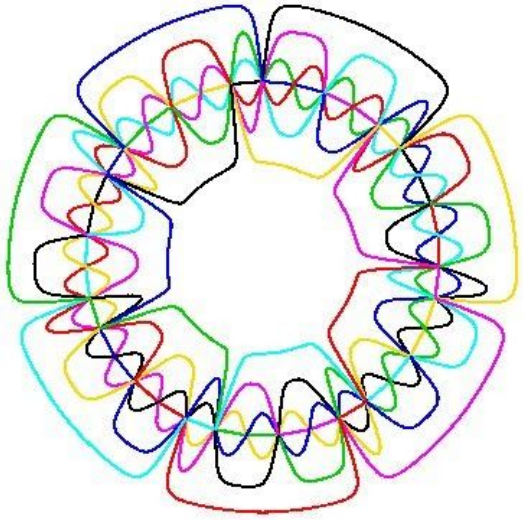
Gattaca loves a power set. Except to enumerate
So there. Don't becurse it. You had better honor it.

A goldfish called Gattaca

Welcome to your first 2C Quest. More freedom. Less direction. More cryptic clues. Less starter code
Many treats in this adventure for Loceff lab lovers

The Subset Sum Problem

previously by Michael Loceff



Here is an interesting puzzle: A radio show host wants to select a group of tunes or commercials from a set whose total running time is as close to the duration of the show or commercial break (e.g. three hour FM music show, or a two minute commercial break) without going over the time allotment. How does he pick items to play?

You can formulate this problem in a way that lets us approach a solution programmatically (why only *approach*, and why *a* solution?)

Consider any set of positive[1] integers, S = {x_1, x_2, ... x_n}. Given a target integer t, we want to identify the subset of S whose sum is as large as possible, but not larger than t.

The Algorithm

Here is what you can call a brute-force method:

1. Form all possible subsets (also called the "power set of S"). For example, if `S = {4, 1, 7}`, the power set of `S` is the set containing `{}` (the empty set), `{4}`, `{1}`, `{7}`, `{4, 1}`, `{1, 7}`, `{4, 7}`, and `{4, 1, 7}`.

2. Find a subset whose members add up to the largest number possible <= t.

It always yields a solution (although it may not be unique - there may be many subsets that add to the same maximal number <= t). It is a very costly technique in terms of timing because it has what we call exponential time complexity (we'll learn what this means soon). The problem is inherently exponential, so there is not much we can do about that.

Nonetheless, we can find improvements to brute-force algorithms by trading off some amount of accuracy and/or completeness for a large improvement in speed.[2] The algorithm you will implement in this quest makes a modest efficiency improvement. It will also explicitly state how

[1] Really, peeps? Is this necessary? Can't I just say *integers*, or *non-negative integers*?
[2] Which of these two are we giving up here? Or is it both or neither?

you iterate through the power set of S. Specifically, it gives you a heuristic with which you can prune your search space by shutting doors early on that you know to not lead to your solution.

Here's how I think about this problem:

Let's say that I can make N subsets out of n items. If I add just one more item, then I can make a whole set of N new subsets (How? I simply form new sets by adding this new item to every set I already have). Each time I complete that, I double the number of sets I have.

So if I started with no sets at all, and then add the empty set, and then after that I want to add the first item, I have two possible subsets: The empty set and the set with just the new item. Then after the second item comes along, I can form two more sets in the same way, bringing the total to 4. Then 8, and 16, and so on. It's easy to see how this is simply the value 2^N for N items. That's the number of subsets I can form from N items. This is also the size of the *power set* of a set.

Now the cool thing is that we can follow this exact technique to iteratively generate our subsets. Start off with an empty set and, iterating through the set of all items, add each item to every set you already have to generate a whole bunch of *new* sets. They're *new* in the sense that you haven't yet seen (or evaluated) any of them.

As you can see, this is a cool way to generate your subsets, but it does nothing for the running time of our search. It's still going to take time proportional to 2^N to visit all the candidate sets.

I think I can hear you say "This is really screaming out for an optimization - Skip elements larger than the target while iterating over the elements" - I think you'll find many more cool rules like that. But remember they are all just special cases of your overarching plan - *to discard unviable candidates as soon as you find them, and thereby prune the search space of all its descendants*.

How? One way is to reject all unpromising searches as soon as we know for sure.

"How do we know for sure?" you ask. Well, you simply keep track of the running total (sum of elements) of each set.

The moment you find a set whose total is greater than the target, you can immediately ignore all its *descendants* without even looking. By *descendants*, I mean every set of which this *unviable* set is a subset.

Suppose you identify one such subset early on... say when you have M items still left to process. That means you've effectively pruned out 2^M potential candidates without needing to explicitly test them. Imagine this - Just two quarters ago, you were mind-blown by the power of binary search over linear. But if you think about it, you'll see that's exactly what's going on here at various levels of intensity. Every time you find an unviable set, all its descendants are gone. Poof! Like that.

Clearly, it makes sense to prune sets early in the process (when M is close to N), right? In the best possible case, you encounter such descendant-killers early in the lineage of a set - when it's only a few items big. I wonder if there's something we can do to the input set to handle each descendant killer as early as possible. Hmm… Well, it looks like there's lots of juicy opportunities to ponder with this one.

Note that every singleton set which is unviable effectively cuts the remaining search space in half. Intuitively this makes sense because it is the same as saying that you first scan all items in linear time and discard elements greater than the target. If you discarded K items, you have thereby just reduced your search space by *up to* 2^K items[3] (how do I estimate this number?). So it seems like this strategy should on average cut the search space by a factor of somewhere between 0.5 and some small number (which should be $1/2^N$). Knowing this you can put some definite upper and lower bounds of performance on this algorithm in your mind: It is at least as efficient as linear search and at most as efficient as binary search. But we stay modest 'cuz we're talking about linear search over a potentially exponential number of items (when?). Hmm… Does that really qualify as linear search? Go figure.

High-Level Plan

Maintain a vector of viable candidates. Initially start out with just an empty set. Then gradually add viable sets to this list as you process each item from the master set in turn. At any time, if you find an exact match to the requested target value, you can output that set and end the search. If you don't find an exact match after exhausting the powerset, output a viable candidate with the greatest sum less than the target.

Implementation Specifics

Create a template class called `Set`. Your list of candidates is now `std::vector<Set>`. However, since your `Set` is a template class, I, as the user of your class will decide what things I want sets of. But to keep things simple on your side, you only need to write the logic to handle sets of integers. As long as the integer operations required by you (e.g. addition) are supported with identical syntax by some other type, you can be blissfully unaware of its other details.

For example, it's up to me to make sure that if I use `SongEntry` as my template parameter, my `SongEntry` class behaves like an integer, in supporting the ability to be added to an integer to yield an integer.

Since your `Set` is going to be a template class, you'll submit a single source file, a file called `Set.h`. It's not a great deal of code, and probably just under a hundred lines. But you'll need to think through the problem carefully to write it.

---

[3] In the best case, I can eliminate half the *remaining* search space with each examination. This will result in a total of 2^K elements you can skip checking (How? It's simply the sum $1 + 2 + \ldots + 2^{K-1}$ - yes? In the worst case, you have to check every element. How do you estimate the average case?)

The Representation

How do you represent a set internally?

At first thought it might seem like a simple problem. The trivial solution is to have each set be a collection (e.g. a vector) of objects.

While this seems like a simple solution on face value, it can be horribly inefficient. What if I wanted to make sets of media objects that were each unmanageably large?

A better solution is to store all possible set members, the so-called Universal or Master set, within the object as a class (static) member. Each instantiated set would then be simply composed of a collection of integers, which are simply pointers to the actual objects in the Master set.

This sounds like a good idea. But there's an even better one. With a static master set, you still have to initialize it by copying over the supplied elements into it. With an understanding that you will not change the master set, why not have your client (the instantiator of the Set class) create and maintain the master set as a vector, and you only deal with integer indices into this (externally maintained) master set?[4]

That's the approach we will take in this quest. Although you don't have to code a lot, most of the time, I presume, may be taken up by simply wrapping your head around this:

- The Set object will contain a pointer to an externally maintained vector of elements we call the master set. This pointer must be set during set initialization.
- Each Set object will contain a vector of integers, each of which is an index in the master set to a valid element contained in that Set. (See Figure 1)

I'm not a Figger. Quit starin' at me

[4] Although the contents of this master list is vulnerable to change from outside (since it actually lives outside the class), trying to overcome this shortcoming by making our own static copy doesn't seem to me to be such a good tradeoff. In the interests of learning the algorithm, we'll assume a certain level of intelligence and self-preservation instinct in the users of your class.

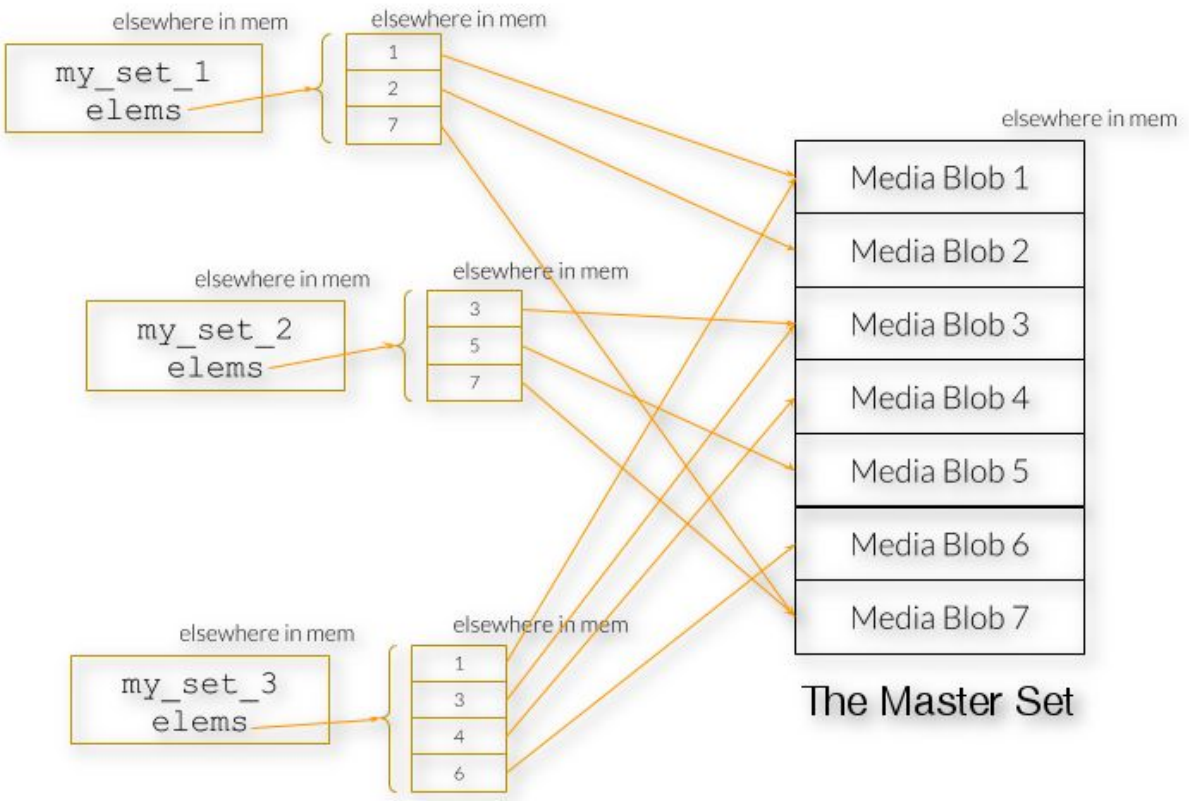


Figure 1. Many sets contain indices into elements stored in the master set

Tips

1. You don't want to iterate over each candidate set to calculate the sum of its elements in real time. Instead, maintain a `sum` member in your `Set` class such that every `Set` object knows its own sum.
2. You'll have a nested loop in your search method: One over all the items in your master vector, and another over all the subsets in your vector of candidates. This inner loop does not have a growing upper limit if you end up adding subsets inside the loop. So make sure that at each pass over the inner loop, you only add direct descendants of the sets you started with. Recall that by *descendants of a set*, I mean all the sets you can obtain by adding one or more extra elements to it. A direct descendant is a descendant that differs from its parent by exactly one element.
3. You can and should test in your inner loop to see if you have added a new sub-list whose `sum() == t` because, if you have, you are done and should break out of the outer loop. Normally there will be zillions of subsets and we can usually reach the target long before we have tested them all.

Figure 2 shows a fuzzy photo of the Set class.

```
template <typename T>
class Set {
private:
    vector<T> *_master_ptr;
    vector<size_t> _elems;  // List of indices into this->master
    size_t _sum;

public:
    Set(vector<T> *mast_ptr = nullptr) : _master_ptr(mast_ptr), _sum(0) {}

    const size_t size() const { return _elems.size(); }
    const vector<T> *get_master_ptr() const { return _master_ptr; }
    const size_t get_sum() const { return _sum; }

    bool add_elem(size_t n); // n is the index in master
    bool add_all_elems();    // Add everything in the master

    Set<T> find_biggest_subset_le(size_t target);

    friend ostream &operator<<(ostream& os, const Set<T> &set) {
        const vector<T> *mast_ptr = set.get_master_ptr();
        os << "{\n";
        for (size_t index : set._elems)
            os << " " << mast_ptr->at(index) << "\n";
        return os << "}";
    }

    friend class Tests; // Don't remove this line
};
```

Figure 2. A fuzzy pic of the `Set` class

Figure 3 shows some code that I use to test your Set. Use yer own.

One important point to keep in mind before forging ahead: In general, you may have multiple sets adding to the same target. How do you know which one is the winning set? It's like lotto. Except you can actually figure out how to get the winning ticket.

Onward to the miniquests... If messages you receive seem too cryptic, see if anybody on our subreddit knows what they mean.

Miniquest 1 - Default constructor

Implement the default constructor for your Set class. It should correctly make an empty set.

Miniquest 2 - The Empty Master

You don't need to do anything here, but if your implementation agrees that an empty master has nothing to give, you get a reward. How's that?

Oh yeah, you can't move forward unless you do. Small detail.

```
vector<int> master;
size_t master_total = 0;
for (size_t i = 0; i < NUM_ENTRIES; i++) {
    int val = rand() % 300;
    master_total += val;
    master.push_back(val);
}

// Select an arbitrary target.
size_t target = master_total/2;

Set<int> master_set(&master);
Set<int> best_subset = master_set.find_biggest_subset_le(target);

cout << "Target = " << target <<endl;
cout << "Best sum = " << best_subset.get_sum() <<endl;
cout << best_subset << endl;
```

Figure 3. Some test code

Miniquest 3 - Non-default constructor

Actually, this one's a freebie too. 'Cuz the fuzzy photo gives you both constructors.

Miniquest 4 - Add 'em all

I'm gonna try and add all the items in the master. If it works, you get a reward! Sweet.

Miniquest 5 - Be Legal

I'm gonna tempt you and test you. You get rewards for staying on the right side of the law!

Miniquest 6 - Nothing and Everything

I think that to make anything, you need to first be able to make nothing.

If you can't make nothing, then you can make nothing.

Take your `find_biggest_subset_le(size_t target)` for instance.

If I invoke it with a target of 0, you must return nothing, the empty set.

If I invoke it with a target it can never meet, then I expect it to give me its all. That means everything.

These may seem like two special cases you can handle separately for easy rewards.

However, whether you treat them as special cases or let them naturally fall out of your search makes all the difference to how easy you'll have it when troubleshooting.

Remember - *Once you can make nothing, you have a way to make anything, including everything.*

Finally, note that `find_biggest_subset_le()` returns a copy of a set - not a reference or pointer. (Why is it *not* a big deal to return a copy here?)

Miniquest 7 - Solve small cases

I'm gonna do a bunch of random moves here. All you have to do is to watch and do exactly the same.

Well, YOU don't have to do anything. Your code will. Let's see if it can dance my small dances.

Miniquest 8 - Solve small song lists

Yeah - Same as before. Let's see if your code can keep up with my disc jockeying.

Miniquest 9 - Solve big cases

Ok. These bigger dances will weed out the slow pokes. I'll see how many of y'all master these moves.

Miniquest 10 - Solve bigger song lists

Or these songs.

Submission

Yeah. This is one of those cool quests where it seems like a lot of work, but really, you only need to do a couple of things right and everything else just slides into place satisfyingly. When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret password for this quest in the box.
3. Drag and drop your `Set.h` file into the button and press it

Wait for me to complete my tests and report back (usually a minute or less).

To get your trophy count recorded, you can include your student id (or email) within a comment (in the required format) in one of the files. You can then check your standing at the /q site.

Some questions to get you started on our sub

1. Does a particular order of processing the master set elements work better than others? (e.g. Does sorting the list of items help?)
2. Does the value of the target matter (i.e. are certain targets likely to be reached sooner)?
3. Does the nature of the probability distribution from which the master set elements are drawn matter? (e.g. Can the algorithm take advantage of the fact if you knew that the master set elements were drawn from a particular probability distribution (e.g. a Gaussian, uniform or exponential distribution, etc.)

Self calibration for 2C

If you complete this first quest of 2C on your own, only looking for information and/or help from your friends (or online) to understand what to do next, you'll get a very good idea of how you will fare in 2C. The remaining quests will be of the same (or gradually increasing) level of difficulty as this one (with an occasional easy treat).

If you enjoyed working on this quest without being frustrated by endless test failures, that means you will have a fantastic time in 2C.

Otherwise, you can still have a fantastic time by investing a little bit of effort before. I suggest you clear the 2B quests first. To get to the 2B quests, you may need to go to a tiger named Fangs. But don't worry. He no bite.

Happy Questing,

&