Brute-force searches

Consider the following problem:

> Write a function that takes a number of arguments. The first is a positive real number that gives the volume of a knapsack. The second argument is an array of positive real numbers representing the volume of a number of items that could be put into the knapsack. Your goal is to determine the maximum number of objects that can be placed into the knapsack without exceeding the volume, and to then print the indices associated with each entry that would be placed into the knapsack.

Here is a similar, but more difficult problem:

> Write a function that takes a number of arguments. The first two are positive real numbers that gives the volume and bearing capacity (mass) of a knapsack. The third and fourth arguments are array of positive real numbers representing the volume and mass of the k^{th} item, respectively. Your goal is to determine the maximum number of objects that can be placed into the knapsack without exceeding the volume or bearing capacity, and to then print the indices associated with each entry that would be placed into the knapsack.

Here are two more interesting variations of these problems:

> Write a function that takes a number of arguments. The first is a positive real number that gives the volume of a knapsack. The second argument is an array of positive real numbers representing the profit that would be gained by transporting the k^{th} item in the knapsack. The third argument is an array of positive real numbers representing the volume of the k^{th} item. Your goal is to find and print that collection of items that can be placed into the knapsack that maximizes the profit without exceeding the volume.

> Write a function that takes a number of arguments. The first two are positive real numbers that gives the volume and bearing capacity (mass) of a knapsack. The third argument is an array of positive real numbers representing the profit that would be gained by transporting the k^{th} item in the knapsack. The fourth and fifth arguments are array of positive real numbers representing the volume and mass of the k^{th} item, respectively. Your goal is to find and print that collection of items that can be placed into the knapsack that maximizes the profit without exceeding the volume or bearing capacity.

These are called *Knapsack Problems*. You will often come across a number of problems that have fanciful, silly, entertaining or otherwise memorable names. The purpose is so that you remember them. This includes, for example, the *Traveling Sales Representative Problem*, which asks, given n cities, in which order should a sales representative visit these n cities so as to minimize the total distance traveled in visiting all of the cities. You may also, if you begin to work with independent objects (say, drones) communicating with each other, become aware of the *Byzantine General's Problem*.

These memorable names are there to help developers remember both the problem and the solutions, or more likely, to help developers remember that solutions exist.

The *Knapsack Problem* has many applications. For example, a company may be able to choose from many different contracts over the next month or year; however, each contract requires a fixed number of employees to work on that contract, and each contract will require a fixed amount of hours that would be required to complete that contract. A company generally has fixed human resources available and there is only so much time available in a month. Consequently, the executive that is deciding which contracts to accept and which to reject would like to choose those that will, of course, maximize profits.

In the original problem as proposed above, what can be put into the knapsack has one or more constraints. For the executive deciding which contracts to take, once again, there are two constraints: time and human resources. It is not a good idea to hire new people just to complete more projects, as more time is required to train and integrate those new hires into the company than may be worth it.

Problem 1: Maximum number not exceeding the volume

We will start by looking at maximizing the number of items that can be placed into the knapsack without exceeding its volume capacity. Before we begin to describe the solution to this problem, try to determine how you would solve this problem. You actually have the tools right now to do this. Don’t go onto the next page until you have thought about this for some time. There is a reasonably fast algorithm, and you may consider looking at where this problem solving question is placed. ☺

As you may have guessed, to maximize the number of items that can be placed into the knapsack without exceeding the volume of that knapsack, start by putting the smallest item into the knapsack first, then the second, and so on, until placing the next item in exceeds the knapsack's capacity (either that, or you've fit everything possible into the knapsack). For a computer to do this, to order the items from least volume to greatest volume, it is necessary to sort the items from least volume to greatest volume, and then start placing the items from least to greatest into the knapsack until no further items can be placed into the knapsack.

This approach of sorting first can actually be used in industry. Suppose you have three people in line in a grocery store. One has a bag of milk, the second has the dozen or so items required to make tonight's dinner, and the third has a shopping cart full of items. Ignoring any individual's feelings, to reduce how much everyone is waiting, it makes more sense to serve the person with one item first, then the person with a dozen items, and then the person with the cart-full of items. Indeed, this author used to regularly allows an individual behind this individual's self to go first if that person has only one or two items and this author has twenty or more items; however, with self-serve check-outs, this has become less-and-less necessary. A similar strategy could be used to service web requests: if there are requests that are known to be serviceable in only 1 ms, complete those before those that will require one second. This works, so long as there are not hundreds of smaller requests, and this keeps the users with the quicker requests happy, and the user that made the larger request now has to wait 1.05 seconds instead of one second.

Returning to solving the problem, however, our goal is to print out the indices associated with the items that can fit into the knapsack. If we sort the array, then we will print out 0, 1, 2, 3, ..., k where k is the last item that can be put into the knapsack.

Thus, we need to have some way of sorting the items, but still being able to recall which original index each item was associated with in the original array. We will look at two solutions that solve this problem, but before we look at these solutions, try to come up with one on your own, that is: How do we sort a list of items, but are still able to recall which index the item came from before the list was sorted?

Sorting two arrays based on the order in one

One solution is to sort the array that contains the value we want to sort, but each time we swap two entries in that array, we also swap two entries in the other array, as well. Thus, for example, if we started with two arrays, the first containing numbers we would like to sort, and the second containing the integer values from 0 to capacity – 1. Here is such an example:

0	1	2	3	4	5	6	7
0.7343	-0.9248	1.5432	9.2345	-5.4291	8.7352	3.5927	6.4938
0	1	2	3	4	5	6	7

If we were to now sort the first array, and each time we temporarily stored or assigned an entry in the first array, we would do the same operation on the corresponding entries in the second, we would get

0	1	2	3	4	5	6	7
-5.4291	-0.9248	0.7343	1.5432	3.5927	6.4938	8.7352	9.2345
4	1	0	2	6	7	5	3

From this, we note that in the original array, the entry at index 4 is the smallest, followed by the entry at index 1; and the entry at index 3 is the largest with the entry at index 5 being the next largest.

The problem with this approach is that it:

1. Requires us to change the order of the arrays containing the information we need, which may not be desirable (we may need the arrays later in their original order), and
2. What happens if we keep adding more and more arrays, containing additional information about the *k*th item in the original unsorted array. Do we have to write more functions that performs the same operations, only now reordering additional arrays?

Sorting an array of indices

Thus, a different approach may be better: Pass in an array to be sorted, but rather than sorting the array, return an array of indices so that

```
   array[indices[0]] <= array[indices[1]]
&& array[indices[1]] <= array[indices[2]]
&& array[indices[2]] <= array[indices[3]]
&& …
&& array[indices[capacity - 2]] <= array[indices[capacity - 1]]
```

Thus, the entry of `indices[0]` gives the index in array where the smallest entry is, and so on and so forth. This need not change the order of the original array, so as a benefit, the array can also be declared to be const.

The following modification to our insertion sort functions will do exactly that. You will note that each time we compare two entries to see which is larger or smaller, we always access `array[indices[k]]`, but each time we change any of the entries, we only change the entries in the array `indices`:

```
void insertion_sort( double      const array[],
                     std::size_t       indices[],
                     std::size_t const capacity ) {
    for ( std::size_t k{0}; k < capacity; ++k ) {
        indices[k] = k;
    }

    for ( std::size_t k{2}; k <= capacity; ++k ) {
        insert( array, indices, k );
    }
}

void insert( double      const array[],
             std::size_t       indices[],
             std::size_t const capacity ) {
    std::size_t index{ indices[capacity - 1] };

    std::size_t k{};

    for ( k = capacity - 1;
         (k > 0) && (array[indices[k – 1]] > array[index]); --k ) {
        indices[k] = indices[k - 1];
    }

    indices[k] = index;
}
```

Now that we have a solution for *sorting* the array, or more specifically, getting a list of indices that give the required sorting, we will now proceed to finding the maximum number of objects that we can put into the knapsack if our only constraint is that the total volume cannot exceed the volume capacity of the knapsack.

Filling the knapsack with a fixed volume

Thus, we may now solve the problem of finding the maximum number of items, and which items in particular, we can fit into a knapsack with a given volume, with the only constraint being that the volume of the items being placed into the knapsack do not exceed the volume. We will:

1. Sort the items, or more specifically, get a list of indices that indicate the order of the items.
2. Start adding the items from smallest to largest until the volume capacity is exceeded or we have added all objects.

We will print the indices of the items that were added in the order they were added, and we will return the number of items that were added:

```
std::size_t maximize_volume( double      const knapsack_volume_capacity,
                             double      const item_volumes[],
                             std::size_t const capacity ) {
    std::size_t indices[capacity];

    insertion_sort( items, indices, capacity );

    double current_volume{ 0.0 };

    for ( std::size_t k{0}; k < capacity; ++k ) {
        if ( current_volume + item_volumes[indices[k]]
                                  <= knapsack_volume_capacity ) {
            std::cout << indices[k] << " ";
            current_volume += item_volumes[indices[k]];
        } else {
            std::cout << std::endl;
            // Items from index 0 to 'k - 1' were added,
            //    so 'k' items were added
            return k;
        }
    }

    // We have added all the objects.
    std::cout << std::endl;
    return capacity;
}
```

Other problems: a brute force search

For any other problem described above, there is no such simple and fast algorithm that will find the optimal solution. There are simple and fast algorithms that often find near-optimal solutions, but right now, we will discuss a brute-force search that tries all combinations of the objects to be added, and then prints that combination that maximizes whatever we are attempting to maximize (either the number of items in the knapsack, or the total profit). Conceptually, this is simple, but from a run-time point-of-view, it is the worst possible choice. Never-the-less, it is guaranteed to find the optimal solution.

If we have n items, how do we try every combination of those n items? We'll start by trying something simpler: given four items, how can we try all combinations of those four items? Through trial and error, you may come up with a list like the following, that includes all possible combinations of either adding an item ("T" for "true") or not adding an item ("F" for "false"):

```
F F F F
F F F T
F F T T
F F T F
F T T F
F T T T
F T F T
F T F F
T T F F
T T F T
T T T T
T T T F
T F T F
T F T T
T F F T
T F F F
```

To have a computer go through such a list, however, requires an algorithm. If you were to reinterpret each the "`T`" as "`1`" and each of the "`F`" as "`0`", notice what you get the sixteen binary numbers. Do you notice anything interesting about these sixteen numbers?

```
0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000
```

Through careful observation, you may have noticed this list includes all numbers from 0 (`0000`) to 15 (`1111`) in binary. Thus, we could have a for loop where *n* from 0 to 15, and at each step, interpret the *k*th bit of *n* to say whether or not the *k*th item is being included. Thus, for four items, we need four bits. If we had *N* items, it is not to difficult to see that this would require *N* bits.

Now, the largest integer data type in C++ is `long` (or `long long` if you are using the Microsoft Visual Studio compiler) which has 64 bits. Thus, this data type could be used to determine if up to 64 items can be either added or not added into the knapsack. That does not seem like a lot, but let us consider the following:

Suppose it takes one nanosecond (10^{-9} s) to check each combination. Thus, if we were checking all 2^{64} possible combinations of either adding or not adding 64 items, this would take $2^{64}\cdot10^{-9}$ s or approximately $16\cdot10^{18}\cdot10^{-9}$ s or 16 000 000 000 seconds or approximately half a millennium. Thus, as one can see, this is not necessarily the most efficient algorithm, and subsequent courses, you will be introduced to better algorithmic techniques that significantly reduce the run time. Now, ideally, we should be able to go up to 64 items using the type `long`; however, this requires some not-so-obvious and sometimes subtle coding techniques that would take away from the understanding of the general approach, so we will allow up to 63 items, instead.

Also, we will look at the most difficult program: one that constrains both volume and the mass of the items being considered while maximizing the profit. From this, you should be able to implement solutions for the simpler problems.

Important programming rule: Always do similar tasks in the same order. If you look at the order of the parameters, you will see that the arrays are in the order of profits, volumes, and masses. Profit comes first, as that is the critical factor (what we want to maximize), and colloquially we generally refer to the 'volume of and mass' of objects, as opposed the 'mass and volumes' of objects. Indeed, a Google search of both these exact phrases has approximately twice the number of hits for the first as for the second. Consequently, every time in the function that these arrays are being accessed, it is best practice to perform operations in the same order: first update or access profits, then the volumes, and then the masses, in that order. That way, another programmer reading your code can become familiar and understands what to expect. When this author first wrote this code, often this person was tempted to put, for example, volume and mass before calculating profits; however, this author subsequently went back and reordered all statements to be consistent throughout the entire function. Such attention to detail will simplify the lives of anyone who is obligated to read through the code you have written.

Note also that the profit is actually implemented as an integer. One normally thinks of money as being a real number; for example, $5.23. Using double, however, is a poor choice for money, as it is not necessarily exact, and thus subject to rounding errors as well as other errors. It is better to think as money in terms of the total number of cents something is valued at, as opposed to the number of dollars it is valued at. This ensures that exact arithmetic is used at all times when calculating anything with respect to money.

```
void maximize_profit(double       const knapsack_volume_capacity,
                     double       const knapsack_mass_capacity,
                     unsigned int const item_profits[],
                     double       const item_volumes[],
                     double       const item_masses[],
                     std::size_t  const capacity ) {
    // This will not work for more than 63 items
    //  - We should also check to make sure that none of the volumes or masses
    //    are negative. It may be possible that an item carries with it a negative
    //    profit (that is, a cost incurred in including it), but in that case,
    //    it should simply be removed before we even call this algorithm.
    assert( capacity <= 63 );

    unsigned int max_profit{ 0 };
    unsigned long max_items{ 0 };     //                  capacity
    int max_count{ 1 << capacity }; // This calculates 2

    //                                           capacity
    // Test every capacity-bit number from 0 to 2         - 1
    for ( std::size_t items{ 0 }; items < max_count; ++items ) {
        // For the current collection, initialize
        //     the profit, volume and mass to 0.
        unsigned int profit{ 0 };
        double       volume{ 0.0 };
        double       mass{ 0.0 };

        // Here, we initialize two loop variables instead of one.
        // The loop variables follow the order (0, 1), (1, 2), (2, 4), (3, 8), ...
        // We only halt if the first equals capacity, but at the end of the loop
        // body, 'item' is incremented, while 'bit' is shifted to the left by one.
        for ( std::size_t item{ 0 }, bit{ 1 }; item < capacity; ++item, bit <<= 1 ) {
            // If the item'th bit is set to '1', then include the
            // item in the current collection.
            if ( items & bit ) {
                profit += item_profits[item];
                volume += item_volumes[item];
                mass   +=  item_masses[item];
            }
        }

        // Having added all the items, if the current profit exceeds
        // the currently stored best profit, and the volume and mass do
        // not exceed the capacities of the knapsack, store both the
        // items selected and the corresponding profit as the maximum
        // identified so far.
        if (    (profit >  max_profit)
             && (volume <= knapsack_volume_capacity)
             && (mass   <= knapsack_mass_capacity) ) {
          max_profit = profit;
          max_items  = items;
        }
    }
    // Print out the details about the items that were included
    // in the combination that maximized profit while having the
    // total volume and mass stay within the capacities of the
    // knapsack.

    std::cout << "Maximum profit: " << max_profit << std::endl;
    std::cout << "Items added: ";
    double volume{ 0.0 };
    double mass{ 0.0 };

    // The loop variables follow the order (0, 1), (1, 2), (2, 4), (3, 8), ...
    for ( std::size_t item{0}, bit{1}; item < capacity; ++item, bit <<= 1 ) {
        if ( max_items & bit ) {
            std::cout << item << " ";
            volume += item_volumes[item];
            mass   += item_masses[item];
        }
    }

    std::cout << std::endl;
    std::cout << "Total volume: " << volume << std::endl;
    std::cout << "Total mass:   " << mass   << std::endl;
    std::cout << std::endl;
}
```

Online

Once you have tried this yourself, you can look at the complete solutions at `repl.it` with the URLs

> https://repl.it/@dwharder/Insertion-sort-on-indices,
> https://repl.it/@dwharder/Knapsack-maximizing-volume, and
> https://repl.it/@dwharder/Knapsack-maximizing-profit-through-brute-force.