# COMP26120
# Academic Session: 2022-23

# Lab Exercise 5: The 0/1 Knapsack Problem

Duration: 3 weeks

You should do all your work in the lab5 directory of the COMP26120_2022 repository - see Blackboard for further details. You will need to make use of the existing code in the branch as a starting point.

**Important:** You submit this lab via a quiz on Blackboard. This will:

1. Ask you some questions about your implementation, including the **hash and tag** of the commit you want us to mark (see below for details of this).
2. Ask you to upload a zip file of the python, c or java folder you have been working in.
3. Ask you to upload PDF reports of the experiments you will run in Part 3 of this lab (one report for each experiment)

You can save your answers and submit later so we recommend filling in the questions for each part as you complete it, rather than entering everything at once at the end.

**NB:** We have made some changes to this lab since the start of semester 1 in the hopes it will be quicker to mark. If you want a helper script for generating input for experiments and a LaTeX template for the report that reflects how it should be structured now, please **pull from upstream**:

You can do this by typing the following commands in your gitlab directory:

```
git remote remove upstream
git remote add upstream https://gitlab.cs.man.ac.uk/t95229ld/comp26120_2022_base.git
git fetch upstream
git merge upstream/master
```

## Code Submission Details

You have the choice to complete the lab in C, Java or Python. Program stubs for this exercise exist for each language. ***Only one*** language solution will be marked.

Because people had a number of issues with GitLab last year we are going to take a multiple redundancy approach to submission of code. This involves both pushing and tagging a commit to GitLab and uploading a zip of your code to Blackboard. By preference we will mark the code you submitted to GitLab but if we can't find it or it doesn't check out properly then we will look in the zip file on Blackboard. Please **do both** to maximise the chance that one of them will work.

When you submit the assignment through Blackboard you will be asked for the *hash* and *tag* of the commit you want marked. This is to make sure the TAs can identify exactly which GitLab commit

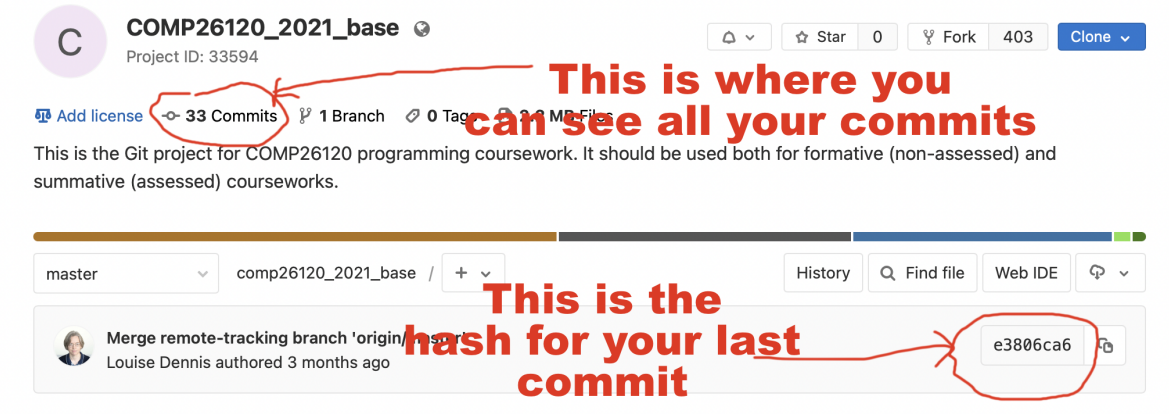


Figure 1: Identifying the hash of your most recent commit in GitLab

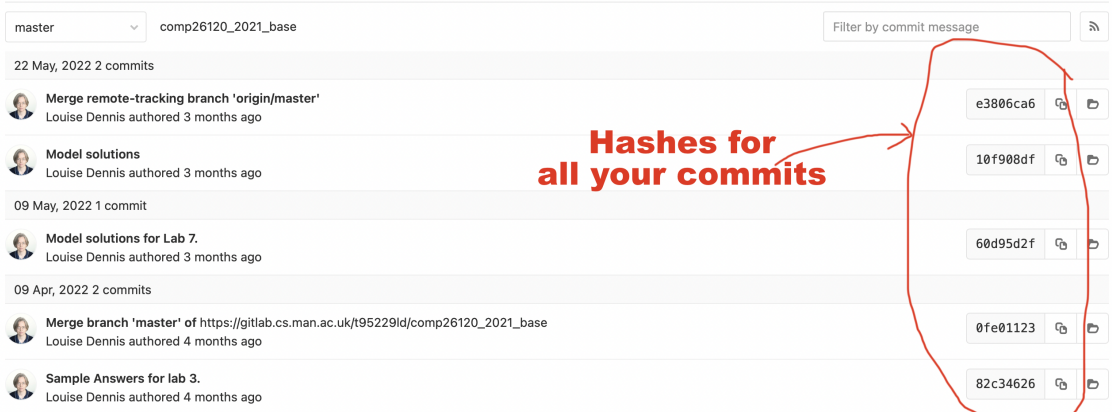


Figure 2: Identifying the hashes of previous commits in GitLab

you want marked. You tag a commit **lab5_solution** (we recommend you use this tag, but you do not have to) by typing the following at the command line:

```
git tag lab5_solution
git push
git push origin lab5_solution
```

You can find the hash of your most recent commit by looking in your repository on GitLab as shown in figure 1.
You can also find the hash for a previous commit by clicking on the "commits" link and then identifying the commit you are interested in. This is shown in figure 2.

Note that while the full hash for commits are quite long, we only need the first 8 characters (as shown in the screenshots) to identify for marking.

**Reminder:** It is bad practice to include automatically generated files in source control (e.g. your git repositories). This applies to object files (C), class files (Java), and compiled bytecode files (Python). It's not fatal if you do this by mistake, but it can sometimes cause confusions while marking.

While it is fine to discuss this coursework with your friends and compare notes, the work submitted **should be your own**. In particular this means you should not have copied any of the source code, or the report. We will be using the turnitin tool to compare reports for similarities.

# Learning Objectives

By the end of this lab you should be able to:

- To explain the 0/1 Knapsack problem and Fractional Knapsack problem.
- To implement a number of exact techniques for solving the 0/1 Knapsack problem.
- To implement one inexact technique - or heuristic - for finding good but not necessarily optimal solutions to the 0/1 Knapsack problem.
- To evaluate and compare running times of these techniques.
- To devise experiments to investigate factors that make Knapsack problems hard for various solution techniques.

# Introduction

In this section we introduce two related 'Knapsack' problems.

## The 0/1 Knapsack Problem and Logistics

Suppose an airline cargo company has 1 aeroplane which it flies from the UK to the US on a daily basis to transport some cargo. In advance of a flight, it receives bids for deliveries from (many) customers.

Customers state

- the weight of the cargo item they would like to be delivered;
- the amount they are prepared to pay.

The company must choose a subset of the packages (bids) to carry in order to make the maximum possible profit, given the total weight limit that the plane is allowed to carry.

In mathematical form the problem is: Given a set of $N$ items each with weight $w_i$ and value $v_i$, for $i = 1$ to $N$, choose a subset of items (e.g. to carry in a knapsack, or in this case an aeroplane) so that the total value carried is **maximised**, and the total weight carried is less than or equal to a given carrying capacity, $C$. As we are maximising a value given some constraints this is an *optimisation* problem.

This kind of problem is known as a **0/1 Knapsack problem**. A Knapsack problem is any problem that involves packing things into limited space or a limited weight capacity. The problem above is "0/1" because we either do carry an item: "1"; or we don't: "0". Other problems allow that we can take more than 1 or less than 1 (a fraction) of an item. Below is a description of a fractional problem.

See the description in *Algorithm Design and Applications*, p. 498. or the briefer description in *Introduction to Algorithms*, p. 417.

### An Enumeration Method for solving 0/1 Knapsack

A straightforward method for solving any 0/1 Knapsack problem is to try out all possible ways of packing/leaving out the items. We can then choose the most valuable packing that is within the weight limit.

For example, consider the following knapsack problem instance:

| Sample Input |
|---|
| 3<br>1 5 4<br>2 12 10<br>3 8 5<br>11 |

The first line gives the number of items; the last line gives the capacity of the knapsack; the remaining lines give the index, value and weight of each item e.g. item 2 has value 12 and weight 10.

The full enumeration of possible packings would be as follows:

| **Items Packed** | **Value** | **Weight** | **Feasible?** | |
|---|---|---|---|---|
| 000 | 0 | 0 | Yes | |
| 001 | 8 | 5 | Yes | |
| 010 | 12 | 10 | Yes | |
| 011 | 20 | 15 | No | |
| 100 | 5 | 4 | Yes | |
| 101 | 13 | 9 | Yes | OPTIMAL |
| 110 | 17 | 14 | No | |
| 111 | 25 | 19 | No | |

The items packed column represents the packings as a binary string, where "1" in position $i$ means pack item $i$, and 0 means do not pack it. Every combination of 0s and 1s has been tried. The one which is best is 101 (take items 1 and 3), which has weight 9 (so less than $C = 11$) and value 13. We can also represent a solution as an array of booleans (this approach is taken in the Java and Python stubs).

### Some vocabulary

- **A solution:** Any binary or boolean array of length $N$ is referred to as a packing or a solution; This only means it is a correctly formatted instruction of what items to pack.
- **A feasible solution:** A solution that also has weight less than the capacity $C$ of the knapsack.
- **An optimal solution:** The best possible feasible solution (in terms of value).
- **An approximate solution:** Only a high value solution, but not necessarily optimal.

In this lab we will investigate some efficient ways of finding optimal solutions and approximate solutions.

## Description

This lab asks you to implement four different solutions to the 1/0 Knapsack problem over three weeks. We have provided partial solutions and it is your job to complete them.

**Important Note 1:** The C support code represents Knapsack solutions as a bitstring (an array of 0s or 1s) indicating whether an items should (1) or should not (0) be packed into the knapsack. The Java and Python code represent solutions as arrays of booleans (True and False). The Java and Python support code, converts the boolean values to 0 or 1 for printing so there can be a uniform presentation of results. In what follows we will sometimes use T as shorthand for True and F as shorthand for False.

**Important Note 2:** In the input files, items for the Knapsack are numbered from 1 to N. The support functions read these into arrays of size N+1 (one for item weights, one for item values and one to map the index of the item in the input file to that in a sorted array (for the algorithms where sorting by value/weight ratios is useful)). These arrays all have a null or None value (depending on language) as the 0th element of the array. In this way the array indices match up with the numbering in the input files, but it does mean that functions and methods working with these arrays have to account for the irrelevant 0th element. This is easier in C where you can simply pass a pointer to the element of the array at position 1 than in Python or Java where you need to start any iterations etc., explicitly at element 1.

## Task 1a: Full Enumeration

**Time Budget:** Task 1a is primarily intended to help you familiarise yourself with the input files and running the knapsack program. You should not spend more than an hour on this task and it is not needed for later tasks in this coursework. If you are stuck on this for some reason but are confident that you understand how our input files work, how problems are represented internally in the program and how the program can be run, then we would recommend you move on to the rest of this coursework after an hour.

We have provided an implementation of the enumeration method for the Knapsack problem in each language. You can (compile, if necessary, and) run this program on `data/easy.20.txt`. The program enumerates the value, weight and feasibility of every solution and prints them to the screen. However, it does not "remember" the best (highest value) feasible solution or display it at the end.

1. Adapt the code so that it does that. NB. on `data/easy.20.txt` this should compute a solution value of 377.

2. It would also be useful to display how much of the enumeration has been done – like a progress bar. Add code to the enumeration loop to print out the fraction of the enumeration that is complete and the value of the current best solution. Note: If you update the progress bar too often it will slow you down a lot. You may want to think about how often you update it if you want a more efficient program.

You may change the value of the QUIET variable to suppress output to the screen and make the code run faster.

### Running on our Example Inputs

In the `data` directory you will find four files:

easy.20.txt
easy.200.txt
hard.200.txt
hard.2000.txt

python3 enum_kp.py ../data/easy.20.txt

In each case the number in the file name (20, 200, 2000) indicates how many items the example wants you to put into the knapsack. Your enumeration algorithm will probably only manage to solve the 20 item knapsack. The correct optimal answers for the problems are 377 (easy.20.txt), 4077 (easy.200.txt), 126968 (hard.200.txt), 1205259 (hard.2000.txt).

### Blackboard Submission

Once you have completed this implementation you should look at the Blackboard submission form where you will be asked the following question about Part 1a.

1. When you ran your enumeration solution on `data/easy.20.txt` (no more than 100 words):
   (a) How long did it take to run? You can use the Unix `time` utility for this. It is fine to just report the `real` time from this function.
   (b) What did value did it report as the maximal possible knapsack value?
   (c) What was the difference between the reported value and the optimal value (377)? (NB. The answer to this will be 0 if your implementation returned the optimal solution).

   For instance, if your code took 20.34 seconds to run and reported a maximal possible knapsack value of 374. I would expect to see here something like:

   ```
   easy.20.txt: 20.34s, reported value: 374, difference to optimal:3
   ```

## Task 1b: Dynamic Programming

**Time Budget:** Task 1b is the key task in this coursework and you will need it working for your report. The intention is that task 1b should take you 1-2 hours to complete. Dynamic programming can sometimes be a bit fiddly so you might want to budget this time to include one of the drop-in sessions so you can access TA help if necessary. If you are going over 3 hours it might be worth "borrowing" some of the time allowed for later parts of the coursework, but you might want to look ahead, check out the marking scheme, and think about where you can best spend time.

Unlike the enumeration approach (which generated all possible solutions and picks the best), dynamic programming approaches iteratively compute the solution to larger and larger subproblems using the results of smaller subproblems until we have the solution to the overall problem.

Complete the program that solves the 0/1 Knapsack Problem by dynamic programming. Using the program stubs and support files we provide for you.

### The Dynamic Programming Solution

There is a detailed explanation of the dynamic programming approach to the 0/1 Knapsack Problem with examples on p. 343-345 of *Algorithm Design and Applications*. In brief the solution is as follows:

**Identify sub-problems in a tabular form** We will use a two dimensional array, $V$, for our sub-problems. $V[i][w]$ is the maximum value we can achieve with the first '$i$' items in our input list using at most weight '$w$'. If we have a list of $N$ items and a capacity of $C$, then if we can compute all values in the array $V$ the value at $V[N][C]$ will contain the optimal value of the items can can fit into our aeroplane.

**Initialise the array** If we have no items then our optimal value is 0, so $V[0][w] = 0$ for all $0 \leq w \leq C$. All other values in the array we initialise with null or None (depending upon the language).

**Recursive Step** The maximum value we can make with the first $i$ items and a weight limit of $w$, where the value of the $i$th item is $v_i$ and the weight of the ith item is $w_i$ is either:

1. The maximum value we could make with the first $i-1$ items (ignoring the $i$th item), or
2. The maximum value we could make by adding the value of the $i$th item, $v_i$, to the maximum value we could make using the first $i-1$ items and a maximum weight of $w - w_i$.

This is $V[i][w] = max(V[i-1][w], v_i + V[i-1][w-w_i])$ for $1 \leq i \leq N, 0 \leq w \leq C$.

The table $V$ just tells us the best value we can get after considering $i$ items it doesn't tell us which ones we added. We can update the algorithm to keep track of this information by using an auxilliary array, *keep* where $keep[i][w]$ records whether the $i$th item is used in the maximal solution for $V[i][w]$. If $keep[i][w] = 0$ then we know that the $i$th item has been ignored and this maximal solution has been constructed from the maximal solution for $V[i-1][w]$ so we should use $keep[i-1][w]$ to find out which other items are included. If $keep[i][w] = 1$ then we know that the $i$th item has been included and this maximal solution has been constructed from the maximal solution for $V[i-1][w-w_i]$ so we should use $keep[i-1][w-w_i]$ to find out which other items are included.

The following pseudo-code outlines the complete solution

```
KnapSack(v, w, N, C) {
   for (w = 0 to C) V[0][w] = 0
   for (i = 1 to N)
      for (w = 0 to C)
         if (w[i] <= w) and (v[i] + V[i - 1][w - w[i]] > V[i - 1, w]) {
            V[i][w] = v[i] + V[i - 1][w - w[i]]
            keep[i][w] = 1
         } else {
            V[i][w] = V[i - 1][w]
            keep[i][w] = 0
        }
   K = C
   for (i = N downto 1)
      if (keep[i][K] == 1) {
         output i
         K = K - w[i]
      }
   return V[N, C]
```

You may want to run a few simple examples of the algorithm by hand to check you understand how it works. For instance, how does it behave with the following four items:

| $i$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $v_i$ | 10 | 40 | 30 | 50 |
| $w_i$ | 5 | 4 | 6 | 3 |

### Implementing and Testing the Dynamic Programming Solution

You can find further instructions for approach to this in *dp.c/dp_kp.java/dp_kp.py* file (depending upon which language you are using). If you wish, you may ignore these program stubs and implement your own dynamic programming approach to the 0/1 Knapsack problem.

Test your code on the instance file, `data/easy.20.txt`. You should get the same answer as with the enum program.

Try the harder problems in the `data` directory. If interested, and you have time, you might want to investigate the space and time complexity of your solution.

Once you have completed this implementation you should look at the Blackboard submission form where you will be asked the following questions about Part 1b.

### Blackboard Submission

Once you have completed this implementation you should look at the Blackboard submission form where you will be asked the following question about Part 1b.

1. Run your dynamic programming solution on the four files in the data directory, killing it if it runs for more than 60 seconds. Then answer the questions below **for each of the four files** (no more than 100 words in total):

   (a) Did the run complete in under 60s? If so, how long did it take to run? You can use the Unix `time` utility for this. It is fine to just report the `real` time from this function.

   (b) Did it report a value as a possible knapsack value when it finished running or you killed it? If so what was that value?

   (c) If it reported a value, what was the difference between this and optimal value for that problem (See the end of Task 1a for a list of the optimal values)?

   I would expect to see here something like:

   ```
   easy.20.txt: 20.34s, reported value: 374, difference to optimal:3
   easy.200.txt: 35.5s, reported value: 4077, difference to optimal:0
   hard.200.txt: killed, reported value: none, difference to optimal: not applicable
   hard.2000.txt: killed, reported value: none, difference to optimal: not applicable
   ```

2. How does your implementation of dynamic programming work. Illustrate this with the part of your code that is equivalent to lines 3-11 of the pseudocode we have given above. In particular, explain in your own words, the role of your equivalent of the arrays `V` and `keep` in your implementation. (no more than 300 words, not including code snippet)

**You should be here by the end of week 1 of this lab.**

---

## Task 2a: Fractional Knapsack Bound

**Time Budget:** Tasks 2a and 2b, taken together, are intended to take between 1 and 2 hours to complete. They are not needed for any other part of this lab. If you have spent more than two hours on Part 2 we recommend proceeding to tasks 3a and 3b.

Imagine we have decided we definitely want to pack some particular items, and definitely don't want to pack some particular other ones. For the remaining items we are not sure yet. We can represent this situation as

001110*****  or  FFTTTF*****

where 0 or F (False) means definitely won't take the item, 1 or T (True) means definitely will, and * means don't know. We call these **partial solutions**.

We can now calculate an estimate of the best value of all possible ways of replacing the *s by 0/Fs and 1/Ts. This will be an overestimate. We call it an upper bound.

To calculate the estimate we take the * items in decreasing order of value-to-weight ratio and add them to the knapsack, until we go over the capacity. For the last item added, we take it out again and only add that fraction of its weight that would fit, and adding the same fraction of its value to the total value of the knapsack. This is a kind of cheating; however, we are only interested in making an estimate.

In the *bnb.c/bnb_kp.java/bnb_kp.py* code, we have already provided an almost complete function *frac_bound*() which accepts a partial solution of the form 01101****** (C) or FTTFT****** (Java/Python) as input and does the following things:

1. Checks the feasibility of the partial solution and sets the solution value to -1 if it is infeasible, and returns.
2. If the partial solution is feasible then its value is calculated, i.e. the value of the items already packed, and the value is updated
3. If the partial solution is feasible then its upper bound is also computed and updated.

Make sure you understand this *frac_bound*() function and complete it by filling in the two missing lines.

## Task 2b: Branch-and-Bound

The branch and bound approach was covered in lectures as a backtracking method for optimisation problems. You should also see pages 521-524 of *Algorithm Design and Applications*.

Let's consider how we might, as people, solve the 0/1 knapsack problem. We would try the 'best' item first (e.g. the one with the highest value-to-weight ratio) and see if we can fit the best items in. This is also a reasonable approach for a computer e.g. to sort the items in descending order of value-to-weight ratio, and add items in that order until the knapsack is full. However, we might realise that putting that really big item in the bag means there is left over space and putting two smaller items in would have been better i.e. we **back-track** by removing an item and trying something else. We get computers to do the same thing when systematically exploring the search space.

In order to prevent us backtracking through every possible solution, we can "prune" off parts of the set of the solutions we know cannot contain a feasible/optimal solution. If we know that a particular subset of items is heavier than the capacity, we do not need to consider any solutions that use that subset. Similarly, if we know that not including a particular item (e.g. the first item) the best we can do is value $v_{upper}$, and we have already found a solution better than $v_{upper}$, then we no longer have to consider any solution that does not contain that crucial item.

You must now use the *frac_bound*() function to complete the branch-and-bound implementation. The outline of the algorithm can be given as follows (see *Algorithm Design and Applications* for more details).

1. Sort the items by decreasing value-to-weight ratio.
2. Compute the upper bound of the solution ****...
3. Compute the current values of each of the two solutions 0***... (or F***...) and 1***... (or T***...) (i.e. the total value of all the 1/Ts in each string), also their upper bound values, and check they are feasible.
4. If they are feasible we place them on a priority queue (Note we have provided implementations of a priority queue for all three languages).
5. In the next and all subsequent iterations, we remove the item with best bound value off the priority queue and again consider appending a 0 (C) or False (Java and Python) and a 1 (C) or True (Java and Python) to it.
6. The algorithm stops when the queue is empty or a solution (a complete solution with no stars in it) with value equal to the current upper bound is found.

Complete the *branch_and_bound* function given in *bnb.c/bnb_kp.java/bnb_kp.py*. This will call the fractional knapsack bound function *frac_bound*() and use the priority queue functions provided. More instructions are given in the code files.

When testing branch-and-bound you should consider stopping the program early if it is taking too long and think about how close the current best solution is to the actual optimal solution. Note that if you wait long enough you might hit the capacity of the priority queue – this probably indicates that you should give up trying to find an exact solution with branch-and-bound, although you can try making the queue larger.

**Blackboard Submission**

Once you have completed this implementation (or got as far as you can with it) you should look at the Blackboard submission form where you will be asked the following question about Part 2.

1. Run your branch-and-bound solution on the four files in the data directory, killing it if it runs for more than 60 seconds. Then answer the questions below for each file (no more than 100 words in total):

   (a) Did the run complete in under 60s? If so, how long did it take to run? You can use the Unix `time` utility for this. It is fine to just report the `real` time from this function.

   (b) Did it report a value as a possible knapsack value when it finished running or you killed it? If so what was that value?

   (c) If it reported a value, what was the difference between this and optimal value for that problem (See the end of Task 1a for a list of the optimal values)?

   I would expect to see here something like:

   ```
   easy.20.txt: 20.34s, reported value: 377, difference to optimal:0
   easy.200.txt: 35.5s, reported value: 4077, difference to optimal:0
   hard.200.txt: killed, reported value: 126967, difference to optimal: 1
   hard.2000.txt: killed, reported value: 1205250, difference to optimal: 9
   ```

2. What does the *frac_bound* function compute (there should be two things)? How do these relate to a partial solution? What is a feasible partial solution? (no more than 200 words).

3. Have you used a priority queue in your solution? If so, what does the queue use to prioritise its elements, where have you used it and what role is it playing? If not, what have you used instead of the priority queue where this was suggested in the code stub and what role is it playing? Illustrate your answer with the relevant parts of your code – you may omit lines of the code (use `...`) to focus on those bits most relevant to the use of the priority queue (don't show the code that implements the queue, but the code where you check elements of the queue, remove and add things to the queue). (no more than 300 words, not including code snippet).

**You should be here by the end of week 2 of this lab.**

---

## Task 3a: Greedy Algorithm

**Time Budget:** Task 3a is intended to take less than half an hour to complete. It is not needed elsewhere in the lab and should be abandoned if time is running out.

The greedy algorithm is very simple. It sorts the items in decreasing value-to-weight ratio. Then it adds them in one by one in that order, skipping over any items that cannot fit in the knapsack, but continuing to add items that do fit until the last item is considered. There is no backtracking to be done.

Write your own greedy algorithm in *greedy.c/greedy_kp.java/greedy_kp.py*. Note that we have provided a `sort_by_ratio` function/method (you should have encountered this when implementing the branch-and-bound solution).

### Blackboard Submission

Once you have completed this implementation (or got as far as you can with it) you should look at the Blackboard submission form where you will be asked the following question about Part 3a.

1. Run your greedy solution on the four files in the data directory, killing it if it runs for more than 60 seconds. Then answer the questions below for each file (no more than 100 words in total):

   (a) Did the run complete in under 60s? If so, how long did it take to run? You can use the Unix `time` utility for this. It is fine to just report the `real` time from this function.

   (b) Did it report a value as a possible knapsack value when it finished running or you killed it? If so what was that value?

   (c) If it reported a value, what was the difference between this and optimal value for that problem (See the end of Task 1a for a list of the optimal values)?

   I would expect to see here something like:

```
easy.20.txt: 0.05s, reported value: 368, difference to optimal:9
easy.200.txt: 0.1s, reported value: 4075, difference to optimal:2
hard.200.txt: 2s, reported value: 126579, difference to optimal: 399
hard.2000.txt: 2.8s, reported value: 1205167, difference to optimal: 92
```

## Task 3b: What makes a Knapsack problem Hard for Dynamic Programming?

**Time Budget:** Task 3b is intended to take 1-3 hours to complete. Please bear this in mind when investing time trying to figure out what happened if you do not get the results you expected.

### Generating Data

To help with experiments we have supplied a python script called `kp_generate.py` which will automatically generate knapsack instance files for you. It takes four arguments:

1. The first argument is the number of items you want to put in the knapsack
2. The second argument is the capacity of the knapsack
3. The third argument is an upper bound on the profit and weight of each item – the script will randomly generate values for profit and weight between 1 and this number.
4. The fourth argument is the name of the output file.

So, for instance if you call the script from the command line with

```
python3 kp_generate.py 5 6 7 test.txt
```

It will generate a file called `test.txt` whose contents will look something like:

```
5
1 6 5
2 3 3
3 5 1
4 3 4
5 7 5
6
```

Feel free to adapt and modify this script for your own use.

**NOTE:** This file isn't in the original `lab5` folder. You will need to pull from upstream to get it. See the instructions on Blackboard and at the start of this document.

### The Experiment

Obviously Knapsack problems get harder to solve the more potential items there are to put into the Knapsack, but that's not the only factor.

Construct an experiment to explore what else (beyond number of items) makes a knapsack problem hard for dynamic programming and write this up in your report. Think about how the theoretical complexity for dynamic programming is computed and what factors are considered in that analysis.

Your write up should have the following sections:

- a **hypothesis** stating what aspect of a knapsack problem *not counting the number of items* might make the problem harder for a dynamic programming solution with a brief explanation why. Ideally this should say how you think this factor will affect the running time of your implementation.
- a **design** which should cover the following points with justifications where necessary:
  1. What variable will you change across runs of the program?
  2. How did you create inputs that changed this variable? Did you use multiple inputs for each potential value of the variable?
  3. What did you measure and how?

  Please make these three things as clear as possible to see in your report. This will speed up marking and reduce the chance that your marker misses a point.
- a **results** section which should include a table or graph of your results, a description of any processing such as computing averages that took place, and the formulae used to generate any best fit lines that are shown.
- a **discussion** section that states whether your results support your hypothesis and why. If the results don't support your hypothesis then the discussion should consider why this might have happened.
- a **data statement** stating where any data or scripts you used or generated may be found (this could be in an appendix or in the gitlab repository).

In total your experiment write up (this includes, tables and graphs of summarised results but not appendices of raw data or scripts) should take up **at most** three sides of A4 at a sensible font size (12pt is a sensible font size, we won't be measuring font size but you will lose marks if we can't read your report comfortably because the letters are so small). You can find a template `report.tex` for your report in the `lab5` folder. You will need to pull from upstream to get the most up-to-date version of this (see instructions on Blackboard). When you have written it please upload your report *as a PDF* to Blackboard.

**What Makes a Knapsack Problem Hard for Branch and Bound?**

We're not assessing this, but will include discussion of an experiment in our sample answers and this might be assessed in the semester 2 exam.

The question of what makes a Knapsack problem hard for branch and bound is quite complex. Obviously, in the worst case, it performs no better than enumeration but in many cases, as you should have seen, it performs much better than enumeration. So what makes a knapsack instance cause branch and bound to behave more or less like enumeration? You might want to read up on weakly correlated knapsack instances to give you some ideas here and see if you come up with the same answers we will give in the lab sample answers.

# 1 C Instructions

This section contains some additional notes on completing the exercise in C.

### Task 1a: Full Enumeration

Take a look at the files.

- *enum.c* implements the full enumeration of all possible solutions as described above.
- *knapsack-util.c* provides some functions to read in a knapsack instance, print the instance details, print out a solution, and evaluate the values and weights of a solution. There's no need to edit *knapsack-util.c* during this lab, but you may do so if you wish.

Make enum and run it on `easy.20.1.txt`, which is a 0/1 knapsack problem instance with 20 items to pack:

```
make enum
./enum ../data/easy.20.1.txt
```

### Task 1b: Dynamic Programming

Complete the program that solves the 0/1 Knapsack Problem by dynamic programming. Use the files *knapsack-util.c* and *dp.c* we provide for you. Further instructions are in *dp.c*.

### Tasks 2a and 2b: Branch-and-Bound

Complete the program that solves the 0/1 Knapsack Problem by branch-and-bound. Use the files *knapsack-util.c* and *bnb.c* we provide for you. Further instructions are in *bnb.c*.

### Task 3a: Greedy

Complete the program that solves the 0/1 Knapsack Problem with a greedy algorithm. Use the files *knapsack-util.c* and *greedy.c* we provide for you. Further instructions are in *greedy.c*.

# 2 Java Instructions

This section contains some additional notes on completing the exercise in Java.

### Task 1a: Full Enumeration

Take a look at the files.

- *enum_kp.java* implements the full enumeration of all possible solutions as described above.
- *KnapSack.java* provides a class with some functions to read in a knapsack instance, print the instance details, print out a solution, and evaluate the values and weights of a solution. There's no need to edit *KnapSack.java* during this lab, but you may do so if you wish.

Note that *enum_kp.java* sub-classes *KnapSack*, as do all the other Knapsack solutions in this lab.

Compile enum_kp and run it on `easy.20.1.txt`, which is a 0/1 knapsack problem instance with 20 items to pack:

```
cd java/comp26120
javac enum_kp.java KnapSack.java
cd ..
java comp26120.enum_kp ../data/easy.20.1.txt
```

### Task 1b: Dynamic Programming

Complete the program that solves the 0/1 Knapsack Problem by dynamic programming. Use the files *KnapSack.java* and *dp_kp.java* we provide for you. Further instructions are in *dp_kp.java*.

### Tasks 2a and 2b: Branch-and-Bound

Complete the program that solves the 0/1 Knapsack Problem by branch-and-bound. Use the files *KnapSack.java* and *bnb_kp.java* we provide for you. Further instructions are in *bnb_kp.java*.

### Task 3a: Greedy

Complete the program that solves the 0/1 Knapsack Problem with a greedy algorithm. Use the files *KnapSack.java* and *greedy_kp.java* we provide for you. Further instructions are in *greedy_kp.java*.

## 3 Python Instructions

This section contains some additional notes on completing the exercise in Python.

### Task 1a: Full Enumeration

Take a look at the files.

- *enum_kp.py* implements the full enumeration of all possible solutions as described above.
- *knapsack.py* provides a class with some functions to read in a knapsack instance, print the instance details, print out a solution, and evaluate the values and weights of a solution. There's no need to edit *knapsack.py* during this lab, but you may do so if you wish.

Note that *enum_kp.py* sub-classes *knapsack*, as do all the other Knapsack solutions in this lab.
Run *enum_knp.py* on `easy.20.1.txt`, which is a 0/1 knapsack problem instance with 20 items to pack:

```
python3 enum_kp.py ../data/easy.20.1.txt
```

### Task 1b: Dynamic Programming

Complete the program that solves the 0/1 Knapsack Problem by dynamic programming. Use the files *knapsack.py* and *dp_kp.py* we provide for you. Further instructions are in *dp_kp.py*.

### Tasks 2a and 2b: Branch-and-Bound

Complete the program that solves the 0/1 Knapsack Problem by branch-and-bound. Use the files *knapsack.py* and *bnb_kp.py* we provide for you. Further instructions are in *bnb_kp.py*.

### Task 3a: Greedy

Complete the program that solves the 0/1 Knapsack Problem with a greedy algorithm. Use the files *knapsack.py* and *greedy_kp.py* we provide for you. Further instructions are in *greedy_kp.py*.

## Marking Scheme

This coursework is worth 15% of your final mark for COMP26120. This means each mark in this mark scheme is worth a little under 0.4% of your final mark for the module.

In the rubric below the marks an item is worth are slightly approximate due to the way Blackboard allocates percentages across sections. In general unsatisfactory performance will get you 10% of the available marks for a section, satisfactory will get you 50% and excellent will get you 100% but this can vary for items worth more than 1 mark because several criteria may be involved each of which can be marked unsatisfactory, satisfactory or excellent.

### Code Submission (1 mark)

| Description | Satisfactory | Excellent |
|---|---|---|
| ~~Commit Hash and Tag (1 mark)~~ | ~~A markable submission~~ can be identified (with a bit of work) using either the commit hash or tag provided, but either they point to different commits, only one is supplied, or neither directly identify a markable submission. | ~~The commit hash and tag~~ accurately identify to a markable submission |

### Part 1A (6 marks)

| ~~Description~~ | ~~Satisfactory~~ | ~~Excellent~~ |
|---|---|---|
| ~~Execution~~ | | ~~Code executes correctly~~ on `easy.20.txt` |
| ~~Progress Bar (2 marks)~~ | ~~A progress bar has been~~ implemented, but lacks thought about when to print (e.g., it prints every loop, or it prints only when current best changes) | ~~Good implementation of~~ a progress bar. |
| ~~Implementation (2 marks)~~ | ~~The implementation~~ logic is basically correct but contains minor errors (e.g., syntax errors) | ~~The implementation of~~ enumeration is correct. |
| ~~Student Testing~~ | ~~At least one of the questions~~ is believably answered | ~~All of the questions are~~ believably answered |

## Part 1B (7 marks)

| Description | Unsatisfactory | Satisfactory | Excellent |
|---|---|---|---|
| Execution (1 mark) | Code executes correctly only only one of `easy.20.txt`, `easy.200.txt` or `hard.200.txt`. | Code executes correctly on two of `easy.20.txt`, `easy.200.txt` and `hard.200.txt`. | Code executes correctly on `easy.20.txt`, `easy.200.txt` and `hard.200.txt` |
| Student Testing | | At least one of the questions is believably answered | All of the questions are believably answered |
| Implementation (2 marks) | Something that looks like dynamic programming but has major flaws | The implementation logic is basically correct but contains minor errors (e.g., syntax errors) | The implementation is correct. |
| Description (3 marks) | The description is incorrect or very difficult to follow. V and keep have been identifed but not correctly described. | The description of the implementation of dynamic programming is good but doesn't cover all the points needed for full marks. | The implementation is described clearly and concisely. This includes identifying V and keep, and describing the role they play in the program. |

## Part 2 (12 marks)

| Description | Unsatisfactory | Satisfactory | Excellent |
|---|---|---|---|
| Execution (1 mark) | Code executes correctly on only one of `easy.20.txt` or `easy.200.txt` | | Code executes correctly on `easy.20.txt` and `easy.200.txt` |
| Student Testing (1 mark) | | At least one of the questions is believably answered | All of the questions are believably answered |
| Implementation (3 marks) | Something that looks like branch-and-bound but has major flaws | The implementation logic is basically correct but contains minor errors (e.g., syntax errors), frac_bound isn't fully exploited in the solution. | The implementation is correct and uses all the functionality provided by frac_bound. |
| frac_bound (3 marks) | Discussion is unclear about what the function does and what partial and feasible solutions are. | Only one of the things it computes is clearly explained. There are minor errors in the description or it is over-length. | There is a clear and concise description of what frac_bound computes and what partial and feasible solutions are. |
| Implementation and Priority Queue Discussion (4 marks) | Correct statement of whether or not a priority queue was used, but no discussion of where (or what was used instead). There is some discussion of how the mechanism works but it is difficult to follow. | The discussion of the use (or otherwise) of priority queues contains minor errors or is over-length. | The implementation is correct. The discussion of the use (or otherwise) of the priority queue is clear, correct and concise. |

## Part 3A (4 marks)

| Description | Unsatisfactory | Satisfactory | Excellent |
|---|---|---|---|
| Execution | Code executes correctly only only one of `easy.20.txt`, `easy.200.txt`, `hard.200.txt` or `hard.2000.txt` | Code executes correctly on two of `easy.20.txt`, `easy.200.txt`, `hard.200.txt` and `hard.2000.txt` | Code executes correctly on `easy.20.txt`, `easy.200.txt`, `hard.200.txt` and `hard.2000.txt` |
| Implementation (2 marks) | There an attempt to implement the greedy approach, but it is very flawed causing multiple failures. | There are minor errors in the implementation but the logic is correct. | The implementation is correct. |
| Student Testing | | At least one of the questions is believably answered | All of the questions are believably answered |

## Part 3B

**Experiment 1 (10 marks)**

| Description | Unsatisfactory | Satisfactory | Excellent |
|---|---|---|---|
| Hypothesis (2 marks) | The stated hypothesis and explanation appear unrelated to the question posed. | A sensible hypothesis is given but not all details are supplied. The explanation for the hypothesis is unclear. | A sensible hypothesis is clearly stated and explained. |
| Design (3 marks) | A deeply flawed design, unlikely to yield results that can confirm or disprove the hypothesis or don't relate to the question | A reasonable design though with some flaws which might make results harder to interpret. Mention is made of what knapsack instances were generated, and what was measured. | A good design likely to confirm or disprove the hypothesis. It is clear what knapsack instances were generated, what was run and what was measured. |
| Results (1 mark) | Results are hard to understand or don't seem to relate to hypothesis and design. | Results are presented in a graph or table though there may be issues with missing labels, or there seems to have been some processing that isn't described. | Results are presented in a clearly labelled table or graph. Any processing of raw data is clearly described. There is a description that highlights key features of the results. |
| Discussion (1 mark) | There is a discussion of the results but it is over-confident or misinterpreting the results. | There is a discussion of the results that correctly states whether the hypothesis has been confirmed, disproved or is equivocal but either the hypothesis and design are too flawed to allow the conclusion to be drawn, the discussion is too brief to really elaborate what has been shown and why, or hypotheses about what went wrong are not plausible. | There is a discussion of the results that correctly states whether the hypothesis has been confirmed, disproved or is equivocal. If there result was unexpected there is a good discussion of why that might be. |
| Data Statement (1 mark) | No raw data can be found but some documentation does exist which would aid reproduction | Raw data for at least some of the results exists but is partial or other documentation is missing | Everything needed to reproduce the experiment is included. |
| Presentation (2 marks) | Presentation is poor. Either less than half the requested sections are in place, or there are persistent spelling and grammar errors which should have been caught by a spelling and grammar checker. | The report was submitted as a PDF. Presentation is acceptable. There may be minor spelling and grammar errors but these are not things that would be easily spotted by a spelling or grammar checker. All but one or two of the requested sections are present. Sometimes information would ideally be in a different section to the one in which it has been placed | The report was submitted as a PDF. The report appears to have been spell-checked. There are no major grammatical errors. All the requested sections are present with the appropriate information in them. |