CS106B

Winter 2023

Practice Final Exam 3

This is a printable version of the third set of practice problems from the practice final exam. It's designed to have the same format and style as the actual CS106B final exam.

Solutions can be found by visiting the Additional Practice Problems resource on the CS106B website and looking up solutions to the corresponding problems.

Problem One: Cosmic Care Packages

At present, it costs around $5,000 to place one kilogram of material into orbit. This means that if you're floating on the International Space Station, you probably shouldn't expect that much out of your meals. They're calibrated to be nutritionally complete and lightweight, and while there's some effort made to provide nice things like "flavor" and "texture," that's not always possible.

Let's imagine that you want to be nice and send a cosmic care package up to the ISS with some sweets for the crew to share. Every penny matters at five grand a kilo, so you'll want to be very sure that what you send up into the vast abyss will actually make the folks there happy. You have a list of the desserts that each individual ISS crew member would enjoy. What's the smallest collection of treats you could send up that would ensure everyone gets something they like?

For example, suppose the ISS crew members have the following preferences, with each row representing one person's cravings:

```
{ "Pumpkin pie", "Revani", "Sufganiyot" }
{ "Castella", "Kheer", "Melktert" }
{ "Po'e", "Tangyuan", "Tres leches" }
{ "Apfelstrudel", "Tangyuan" }
{ "Alfajores", "Brigadeiro", "Revani" }
{ "Baklava", "Revani" }
{ "Melktert" }
{ "Brownies", "Cannolis", "Castella", "Po'e", "Revani" }
{ "Cendol", "Kashata", "Tangyuan" }
{ "Baklava", "Castella", "Kheer", "Revani" }
```

The smallest care package that would cheer up the whole crew would have three items: revani, melktert, and tangyuan. Anything smaller than this will leave someone unhappy.

Write a function

```
Set<string> smallestCarePackageFor(const Vector<Set<string>>& preferences);
```

that takes as input a list containing sets of what treats each crew member would like, then returns the smallest care package that would make everyone on the ISS happy.

While in principle you could solve this problem by listing off all possible subsets of treats and seeing which ones satisfy everyone's sweet tooths, this approach is probably not going to be very fast if there are a lot of different choices for sweets. Instead, use the following approach: pick a person who hasn't yet had anything sent up that would make them happy, then consider each way you could send them something that would fill them with joy.

Some notes on this problem:

- Yep, you guessed it! You have to do this one recursively.
- To clarify, if you send up some item to space, you can assume there's enough of it for everyone on the space station to share. Sending up revani, for example, means sending up enough revani for each crew member to have a piece.
- Each person's preferences will contain at least one item. There is no fixed upper size to how many preferences each person can have.
- If there are multiple care packages that are tied for the smallest, you can return any of them.
- In case it helps, you can assume no treat's name is the empty string.

```
Set<string> smallestCarePackageFor(const Vector<Set<string>>& preferences) {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There’s additional space for your answer on the next page.

Additional space for your answer to Problem One, if you need it.

Problem Two: Slow and Steady Copying

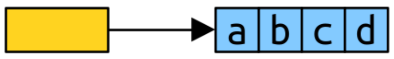
This problem explores an alternate way to implement a vector called a ***backup vector***. In lecture, when we implemented the stack, we used the following strategy:

- Store all the elements in an array.
- When we run out of space in the array, double the array size, copying over all the old elements.

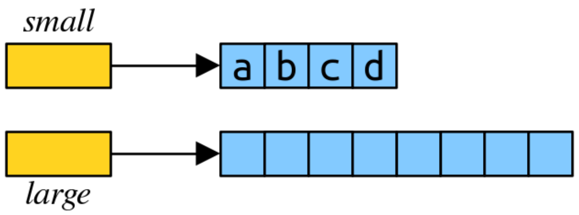
Although we didn't write any code for this in lecture, this is the same way that the `Vector` type is implemented.

This approach works very well in practice, but has one slight drawback. Even though most push operations are fast, every now and then we have to do a huge amount of work to copy over all the elements from the old array to the new one. This then raises a question – is there a way to grow the array without moving everything over all at once?

The backup vector is one possible way to do this. The basic idea is the following: let's imagine that we have a full array of elements, as shown here, and we want to append one more element to the vector:

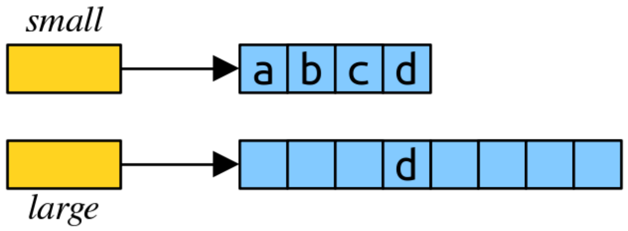


As before, we'll get a new array that's twice as big as our previous one:

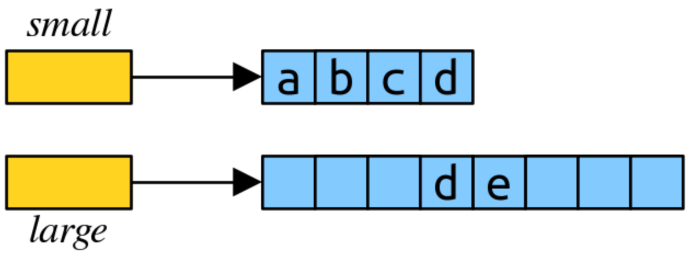


At this point, we have two arrays, the older, smaller array (called the ***small array***) and a newer, larger array called the l***arge array*** that's twice as big as the small array.

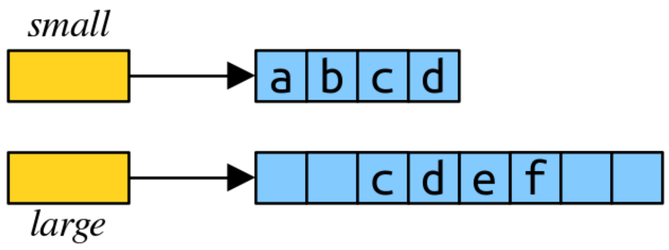
However, here's where things diverge from what we did in class. Rather than copying over all the elements, we'll just copy over the last element from the small array into the new array, as shown here:



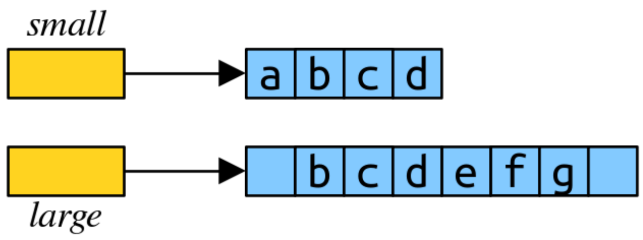
We'll then append our new value to the vector by placing it at the next free slot in the large array, as shown here:



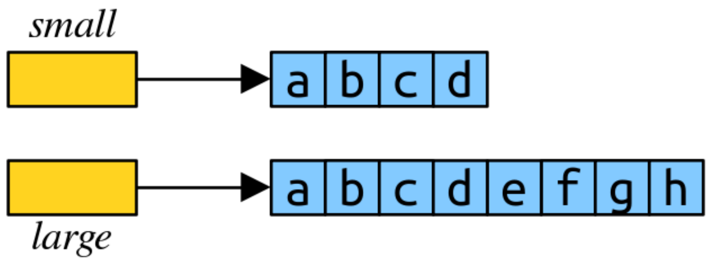
More generally, whenever we append an element to the vector, we'll place it at the next free slot in the large array, then copy over one more element from the small array. For example, here's what happens if we append f to the vector:



Notice that we placed f in the next spot in the large array, then copied the last uncopied element (c) from the small array into the large array. If we append g, we get this result:

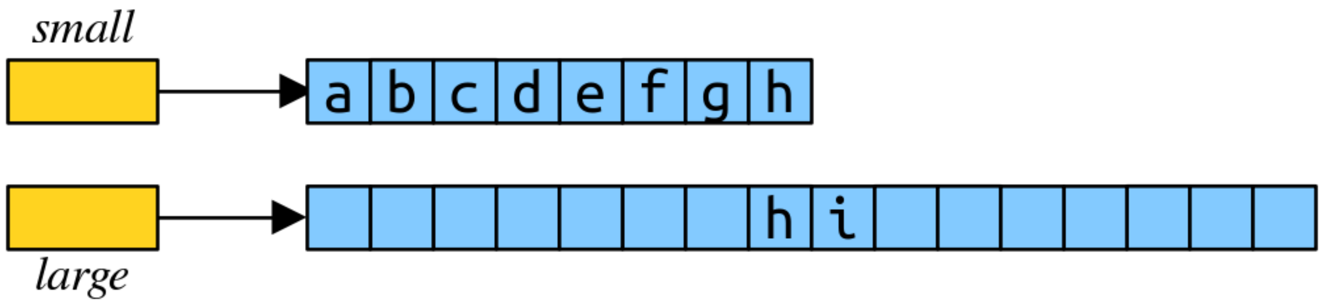


That is, g goes in the next free slot in the large array, and we copy the next uncopied element over from the small array (namely, b). Finally, if we append h, we get this result:



Notice that our large array is now full and has all the elements that used to be in the small array. We've copied everything over, just not all at once.

At this last point, all the elements from the small array have been copied. If we then append one more value to the vector, we can discard it and begin this process again: get a new large array and move a single element over. For example, here's what it would look like if we now appended i:



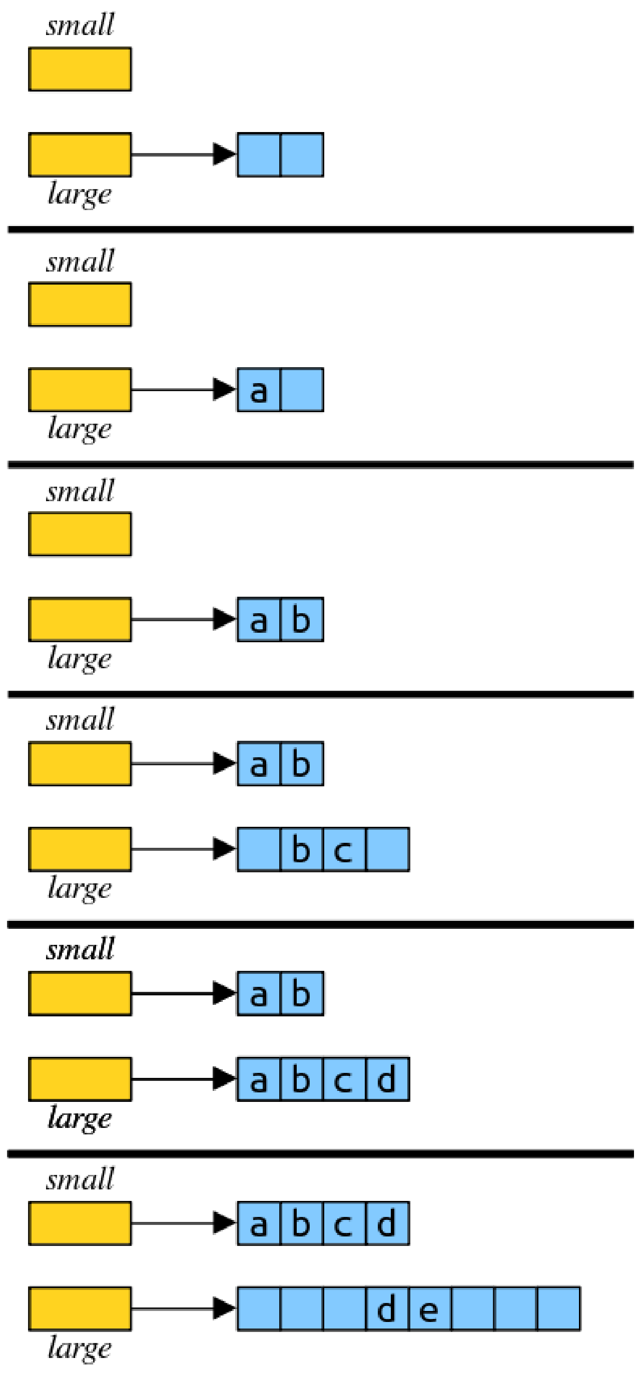
Notice that the old large array is now the small array, and the current large array is a new array twice as big as the previous one.

To summarize:

- At each point in time, we maintain two arrays, a small array and a large array, such that the large array is twice as big as the small array.
- Whenever we append a value, if there's room left in the large array, we place the item at the next free slot in the large array, then copy an element over from the small array. If the large array is full, we deallocate the small array, then allocate a new large array that's twice as big as the old one.

There is one exception to the above rules. Specifically, when the backup vector is first created, we will only have a large array, and there will be no small array. Until that first array fills up, we basically proceed as usual, as though we were working with a regular dynamic array.

As an example, here's what it looks like if we start with an empty backup vector, the insert the a, b, c, d, and e, in that order. This diagram assumes that the initial size of the large vector is two



That leaves the question of how to do lookups. To access the item at some index, you'll need to determine which of the two arrays to look at. Some elements will only be in the large array. Others will be located only in the small array. See if you can find a general pattern you can use to determine where to look.

Your task is to implement the following `BackupVector` class. We've provided the following to you. Do not write anything on this page; you'll implement the class on the upcoming pages.

```
class BackupVector {
public:
    BackupVector();
    ~BackupVector();

    /* Appends a new value to the vector. */
    void append(int toAdd);

    /* Returns the value at the given index. Calls error() if the index is
     * out of bounds.
     */
    int get(int index) const;

private:
    /* The two arrays. */
    int* small;
    int* large;

    /* The rest is up to you! */
};
```

```
/* Use the space here to write out the contents of the private section of the
 * BackupVector class. We've copied what is already there, and you are
 * welcome to add to this.
 */
private:
    /* The two arrays. */
    int* small;
    int* large;

    /* The rest is up to you! */
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work.

```
/* Implement the constructor here. */
BackupVector::BackupVector() {

}

/* Implement the destructor here. */
BackupVector::~BackupVector() {

}
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's more space for your answer on the next page.

```
/* Implement append here. */
void BackupVector::append(int value) {
```

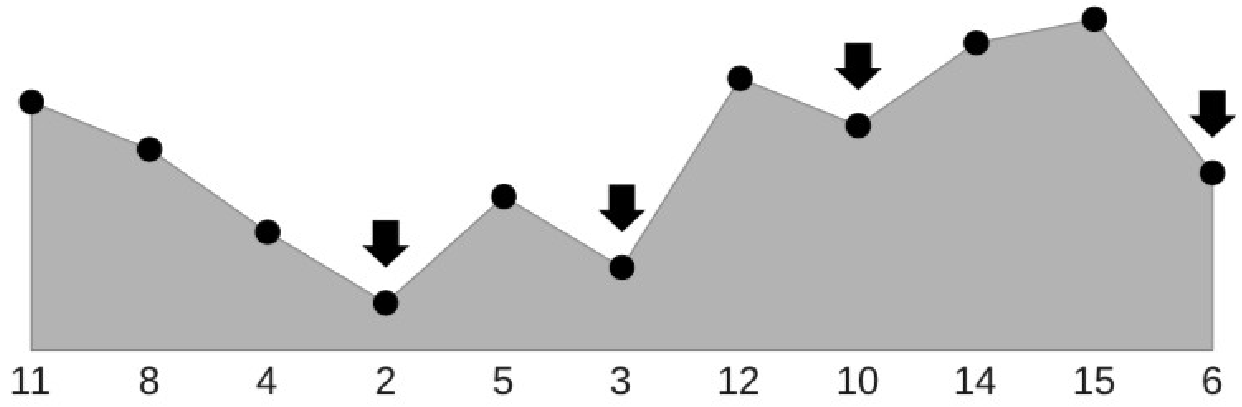
We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's more space for your answer on the next page.

```
/* Implement get here. */
int BackupVector::get(int index) const {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work.

Problem Three: Rainwater Collection

You are interested in setting up a collection point to funnel rainwater into a town's water supply. The town is next to a ridge, which for simplicity we will assume is represented as an array of the elevations of different points along the ridge. When rain falls on the ridge, it will roll downhill along the ridge. We'll call a point where water naturally accumulates (that is, a point lower than all neighboring points) a "good collection point." For example, here is a possible ridge with good collection points identified:



Write a recursive function

```
int goodCollectionPointFor(const Vector<int>& heights);
```

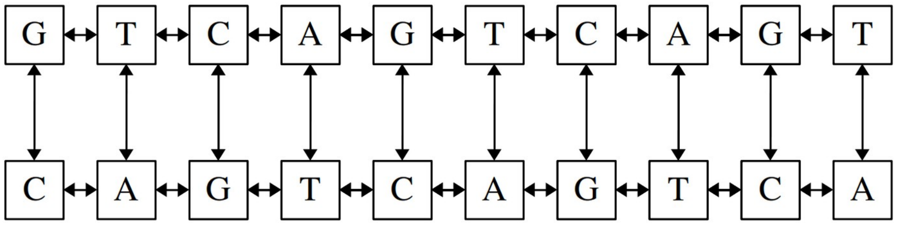
that returns the index of a good collection point. Your solution should run in time O(log *n*). As a hint, think about binary search. You can assume that all elements in the array are distinct. Also, make sure to validate the input. It's okay for heights to be negative, but there has to be at least one element in the heights array.

```
int goodCollectionPointFor(const Vector<int>& heights) {
```

We will grade whatever you write in this box. Write your scratchwork elsewhere.

Problem Four: Complementary Strands

In humans, DNA strands are always paired with a ***complementary strand***. This is a new strand of DNA linked alongside the original strand, except with different nucleotides. Specifically, any time there's an `A` in the initial strand there's a `T` in the complementary strand, and any time there's a `C` in the initial strand there's a `G` in the complementary strand (and vice-versa). Here's an example of what this might look like:



We can represent a double-stranded DNA sequence as a modified doubly-linked list of nucleotides, where each nucleotide links to the nucleotides before and after it in the sequence, as well as to the nucleotide across from it:

```
struct Nucleotide {
    char value;          // 'A', 'C', 'G', or 'T'
    Nucleotide* next;    // To the right
    Nucleotide* prev;    // To the left
    Nucleotide* across; // Vertically, to the other strand.
};
```

For example, in the picture shown above, the nucleotide `G` in the upper-left corner would have its `prev` pointer set to `nullptr`, its `next` pointer set to the `T` nucleotide next to it, and its `across` pointer set to the `C` beneath it. The `C` nucleotide beneath it would have its `next` pointer set to the `A` to its right, its `prev` pointer set to `nullptr`, and its across pointer back to the `G` in the top strand.

Your task is to finish writing a function that works as follows. The function takes as input a single strand of DNA whose `across` pointers have not been initialized. The function then constructs the complementary strand, then links the two strands together by appropriately wiring the `across` pointers of the two strands.

Write a function

```
void addComplementaryStrand(Nucleotide* dna);
```

that takes as input a single strand of DNA whose across pointers have not been initialized. The function then constructs the complementary strand, then links the two strands together by appropriately wiring the across pointers of the two strands.

Some notes on this problem:

- The `across` pointers of each of the nucleotides in the input sequence have not been initialized when this function is called. You should not assume anything about where they point.
- For full credit, your solution must run in time O(*n*), where *n* is the number of nucleotides in `dna`.
- The input pointer will always point to the first nucleotide in the strand, or will be `nullptr` if the input DNA strand is empty.
- You can assume all letters in the DNA strand are either `A`, `C`, `G`, or `T`.
- You ***may not*** use any container types (e.g. `Vector`, `HashSet`, etc.) in the course of solving this problem. This includes the `string` type.

```
void addComplementaryStrand(Nucleotide* dna) {
```

We will grade whatever you write in this box. Use the remaining space on the page for scratch work. There's more space on the next page, if you need it.

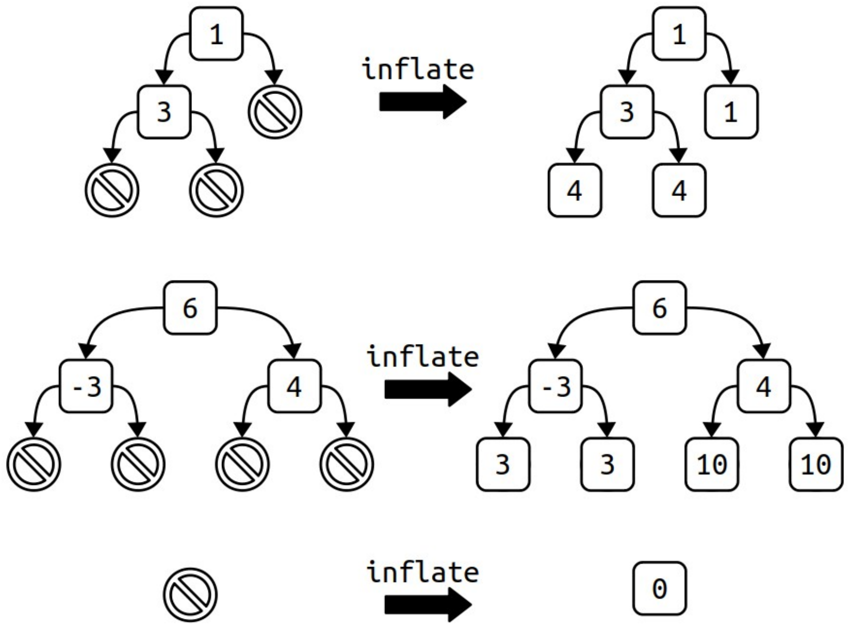
Extra space for your answer to Problem Four, if you need it.

Problem Five: Inflating Trees

Consider the following type representing a node in a binary tree (though not necessarily a binary *search* tree):

```
struct Node {
    int value;
    Node* left;
    Node* right;
};
```

Given a binary tree represented this way, we can ***inflate*** the binary tree by replacing all null pointers that appear in the original tree with new nodes, where each node's value is the sum of the integers on the path from the root of the tree to that node. (One edge case: if the tree is empty, we replace the root with a new node whose value is zero). For example, here's some sample trees and the result of inflating each of them. We've explicitly drawn in the null `left` and `right` pointers on the initial trees to make the transformation clearer. Although the null pointers aren't drawn in on the trees to the right, they are still present.



Write a function

```
void inflate(Node*& root);
```

that takes as input a pointer to the root of a tree (or to `nullptr` if the tree is empty), then inflates the tree. Some notes:

- Your solution does not have to be as efficient as possible, but you should avoid any unnecessary inefficiencies in your solution.
- You ***may not*** use any container types (e.g. `Vector`, `HashSet`, etc.) in the course of solving this problem.

```
void inflate(Node*& root) {
```

We will grade whatever you write in this box. Use the surrounding space for scratch work.