Goals of the project

- Solve the knapsack problem using the traditional and "memory function" approach of dynamic programming.
    - Compare and contrast with a space-efficient variant where the values are not stored in a 2D table but in a hashtable
- Solve the knapsack problem using the "greedy" approach.
    - Compare and contrast with a heap-based implementation of the "greedy" approach.

Input

Given **n** items and corresponding value vector $v_1, v_2, ..., v_n$ and the weight vector $w_1, w_2, ..., w_n$. Let W = capacity of the knapsack.

- All of these will be read from text files with prefix *p0i*:
    - *p0i_c.txt* contains the capacity, *p0i_v.txt* contains ***n*** values and *p0i_w.txt* contains ***n*** weights.
    - Note: Files *p00_c.txt, p00_v.txt and p00_w.txt* contain data for the problem that we solved in class.

Some files to test your program can be downloaded from here containing data for prefixes *p00* - *p08*. After submission, your code will be tested for prefixes *p09* and *p10*.

Task 1a: Dynamic Programming (traditional) approach

- Implement a dynamic programming based algorithm to determine the optimal value of the knapsack and the set of items that yield this optimal value.
    - This requires filling all the cells in the (***n*** x ***W***) table
- Display the optimal value F(n, W), subset yielding the optimal value, and the number of basic operations (for finding both the optimal value and optimal subset). Note, items begin with index 1 and so the set of all items = {1, 2, 3, ..., n}. **(see sample output below)**

Task 1b: Memory function based Dynamic Programming approach

- Implement dynamic programming based on the memory function (page 295 of the textbook) to determine the optimal value of the knapsack and the set of items that yield this optimal value.
    - This requires filling a subset of cells in the (***n*** x ***W***) table
- Display the optimal value F(n, W), subset yielding the optimal value, and the number of basic operations (for finding both the optimal value and optimal subset). Note, items begin with index 1 and so the set of all items = {1, 2, 3, ..., n}. **(see sample output below)**

Task 1c: Space-efficient approach using hash tables

Even though the approach in Task 1b does not fill all cells of the n x W table, it requires O(nW) space to allocate the 2D table. For sufficiently large values of nW (say ~ 1 billion), your computer may run out of memory. For this task, we will use an alternative method that will use less space but may use more time to find the optimal values.

- Implement dynamic programming based on the memory function (page 295 of the textbook) to determine the optimal value of the knapsack and the set of items that yield this optimal value.
- Instead of storing the values of F(i, j) in an (***n*** x ***W***) table, use an open hash table with ***k*** cells and a hash function h(***i***, ***j***) that depends on both ***i*** and ***j***.
    - Resolve collisions using open hashing, where the keys (i, j) and value F(i, j) are stored in a linear linked list.
- Choose h(i, j) in the following way:
    - Create a mapping from the pair (i, j) to an integer *(i-1)*W + j* (where 1 <= j <= W , and 1 <= i <= n)
    - Take a modulus with ***k*** *(so that hash value fits in a hash table with k cells)*
    - Taking the two points together, *h(i , j) = [(i-1)*W + j] mod* **k.**
- Strategically determine the values of ***k*** so that collisions are minimized but the space used (size of the entire hash table) is significantly smaller than ***nW***. Use the test data ***p08*** for this purpose (because the capacity **W** in other data files is too small to make a noticeable difference).
- Display the optimal value, subset yielding the optimal value, and the number of basic operations (for finding both the optimal value and optimal subset). Note, items begin with index 1 and so the set of all items = {1, 2, 3, ..., n}

Task 2a: Greedy approach using sorting

- Implement a *greedy* algorithm that arranges the items in the decreasing order of value to weight ratio (v_i/w_i for i = 1, 2, ..., n), then select the items in this order until the weight of the next item exceeds the remaining capacity (Note: In this greedy version, **we stop right after the first item** whose inclusion would exceed the knapsack capacity).
    - To sort the items, implement a Theta(n log n) sorting method of your choice (Do **not** use inbuilt sort as we need to compute the total number of basic operations).
- Display the best value, subset yielding the best value, and the total number of basic operations (includes operations for sorting, finding the best value, and the corresponding subset). Note, items begin with index 1 and so the set of all items = {1, 2, 3, ..., n}. **(see sample output below)**

Task 2b: Greedy approach using max-heap

- Implement the *greedy* algorithm based on a max-heap that supports the operation *deletemax*.
    - The idea is to use the O(n) algorithm (**bottom-up approach**) to build the heap containing the n keys (v_i/w_i for i = 1, 2, ..., n) then perform a series of *deletemax*.
    - If the number of objects that are selected by the greedy algorithm is *k*, then the total complexity is O(n + *k* log n) which could be better than O(n log n) in some cases (the complexity of best sorting algorithm).
- Display the best value, subset yielding the best value, and the total number of basic operations (includes operations for building the heap, finding the best value, and the corresponding subset). Note, items begin with index 1 and so the set of all items = {1, 2, 3, ..., n}

Task 3: Comparison

- Task 1a vs Task 1b: Produce a graph that compares **the total number of basic operations** taken by the traditional and "memory-function" approaches (1a and 1b) **for all the input cases**. Note: the x-axis is **case-id** and the y-axis is the number of operations.
    - Comment on which approach is more efficient and why?

- Task 1b vs Task 1c: Discuss how the value of **k** was determined such that collisions are minimized but the space used is significantly smaller than ***nW***?
    - Show this in the form of a graph that compares the time (number of basic Ops) and the space taken by approach 1b and approach 1c for different values of **k**. Note: x-axis is space and y-axis is time (number of basic ops).
    - Comment on the advantages/disadvantages of approach 1b vs 1c.
- Task 2: Produce a graph that compares **the total number of basic operations** taken by greedy approaches (2a and 2b) **for all the input cases**. Note: x-axis is **case-id** and the y-axis is number of operations.
    - Comment on which approach is more efficient and why?