.2 Makefiles

We will use `make` to manage compilation. Each program directory will contain a file named `Makefile` that describes at least two targets. The first target must compile the program. An additional

target, `clean`, must delete any files created when compiling the program (typically just the compiled program).

The auto-grader script is distributed with an example makefile, which looks like this (note that an actual makefile must use tabs rather than spaces for indentation):

```
TARGET = knapsack
CC     = gcc
CFLAGS = -g -std=c99 -Wall -Wvla -Werror -fsanitize=address,undefined

$(TARGET): $(TARGET).c
        $(CC) $(CFLAGS) $^ -o $@

clean:
        rm -rf $(TARGET) *.o *.a *.dylib *.dSYM
```

It is simplest to copy this file into your `src` directory and rename it. This will ensure that you programs will be compiled with the recommended options.

It is further recommended that you use `make` to compile your programs, rather than invoking the compiler directly. This will ensure that your personal testing is performed with the same compiler settings as the auto-grader. The makefiles created in the build directory by the auto-grader refer to the makefiles you create in the source directory and therefore pick up any changes made.

You may add additional compiler options as you see fit, but you are advised to leave the compiler warnings, sanitizers, and debugger information (`-g`). The makefile shown here specifies the C99 standard, in order to allow C++-style `//` comments; you may change that to C90, if you prefer.

Compiler options The sample makefile uses the following compiler options, listed in the `CFLAGS` make variable:

-g Include debugger information, used by GDB and AddressSanitizer.

-std=c99 Require conformance with the 1999 C Standard. (Disable GCC extensions.)

-Wall Display most common warning messages.

-Wvla Warn when using variable-length arrays.

-Werror Promote all warnings to errors.

-fsanitize=address,undefined Include run-time checks provided by AddressSanitizer and UBSan. This will add code that detects many memory errors and guards against undefined behavior. (Note that these checks discover problems with your code. Disabling them will not make your code correct, even if it seems to execute correctly.)

4.3 Creating the archive

We will use `tar` to create the archive file. To create the archive, first ensure that your `src` directory contains only the source code and makefiles needed to compile your project. Any compiled programs, object files, or other additional files should be moved or removed.

Next, move to the directory containing `src` and execute this command:

```
pa2$ tar -vzcf pa2.tar src
```

`tar` will create a file `pa2.tar` that contains all files in the directory `src`. This file can now be submitted through Sakai.

To verify that the archive contains the necessary files, you can print a list of the files contained in the archive with this command:

```
pa2$ tar -tf pa2.tar
```

You should also use the auto-grader to confirm that your archive is correctly structured.

```
pa2$ ./grader.py -a pa2.tar
```

4.4 Using the auto-grader

We have provided a tool for checking the correctness of your project. The auto-grader will compile your programs and execute them several times with different arguments, comparing the results against the expected results.

Setup The auto-grader is distributed as an archive file `pa2-grader.tar`. To unpack the archive, move the archive to a directory and use this command:

```
$ tar -xf pa2-grader.tar
```

This will create a directory `pa2` containing the auto-grader itself, `grader.py`, a library `autograde.py`, and a directory of test cases `data`.

Do not modify any of the files provided by the auto-grader. Doing so may prevent the auto-grader from correctly assessing your program.

You may create your `src` directory inside `pa2`. If you prefer to create `src` outside the `pa2` directory, you will need to provide a path to `grader.py` when invoking the auto-grader (see below).

Usage While in the same directory as `grader.py` and `src`, use this command:

```
pa2$ ./grader.py
```

The auto-grader will compile and execute the programs in the directory `src`, assuming `src` has the structure described in section 4.1.

To stop the auto-grader after the first failed test case, use the `--stop` or `-1` option.

To obtain usage information, use the `-h` option.

Program output By default, the auto-grader will not print the output from your program, except for lines that are incorrect. To see all program output for unsuccessful tests, use the `--verbose` or `-v` option:

```
pa2$ ./grader.py -v
```

To see program output for all tests, use `-vv`. To see no program output, use `--quiet` or `-q`.

Checking your archive We recommend that you use the auto-grader to check an archive before submitting. To do this, use the `--archive` or `-a` option with the archive file name. For example,

```
pa2$ ./grader.py -a pa2.tar
```

This will unpack the archive into a temporary directory, grade the programs, and then delete the temporary directory.

Specifying source directory If your `src` directory is not located in the same directory as `grader.py`, you may specify it using the `--src` or `-s` option. For example,

```
pa2$ ./grader.py -s ../path/to/src
```

Refreshing the build directory In the unlikely event that your build directory has become corrupt or otherwise unusable, you can simply delete it using `rm -r build`. Alternatively, the `--fresh` or `-f` option will delete and recreate the build directory before testing.

4.5 Providing your own tests

The tests provided with the auto-grader may not check every possible input scenario. If you wish to be certain that your program is correct, you should design and test additional test cases. This may be done by entering the cases manually and running the program yourself, or by using the user test capability of the auto-grader.

To provide additional tests for the auto-grader, create a directory named `tests` inside the directory containing your souce code and make file (see section 4.1).

First, create one or more manifest files, with names following the pattern `manifest.`X`.txt`, where X is a unique identifier not containing a period.

Next, create one or more reference files. The reference file will contain the expected output of `knapsack` for a particular weight limit and manifest file, which is indicated by the reference file name. A reference file for weight limit L and manifest X should be named `ref.`X`.`L`.txt`.

For example, `ref.abc.13.txt` contains the expected output for `knapsack` given width `13` and manifest file `manifest.abc.txt`.

User tests are not required. The capacity for user tests is provided to assist you in development, but it will not directly affect your grade.