85301 – Algorithms and Data Structures in Biology (2022/23)

Enrico Malizia and Riccardo Treglia

DISI, University of Bologna, Italy

Lab 5 & 6 – Exercise

In this lab session we will see few ways of speeding up the algorithm solving the knapsack problem.

1 The Problem

We will focus during this lab session on the *Knapsack Problem*. We are given in input:

- n items $\{0, \dots, n-1\}$;
- each item i is characterised by a size s_i and a utility u_i; and
- a maximum capacity C.

We want to compute a subset $T \subseteq \{0, \dots, n-1\}$ such that $\sum_{i \in T} s_i \leq C$ (i.e., the total size of the chosen items does not exceed the capacity C), and the total utility $\sum_{i \in T} u_i$ is maximised.

2 Different Versions of the Code

The code for each of the following sections needs to be in a separate file. These files should not reference each other. This means that you will need to copy code between them, which is normally a bad idea, but here we want to keep track of each incremental improvement to the code.

You should download the file `knapsack_tests.py` from Virtuale and save it in your working directory. Each of the files that you will create should begin with the line:

```
from knapsack_tests import testKnapsackCorrectness
```

Each of these files will implement a function `solveKnapsack(capacity, numberOfItems, sizes, utilities)` that takes as arguments:

- a non-negative floating point number `capacity`;
- a non-negative integer `numberOfItems`;
- a `list` `sizes` of positive floating point numbers such that `sizes[i]` is the size of item i; and
- a `list` `utilities` of non-negative floating point numbers, where `utilities[i]` is item i's utility;

and returns a `set` of indices of items (so a subset of $\{0, \dots, n-1\}$) representing an optimum choice of items (i.e., maximizing the total utility while keeping the total sizes below or equal the capacity). The returned set should be encoded as a `Python set`, *not* as a `Python list`.

Once you write this function in your file, you can test your code by calling `testKnapsackCorrectness("small")` (or `"medium"`, or `"large"`, to specify different input sizes). This will tell you whether your function correctly works and, if not, give you an input for which it gives an incorrect answer.

3 Naïve Exhaustive Search

The code of this section should go in a file named `knapsack_naive.py`.

In this section we are going to represent a subset of chosen items via a list of Booleans.[1] For example, if we have four items $\{0, 1, 2, 3\}$, the subset $\{1, 2\}$ is represented with `[False, True, True, False]`, where `True`, resp. `False`, in position `i` indicates that the element `i` is present, resp. not present, in the subset. We will convert this representation to a "standard" set just before giving the answer in output.

[1] You may opt for different representations; e.g., using lists of symbols, like `0` and `1` to represent that the object is absent and present, respectively. When in the following we will refer to the list of Booleans representing a subset, if you use a different representation, then adapt this guidance to your choices. The important thing is *not* to use a `Python set` of items, as we will perform some code optimisations for which we will need that subsets are represented via lists of Booleans/symbols.

Question 3.1. Write a function `generateAllBooleanCombinations(numberOfItems)` that returns the list of all lists of length `numberOfItems` of Boolean values. For example, calling the function `generateAllBooleanCombinations(3)` gives in output:

```
[[False, False, False], [False, False, True], [False, True, False], [False, True, True],
[True, False, False], [True, False, True], [True, True, False], [True, True, True]]
```

Question 3.2. Write a function `totalSize(sizes, choice)` that takes as arguments:

- the `list` `sizes` of the items' sizes;
- a `list` `choice` of Booleans representing a subset;

and returns the total size of the subset of items represented in the list `choice`.

Question 3.3. Write a function `totalUtility(utilities, choice)` that takes as arguments:

- the `list` `utilities` of the items' utilities;
- a `list` `choice` of Booleans representing a subset;

and returns the total utility of the subset of items represented in the list `choice`.

Question 3.4. Write a function `booleanListToSet(choice)` that takes as argument a list of Booleans representing a subset of items, and returns an actual `Python set` of items equivalent to `choice`. For example, `booleanListToSet([True, False, False, True, False])` returns the `Python set` `{0,3}`.[2]

Question 3.5. Write the function `solveKnapsack(capacity, numberOfItems, sizes, utilities)` according to the specification given above, and test it with `testKnapsackCorrectness("small")`.

4 Solving the Memory Hogging

You can test the code that you developed in the previous section on a slightly larger input by calling `testKnapsackCorrectness("medium")`—it may take a few seconds. Now, if you open the resource manager on your computer (on Windows, `CTRL+SHIFT+ESC` or `CTRL+ALT+DEL`) and sort processes by memory usage, you will see that during the test, your algorithm uses a lot of memory. On my computer, this is about 310 megabytes. This can be avoided by not actually generating the list of all possible Boolean combinations. We can here use the metaphor of the odometer seen during the lectures. Essentially we will cycle through all the possible Boolean combinations as if the list of Booleans were a counter.[3]

The code of this section should go in a file named `knapsack_counter.py`. From the file of previous section, copy everything except the function `generateAllBooleanCombinations`.

Question 4.1. Write a function `nextCombination(choice)` that takes as argument a subset of items (represented as a list `choice` of Booleans) and returns the next combination. (The function NextLeaf from the Part II lecture slides can give you a hint).

Question 4.2. Update the function `solveKnapsack` to use the new function generating all the combinations in turn. (The function AllLeaves from the Part II lecture slides can give you a hint).

Test the code that you have developed first with `testKnapsackCorrectness("small")`, then with `testKnapsackCorrectness("medium")`; the latter should not cause anymore a spike in memory usage.

5 Pruning the Search Tree

The module `knapsack_tests` has a last test instance: `"large"`, which tests `solveKnapsack` on an input with `numberOfItems=100`.

Question 5.1. Is there any chance that the code developed for the last section will pass the test `testKnapsackCorrectness("large")` in any reasonable amount of time? Why?

The intuition to deal with such a big instance is to modify the algorithm as follows: as soon as we realize that the total size of the current combination exceeds the `capacity`, we stop the search and backtrack immediately (figuratively pruning the search tree). Indeed, adding more items will certainly not decrease the final size below `capacity`!

[2] Remember to adapt this function if you are using a different representation for subsets.

[3] Again, if you are using a different representation for the subset, adapt this to your own choice.

So, create a file called `knapsack_prune_invalid.py` and, from the previous file, copy everything but the function `nextCombination`.

Since we are going to explore also the intermediate vertices of the search tree, as we need to perform cuts of the tree branches, we change a bit the representation of the current choice so to enforce coherence. From now on, in the list of Booleans representing the current choice, we will put elements `None` for those items for which wa have not taken a decision yet. For example, a list `[True,False,None,None]` will mean that we have decided that item 0 is taken, item 1 is *not* taken, and we have not decided anything yet regarding the two remaining items.

Question 5.2. Write a function `nextVertex(choice, level)` that takes as argument a subset of items (represented as a list `choice` of Booleans/`None`), and an integer `level` which tells up to which level the list `choice` is meaningful, and returns the next vertex in the search tree together with the new level. Implement the function so that the array elements set to `None` are kept coherent with the level of exploration in the search tree. (You can have a look at the function NextVertex form the slides of the Part II of the lectures to have a hint).

Question 5.3. Write a function `bypass(choice, level)`, whose arguments are like the function `nextVertex` above, and returns the next vertex in the search tree if the subtree rooted at `choice` is not explored. Also in this case, implement the function so that the array elements set to `None` are kept coherent with the level of exploration in the search tree. (You can have a look at the function Bypass form the slides of the Part II of the lectures to have a hint).

Question 5.4. Update the functions `totalSize` and `totalUtility` to take an extra argument `level` so that the sum of the sizes and utilities, respectively, is carried out up to the last meaningful element of the list `choice` (this just to save some time while computing these values).

Question 5.5. Update the function `solveKnapsack` to use the new function to traverse the vertices of the search tree, and to use the bypass function to cut the search tree when we find a combination exceeding the capacity.

Test that everything works correctly with `testKnapsackCorrectness("small")`.

Question 5.6. Measure again the computation time on the test instance "`medium`". What happens to the execution time compared to the previous case?

On my computer there is an increase in the computation time! How could we explain this?[4]

Question 5.7. Finally, test your code with `testKnapsackCorrectness("large")` and measure the computation time needed for the completion of the test. (It might take a while...)

Although this may take a while (on my computer it takes about 150 seconds), it is bearable compared to the potential time required by a blind exhaustive search throughout a search space of 2^{100} candidate subsets. Even if we were able to explore one billion combinations per second, we would need more than 40,000 billion years to explore all the possible 2^{100} combinations (just to have a rough idea of the order of magnitude of this, think that the currently estimated age of the Universe is "only" 14 billion years).

6 Avoiding Duplicate/Redundant Work

The computation time of the previous code version can be further improved by observing the following. When in the function `solveKnapsack` we compute a combination's size to decide whether to cut the tree there or not, we perform multiple times the same work. Indeed, consider two combinations A and B of items such that one is a subset of the other, let's say $A \subset B$, when we compute the total size of B we recompute also the total size of A.

For example, if $A = \{2, 4, 5, 7\}$ and $B = \{2, 4, 5, 7, 9\}$, we have that the total size of A is $s_2 + s_4 + s_5 + s_7$ and total size of B is $s_2 + s_4 + s_5 + s_7 + s_9$. Essentially, our code, to compute the sizes of A and B, performs twice the sum $s_2 + s_4 + s_5 + s_7$. We could save time if we incrementally computed the sizes of the generated combinations.

For the code of this section, create a new file `knapsack_prune_invalid_optimized.py` and, from the previous file, copy everything but the function `totalSize`.

[4] A possible reason is that, for the specific input instance considered, we have not cut much; the extra effort carried out to explore the vertices in the middle of the tree and to compute their sizes (in order to decide whether we can cut or not) is so big that in this circumstance it does not pay off.

Question 6.1. Let's update the code of the functions `nextVertex` and `bypass` so that now they compute the next combinations and also incrementally compute the sizes of the combinations returned. Update the two functions so that they take extra arguments `choice_size` and `sizes`, which are the size of the current combination passed in the parameter `choice` to the functions, and the array of the items' sizes. These functions should return pairs `(c,s)` where `c` is the next combination and `s` is the size of `c`. Update function `solveKnapsack`'s code to exploit the new feature of functions `nextVertex` and `bypass`.

Test that everything works correctly with `testKnapsackCorrectness("small")` and then with `testKnapsackCorrectness("medium")`.

Question 6.2. Measure again the computation time on the test instances "`medium`" and "`large`". What happens to the execution time compared to the previous case?

On my computer, for the "`medium`" case there is an improvement of around 50% and for the "`large`" case there is an improvement of around 75%.

7 Branch and Bound

There are a couple of additional optimizations that can be done to speed up the solution of the Knapsack problem: turning exhaustive search (with pruning) into branch and bound.

7.1 (Plain) Branch and Bound

The first idea is to improve the exploration of the search tree in the following way. If we keep track of the utility of the (partial) combinations generated, we can check whether the combination that we are building has hope or not to produce a combination that in the end will be an optimal one.

For example, let's assume that we are in the case that there are ten objects in total and that we have considered objects from 0 to 5, and we have decided to include objects 1, 3, and 4 (so, we have decided to exclude objects 0, 2, and 5). The utility of the current combination $\{1, 3, 4\}$ would be $u_1 + u_3 + u_4$. Since we have already processed the objects from 0 to 5 and we have to consider only the objects from 6 to 9, in the best possible scenario the utility at the end of the construction of the combination is $U(\{1, 3, 4\})$ (i.e., the utility of $\{1, 3, 4\}$) plus the utility that we would gain by adding *all the remaining* objects, i.e., $\sum_{i=6}^{9} u_i$ (in the very optimistic scenario that we will have space enough to accommodate all the remaining objects). If this "best hope utility" does not exceed the best optimum that we have computed so far, then there is no point in exploring the subtree rooted at that combination.

So, for this subsection, create a new file `knapsack_branch_and_bound.py` and, from the previous file, copy everything but the function `totalUtility`.

Question 7.1. Similarly to what we did in Question 6.1, we have to update the code of the functions `nextVertex` and `bypass` so to incrementally compute the utility of a combination as well.

The principle is very similar to what we did for the size (see Question 6.1). In this case, `nextVertex` and `bypass` take two extra arguments `choice_utility` and `utilities`, which are the utility of the current node passed to the functions in the argument `choice`, and the array of the items' utilities, respectively. These functions should return a triple `(c, s, u)`, where `c` is the next combination, `s` is the size of `c`, and `u` is the utility/value of `c`.

Update the code of the function `solveKnapsack` to exploit the new feature of functions `nextVertex` and `bypass`. (Observe that we are not still implementing the idea for the branch and bound).

Test that everything works correctly with `testKnapsackCorrectness("small")` and then with `testKnapsackCorrectness("medium")`.

Question 7.2. To implement our branch and bound idea, we need to know the utility of the remaining objects (see above). We want to do this in clever way and avoid to recompute many times the same things.

Write a function `remainingUtilities(utilities)` taking as input a `list` of utilities and returns a `list` storing at each position i the sum of the utilities of all object from object i (included) to the last: e.g., for a list of utilities `[1,2,3,4]`, `remainingUtilities([1,2,3,4])` returns `[10,9,7,4]`.

Question 7.3. We now implement the branch and bound idea into the function `solveKnapsack`. As a first step, in the `solveKnapsack` function, we pre-compute all the remaining utilities via the function of the question above. This has to be done *before* the search in the tree starts.

Then, in the part of the code in `solveKnapsack` dealing with the (partial) combinations, i.e., the vertices in the middle layers of the search tree, after checking that the current node does not exceed the capacity (and hence it makes sense to keep exploring the subtree), we check whether the (partial) combination considered is "promising" or not. We can perform this by computing the "best hope utility" also thanks to the above defined function `remainingUtilities`. If the "best hope utility" does not exceed the currently computed best value, it does not make sense to explore the subtree further.

Test that everything works correctly with `testKnapsackCorrectness("small")` and then with `testKnapsackCorrectness("medium")`.

Question 7.4. Measure again the computation time on the test instances "`medium`" and "`large`". What happens to the execution time compared to the previous case?

On my computer, for the "`medium`" there is a quite big improvement, while for the "`large`" case the improvement is quite limited. Why could this be the case?

The reason is related to the order in which the candidate combinations are explored. Intuitively, if good combinations are explored at the beginning of the search, then the branch and bound cuts that we perform on the search space based on the current best value can be quite big. On the other hand, if the best combinations are explored as lasts, then the cuts based on the current best value are not so big, because, each time we find something better, and hence we cannot cut much (or nothing at all).

If we could invent a way to generate at the beginning of the search the combinations that are likely to be the best, then we could exploit quite a lot the cuts based on the current best value.

7.2 Branch and Bound with Heuristics

The optimization that we are going to implement now can speed up quite dramatically our algorithm. The idea is essentially to *rearrange the order* in which we consider the objects to build our combinations, trying, for what we can, to consider *first* the most *promising* objects. In this way, we increase the probability to generate, at the beginning of the search, combinations with very high utilities, so that we can have a very high "current best value" since the beginning, and cut much more the search space. We call *heuristics* the principles/criteria upon which we *hope* to build reorderings allowing us to build from the start better combinations.[5] In particular, the idea is the following:

- we can consider the objects in order of *decreasing density*, where *density* of the object i can be defined as $d_i = \frac{u_i}{s_i}$, instead of looking at them in the order in which they are given in input;
- we change the notion of "best hope utility": in particular, for each combination S for which we have considered the objects from 0 to i, we know that the utility of any combination obtained by extending S and considering the remaining objects is at most:

$$U(S) + remaining\_capacity \times max\_density\_of\_unexplored\_objects, \tag{1}$$

  where $U(S)$ is the utility of the combination S;
- if this quantity is not greater than the best utility found so far, then we do not explore the subtree rooted at the current combination, because we know that none of its leaves will ever give an optimal result (they will all be worse than the current best);
- note that the maximum density of the unexplored objects is simply the density of the *next* unexplored object, since we are exploring the objects in order of decreasing densities.

So, create a new file called `knapsack_branch_and_bound_heurisitics.py` and, from the previous file, copy everything but the function `remainingUtilities`.

Question 7.5. Write a function `densityOrder(numberOfItems, sizes, utilities)` that returns the pair `(indices, densities)`, where:

- `indices` is the `list` of the indices of the objects, in order of decreasing density, and
- `densities` is the `list` of the densities of the objects, in decreasing order,

so that for all `i`, the object index `indices[i]` has density `densities[i]`. For example:

[5] We underline here that a heuristics increases only our *hope* to obtain a solution quickly. A heuristics does *not* guarantee that our algorithm will always return an answer quickly. Even with the use of heuristics, the complexity of the algorithm in the worst-case is still *exponential*. A heuristics is like adding a greedy approach to an algorithm that is an exhaustive search. With a heuristic the algorithm does not become entirely greedy, because we check anyway *all* combinations, i.e., we reconsider our choices in search for the optimum one. In a purely greedy approach, we would perform choices based on local optimality criteria *without* subsequently reconsidering our choices.

```
>>> densityOrder(4, [2,4,1,6], [4,5,3,9])
([2, 0, 3, 1], [3.0, 2.0, 1.5, 1.25])
```

You can use the `.sort()` method of the `Python` library to sort a list of pairs in lexicographic order (i.e., comparing the first members first).

Question 7.6. Update the function `solveKnapsack` so that, instead of initially calling the function `remainingUtilities` to compute the remaining utilities, it computes via `densityOrder` the ordered densities and the ordered object indices.

Question 7.7. Update the functions `nextVertex` and `bypass` so that, when building the combinations, objects' indices are considered in order of decreasing densities.

Question 7.8. Update the function `solveKnapsack` so to use to new "best hope utility" estimation according to Equation (1) above.

Test that everything works correctly with `testKnapsackCorrectness("small")` and then with `testKnapsackCorrectness("medium")`. Measure the computation time for the case "`large`". Now it should be blazing fast.

8 Greedy Approaches for the Knapsack Problem

In the previous section, we integrated a heuristic approach with branch and bound. However, as we said, this does not make the branch and bound search into a greedy approach, as in the former the entire search space is anyway explored. A greedy approach, on the other hand, would build a(n approximate) solution for the Knapsack problem by looking at the objects in turn and deciding whether to include them or not (based on a local optimality criterium), and these choices, once performed, are never re-considered.

We can invent several greedy approaches for the Knapsack problem; we are going to consider two of them and an approach that is their combination.

Before starting, create a new file `knapsack_greedy.py` and, from the previous file, copy everything.

8.1 Considering Items in Order of Their Values

Let us consider a first greedy approach, that in the following we will refer to as Approach A, in which items are considered according to their values: Objects with higher values are considered first, and they are added to the solution until there is enough space to accommodate them. As soon as there is not enough space available for the considered object, we stop and we return the solution.

Question 8.1. Write a function `valueOrder(numberOfItems, utilities)` that returns a `list` of the indices of the objects in order of decreasing value. For example:

```
>>> valueOrder(4, [4,5,3,9])
[3,1,0,2]
```

You can use the `.sort()` method of the `Python` library to sort a list of pairs in lexicographic order (i.e., comparing the first members first).

Question 8.2. Implement the greedy Approach A into a `Python` function `greedyKnapsack_A(capacity, numberOfItems, sizes, utilities)`, whose input parameters are as those for `solveKnapsack` and returns a `Python set` of objects according to the greedy Approach A.

Question 8.3. Is the approximation ratio of Approach A bounded or not? (Answer in the Appendix)

Question 8.4. Compare the results provided, and the computation time needed, by this greedy algorithm and the last branch-and-bound algorithm with heuristics.

8.2 Considering Items in Order of Their Densities

From what we have seen above, considering the objects in order of their values does not seem to be a great idea, as the approximation ratio of Approach A is *not* bounded.

We can consider, hence, a second approach, that we will call Approach B. In this approach, we proceed as in the heuristics with which we improved branch and bound in Section 7.2: Objects with higher densities (and not values, in this case) are considered first, and they are added to the solution until there is enough space to accommodate them. As soon as there is not enough space available for the considered object, we stop and we return the solution.

Question 8.5. What is the difference between this greedy approach and the approach in which we use this heuristics together with the branch and bound?

Question 8.6. Implement the greedy Approach B into a `Python` function `greedyKnapsack_B(capacity, numberOfItems, sizes, utilities)`, whose input parameters are as those for `solveKnapsack` and returns a `Python set` of objects according to the greedy Approach B.

Question 8.7. Is the approximation ratio of Approach B bounded or not? (Answer in the Appendix)

Question 8.8. Compare the results provided, and the computation time needed, by this greedy algorithm and the last branch-and-bound algorithm with heuristics.

8.3 A 2-Approximation Algorithm for the Knapsack Problem

Considering the objects in order of their decreasing values or in order of their decreasing densities are two different heuristics (or local optimality ciriteria) giving rise to two different greedy approaches for the Knapsack problem.

Question 8.9. Approaches A and B are similar, but different. Find two Knapsack instances such that Approach A and B compute different results, and, for one instance, Approach A's result is better than Approach B's one, and, for the other instance, Approach B's result is better than Approach A's one.

Question 8.10. Since Approach A and B might provide different results for the same instance, can you think of a simple way to combine the two approaches and obtain an approach that is an improvement over Approaches A or B? (The answer is below; please take some time to think about this)

A simple improvement over individually using Approaches A and B is using them in tandem: given a Knapsack instance I, we compute Approaches A and B's results for I, and we return the best of them. Surprisingly, such a combined approach, although based upon greedy approaches with unbounded approximation ratio, *does* have a bounded approximation ratio of 2 (see Theorem A.4 in the Appendix).

Question 8.11. Implement into a `Python` function `greedyKnapsack(capacity, numberOfItems, sizes, utilities)` the just described combined greedy approach; the arguments are as those for `solveKnapsack` and a `Python set` of objects according to the combined greedy approach of this section is returned.

9 Concluding Remarks

In this exercise, we have dealt with the Knapsack problem, and we implemented various algorithms for its solution. What we have focused on is the way in which the search space is generated and explored, and the possible cuts that we can perform on the search space.

It is important to observe that the way in which our algorithms are designed does not work in all optimization scenarios, but only in those that are "compatible" with the Knapsack problem. More specifically, focus for example on the cut that we perform on the search space when we realize that the current combination is already too big. The idea was to cut because by adding new elements the size could only increase. We were adding elements because our aim was to generate the highest value combination, and to add value to our combinations we need to add objects. In this context, the above mentioned cut is meaningful, as once we hit the capacity threshold, it does not make sense to add more objects.

There are problems in which, on the contrary, the optimum is reached by minimising the "objects" chosen. In these cases, validity constraints can be violated because we are choosing too few objects, and hence considering a combination with even fewer objects does not make sense: this is the moment in which we perform a cut on the search space; as you can see this is pretty different from, and is not compatible with, what we have seen in these exercises. In these cases, to have things properly working, we need to explore the search space by generating the candidate combinations starting from the combination including *all* "objects", and then trying to remove each object in turn to minimise the goal function.

Appendix: Some Answers and Missing Details

Answer A.1 (Answer to Question 8.3)**.** Approach A's approximation ratio is *not* bounded. To see this, consider, e.g., this family of Knapsack instances: the objects are $\{0, \ldots, n\}$ (i.e., $n+1$ objects), the max capacity is C, the object sizes are $s_0 = C$ and $s_i = \frac{C}{n}$, for all objects $1 \leq i \leq n$, and the object utilities

are $u_0 = 2$ and $u_i = 1$, for all objects $1 \leq i \leq n$. Approach A returns as answer the set containing the single object $\{0\}$ (because object 0 has the highest value, but once we add it to the solution, there is not enough space to add any other object), whose value is 2; however the best solution is the set of objects $\{1, \ldots, n\}$ whose value is C. Approach A's approximation ratio is then at least $\frac{C}{2}$, which depends on the input value and in principle can be made very big.

Answer A.2 (Answer to Question 8.7)**.** Approach B's approximation ratio is *not* bounded either. To see this, consider, e.g., this family of Knapsack instances: the objects are $\{0, 1\}$ (i.e., 2 objects), the max capacity is C, the object sizes are $s_0 = 1$ and $s_1 = C$, and the object utilities are $u_0 = 2$ and $u_1 = C$. Approach B returns as answer the set containing the single object $\{0\}$ (because object 0 has a higher density, and once we add it to the solution, then there is not enough space to add object 1), whose value is 2; however the best solution is the set containing the single object $\{1\}$ whose value is C. Also Approach B's approximation ratio is then at least $\frac{C}{2}$, which depends on the input value and in principle can be made very big.

Answer A.3 (Answer to Question 8.9)**.** In the following case, the solution given by Approach B (density order decreasing) is better than the one given by Approach A (value order decreasing). Let C be the capacity and $n > C$ be the number of items. One of the items, say the nth, has size C and value C, and all the remaining items have size 1 and value $C - 1$. The item of the maximum value is item n and alone covers the capacity of the knapsack. However, by taking C other items the overall value is $C \cdot (C - 1)$, reaching a total value higher than just C. An actual instance could be: capacity $C = 3$, number of items $n = 4$, sizes $s = [1, 1, 1, 3]$, and utilities $u = [2, 2, 2, 3]$.

In some cases, it is possible for Approach A to produce a better solution than that of Approach B, for example, where we have just two elements, one with size C and value C, and the latter with size 1 and value 1.1. The idea is that the second element will go into the knapsack just because the density is more than 1. Approach A would correctly have allotted the first element, instead. An actual instance could be: capacity $C = 3$, number of items $n = 2$, sizes $s = [3, 1]$, and utilities $u = [3, 1.1]$.

Theorem A.4. *The greedy approach for the Knapsack problem described in Section 8.3 has an approximation ratio of* 2*.*

Proof. Let $I = \langle \{0, \ldots, n-1\}, \{s_i\}_{0 \leq i \leq n-1}, \{u_i\}_{0 \leq i \leq n-1}, C \rangle$ be an instance of the Knapsack problem. Let us consider the solution for I computed by Approach B. For simplicity, let us assume without loss of generality that the object's indices are already ordered according to their decreasing value density $d_i = \frac{u_i}{s_i}$; i.e., for two objects i and j such that $i < j$, we have that $d_i \geq d_j$ (we can assume this because objects can always be renamed). Let us assume that the solution computed by Approach B is $S = \{0, 1, \ldots, k\}$, i.e., the object with index k is the last object added to the solution, and the object with index $k + 1$ is the first object that is *not* added to the solution. Let Opt_I be the value of a real optimum solution for I (i.e., a solution computed by an exhaustive search). We are going to show that $Opt_I \leq \sum_{i=0}^{k} u_i + u_{k+1}$.

In order to prove the above statement, we need to consider a "relaxed" Knapsack scenario in which we are allowed to fill the bag with *parts* of objects—let us call this scenario *fractional knapsack*; e.g., there is not enough space left in the bag for an entire diamond, and we take only half of it to fill the bag.

If we consider the fractional knapsack, the optimum value for I, as we will show below, is achieved by taking all objects in S (i.e., Approach B's solution for I) plus a *fraction* of the first object excluded from S so to fill the space left available in the bag. This is the *optimum* solution for the *fractional knapsack*, because this solution uses all objects of density greater than d_{k+1} and fills the remaining part of the bag with value density d_{k+1}, and all excluded objects have density not greater than d_{k+1}.

However, the optimum value for the standard (non-fractional) knapsack, i.e., if cutting objects is *not* permitted, can only be smaller than the optimum value for the fractional knapsack, because in the standard case we must take objects in their entirety and we cannot freely cut them. This is even more true if, instead of taking only a portion of the first object excluded from S, we take the entire object $k + 1$. From this we have exactly what we were looking for: $Opt_I \leq \sum_{i=1}^{k} u_i + u_{k+1}$

To conclude, observe that $\sum_{i=0}^{k} u_i = V_b$, i.e., this is the value of the solution returned by Approach B, and, if we denote by V_a the value of the solution returned by Approach A, we have that $u_{k+1} \leq V_a$.

By combining these relations we have that: $Opt_I \leq \sum_{i=0}^{k} u_i + u_{k+1} = V_b + u_{k+1} \leq V_b + V_a$, implying that $Opt_I \leq V_a + V_b$. Since the sum of V_a and V_b is at least Opt_I, it *cannot* be the case that both of them are strictly smaller than $\frac{1}{2} Opt_I$, hence V_a or V_b must be at least $\frac{1}{2} Opt_I$, which implies that the biggest of them is at least $\frac{1}{2} Opt_I$, and thus the approximation ratio is at most 2 (from $\frac{Opt_I}{\frac{1}{2} Opt_I}$). □