* Due: Friday, November 22nd, 11:59pm
* Directory name for this homework (case sensitive): hw5
  + This directory should be in your hw\_username repository
  + This directory needs its own README.md file
  + You should provide a Makefile to compile the code.
* Warning: Start early!

**Problem 1: Heaps (5%)**

1. Draw the tree structure for the array contents CHEMISTRY
2. Explain why this is a minHeap
3. Drawn the resulting tree after add(D)
4. Starting from the resulting tree from part 3, draw the tree after one call to remove()

Place your answer in hw5.txt. If you would like to use a different file type, simply indicate in your README what the file is named.

**Problem 2: Backtracking (25%)**

We all know that some classes are a lot of work, and at the end, you have very little actual knowledge to show for. Others are the other way around. Some are neither work nor useful, and in some (CSCI 104), you work really hard and hopefully learn a lot. In this problem, you are to figure out which combination of classes cause you to learn the most, without exceeding your workload capacity.

Your source file should be learning.cpp, and compile into a file learning. That program should take just one command line parameter, the name of the input file. The input file will have two numbers on the first line: *n* is the total number of classes from which you can choose (we promise that this number will never exceed 25), and the second number *W* is a double, capturing the total amount of work you can do for the semester. Next come *n* lines, each describing a class. The class *i* is described by first the class name *s(i)* (a string that does not contain any white space), followed by one or more whitespaces. Next come two non-negative doubles *w(i)* and *L(i)*. The first of these is the amount of total work you need to do for class *i*, and the second is the total amount you learn from class *i*. Here is an example of what this may look like:

4 3.14159

CSCI104 3.001 10.0

CSCI170 2.41 8.0

ENGR102 0.10 0.36

CSCI280 0.66 2.15

Your goal is to output the maximum total amount you can learn from any combination of classes whose total work does not exceed *W*. In the example above, the correct output would be 10.36, by combining CSCI104 with ENGR102. You cannot combine CSCI104 with any of the other classes (it would exceed your workload capacity), and you don't learn as much if you take CSCI170 with CSCI280; also, CSCI170 and ENGR102 and CSCI280 together exceed your workload capacity. The output above would simply look as follows:

10.36

By far the easiest solution to this problem will use recursion and backtracking, and in fact, we will give full credit only to such solutions. Notice that your solution only has to work for *n* up to 25; we would be surprised if your solution were able to solve *n* exceeding 40 in a reasonable amount of time. (If it can, that may be a sign that it is incorrect.)

When we type make learning, it should produce an executable called learning in your hw5 directory.

**Problem 3 (Create a d-ary Heap, 35%)**

Build your own templated d-ary MinHeap class with the interface given below. Put your entire implementation in heap.h (because the class is templated). You learned in class how to build a binary MinHeap, where each node had 2 children. For a d-ary MinHeap, each node will have d children.

template <typename T>

class MinHeap {

public:

MinHeap (int d);

/\* Constructor that builds a d-ary Min Heap

This should work for any d >= 2,

but doesn't have to do anything for smaller d.\*/

~MinHeap ();

int add (T item, int priority);

/\* adds the item to the heap, with the given priority.

multiple identical items can be in the heap simultaneously.

Returns the number of times add has been called prior to this

call (for use with the update function).\*/

const T & peek () const;

/\* returns the element with smallest priority.

If two elements have the same priority, use operator< on the

T data, and return the one with smaller data.\*/

void remove ();

/\* removes the element with smallest priority, with the same tie-breaker

as peek. \*/

void update (int nth, int priority);

/\* finds the nth item (where nth is 0-based) that has ever been added

to the heap (the node that was created on the nth call to add),

and updates its priority accordingly. \*/

bool isEmpty ();

/\* returns true iff there are no elements on the heap. \*/

private:

// whatever you need to naturally store things.

// You may also add helper functions here.

};

In order to build it, you may use internally the vector container (if you so choose). You can not use anything else from the STL.

In order to guide you to the right solution, think first about the following questions. We strongly recommend that you start your array indexing at 0 (that will make the following calculations easier). In order to figure out the answers, we suggest that you create some examples and find a pattern.

1. If you put a complete d-ary tree in an array, what is the index of the parent of the node at position i?
2. In the same scenario as above, what are the indices of the children of the node at position i?
3. What changes in the heap functions you learned in class when you move to d-ary arrays?

**Problem 4 (A\* Search, 35%)**

The word game "Doublet" was invented by Lewis Carroll, and is a word transformation puzzle. Two words of identical length are given. The objective is to transform the first word into the second word by forming successive words of the same length, changing only one letter at a time. Here is an example from HEAD to TAIL:

HEAD

HEAL

TEAL

TELL

TALL

TAIL

The challenge is to do the transformation in the least number of words.

Your source file should be doublet.cpp, and the executable should be called doublet. It takes three command line parameters. The first indicates the starting word, the second indicates the ending word, and the third is a file which contains a list of valid words. So you might run the program as follows:

./doublet head tail words.txt

Everything should be case-insensitive, so there is no difference if the starting word is HEad or heAD.

The word file will be formatted as follows:

7

head

heAl

hem

Tail

tell

taLL

teal

The first row contains a number n, indicating the number of words in the file. There will be n more rows, each with a single word, each possibly followed by some whitespace. There may be blank lines at the end of the file. You may assume the file is formatted properly. We may give your program very large word files (around 1 million words).

The word file will not contain duplicates, and the start and end word will always be in the word file. But, as mentioned above, there may sometimes be no way to get from the starting word to the ending word.

You will implement the A\* search algorithm to quickly find the shortest transformation. You can think of each word in the word file as a node, and there is an edge between two words of the same length if they differ by exactly one letter.

We encourage that you build the graph explicitly while you read the word list. One hint to keep in mind: if for each word you read, you loop through all words to see if they differ in exactly one letter, your algorithm will be very slow: Ω(n2). Instead, you are much better off creating all possible words that you can form by replacing one letter at a time, then looking up whether they are actually words (and if so, where they are in your word list). Notice that that may require some more sophisticated data structures. We will definitely give you several test cases with very large word lists, so if you have an Ω(n2) algorithm for building your graph or for running A\* search, you are guaranteed to lose a bunch of points.

When we type make doublet, it should produce an executable called doublet in your hw5 folder.

**Review the A\* algorithm**

Recall that A\* makes the move with smallest f-value. f = g + h where g = distance (number of moves made) from the start state while h is a score produced by a heuristic evaluation of the move. We will use the following heuristic:

Incorrect Letters: counts the number of letters in the current word which do not match the letter in the same position in the ending word. So if you are currently at DATA and your final word is SALT, then your heuristic will evaluate to 3.

**Implementation Details**

You will need to use your heap from the previous problem to attain the desired runtime. You may not use the STL priority\_queue on this problem.

It is possible to find a node via two different paths, and you'll need to update that node's priority in the heap. To do this, you'll need each node to remember the order it was added to the heap, so you can use the update function.

To add consistency to your solution, you should break ties in the following manner:

1. Always make the move with smallest f-value.
2. If multiple words have the smallest f-value, choose the one with the smallest h-value (or, equivalently, the largest g-value).
3. If multiple words have the smallest f and h-value, choose the smaller string according to operator<(str1, str2), where str1 and str2 are fully-capitalized versions of the original strings.

To accomplish item 2, you will want to calculate the priority as f\*(n+1)+h, where n is the length of the word you are transforming. Since h can never be larger than n, this properly chooses the smallest f-value while breaking ties according to h-value.

To track how well your algorithm is performing, you should keep track of the number of **expansions**, that is, the number of words your algorithm considers. Every time you remove the min-value word from your MinHeap, you are considering that word, and you should increment the number of expansions. The starting word should increment the number of expansions (from 0 to 1), but the ending word should not.

**Output**

You should output the following two lines **exactly** to cout:

<steps>

<expansions>

Where <steps> is the fewest number of transformations to get from the start word to the end word, and <expansions> is the number of expansions your algorithm required to find the solution.

If there is no way to get from the starting word to the ending word, you should instead just print

No transformation

<expansions>

**An Example**

Starting from AAAAA, ending at BBBBB, with the following dictionary:

AAAAA

AAAAB

AAABB

AABAA

AABBA

AABBB

ABBBA

BAAAA

BBBBA

BBBBB

1. Expanding AAAAA, we would add AAAAB, AABAA, and BAAAA to the heap.
2. Expanding AAAAB (because it comes first according to operator<), we would add AAABB.
3. Expanding AAABB, we would add AABBB.
4. Expanding AABBB (because it has the smaller h-value), we would add AABBA.
5. Expanding AABAA (because it has the smaller f-value), we would update AABBA.
6. Expanding AABBA, we would add ABBBA.
7. Expanding ABBBA, we would add BBBBA.
8. Expanding BBBBA, we would add BBBBB.
9. BBBBB would be the next expansion, so we're done with a total of 8 expansions (we never searched BAAAA).

It would output:

5

8

**Chocolate Problem: A "Better"(?) Heap (1 Chocolate Bar)**

The standard (Min-)Heap Implementation we learned about implements the functions insert, find-min, and remove-min to run in time O(log n) each. Here, you will analyze an implementation that implements insert to run in worst-case time O(1), while the other two functions will run in **amortized** time O(log n). The data structure for this resembles something called log-structured Merge Trees, so you may want to look them, and amortized analysis, up in the course notes as you try to solve this problem.

Our implementation will be based on forests of binary trees. As opposed to the complete binary trees we use for regular heaps, these binary trees will be a little unbalanced, on purpose. They are defined inductively as follows: B(0) is just a single leaf node (no edges). B(k) is one root note with k children. Of these children, one is the root of a B(0), one the root of a B(1), one the root of a B(2), ..., and one the root of a B(k-1). We recommend that you draw the first few B(k) trees to gain a feel for what they look like. As a first question that will be quite useful down the road, you may want to answer/prove: "How many nodes total does a B(k) contain?"

In our heap implementation, we will have multiple B(k) trees, some for different values of k, some for the same. Each of the B(k) trees will contain elements of our heap in each node, and each will satisfy the heap property, i.e., the elements in the children of a node v are always no smaller than the element in node v itself. However, no promises are made how the elements in different trees as part of the data structure compare.

One of the key operations is the combination of two B(k) trees. If you have two B(k) trees, each of which satisfies the heap property, you can easily merge them into a B(k+1). Simply make the root of one of them the child of the root of the other. Which one becomes the child of which? Obviously, you make the one with the larger element in its root the child of the one with the smaller element in its root.

**Prove that when you do this, you indeed get a B(k+1), and that it satisfies the heap property.**

Finally, we are ready to specify the heap operations:

* insert (x): Create a new B(0) tree, and put x in it. That's it. Done!
* cleanup: Starting with the smallest size k=0, while there is some k such that we have two (or more) B(k) trees, use the operation from the previous paragraph to replace them with a B(k+1) tree. Do this until there are no more pairs of trees with the same k value.
* find-min: First run cleanup. Then, check the roots of all remaining trees, and return the smallest element over all the roots.
* delete-min: First run cleanup and find-min to find the smallest element. Say it is a root of a B(k). Delete that element, resulting in child subtrees B(0), B(1), ..., B(k-1). Simply add all of these subtrees to the data structure (possibly resulting in duplicates again).

**Prove that this implementation makes insert run in worst-case time O(1) (Yeah! You probably got that one!), and that find-min and delete-min run in amortized worst-case time O(log n).**

Here are the two key steps you probably want to consider for your analysis:

1. After you have run cleanup, what's the maximum number of trees you can still have left (as a function of n, the total number of elements in your heap)?
2. Design a credit scheme to properly and formally account for everything that is going on. What is your credit invariant?