Avery Wang and Anna Zeng
CS106L Spring 2020

April 22, 2020
Credits to Ali Malik for creating WikiRacer!

Assignment 1: CS106L WikiRacer

Screenshots Due: Wednesday, 29th April, 11:59pm
Assignment Due: Wednesday, 13th May, 11:59pm

Introduction:

Human beings are obsessed with finding patterns. Whether it be in the depths of mathematical study or the playground of hobbies like Chess and Sudoku, it is clear that the human race enjoys engaging with the art and science of discovering trends. It is actually claimed by some that our ability to recognize the most complex of patterns was one of the central factors in our development as an ultra intelligent species. Yet with the recent advances in data mining and information distribution, we have reached an age where the sheer volume of data easily overwhelms our ability to immediately understand it. This is where you, as computer programmers, come in. Just as the advances in computing have allowed us to gather such egregiously large collections of data, so too have advances in algorithm design and programming methodology pushed the boundaries of the complexity of patterns we can detect. With the advent of fields like data science, we are now able to inspect and analyze trends beyond those capable by our minds alone.

One interesting place to look for meaningful trends is the vast collection of articles on Wikipedia. For example, there is a famous observation that repeatedly clicking the first, non-parenthesised/italicised link starting on any Wikipedia page will always get you to the Philosophy page. This fact, popularised by an xkcd comic's hover caption, led a team of mathematicians from the University of Vermont to conjecture that the flow of information in the world's largest, most meticulously indexed collection of human knowledge tends to pages like Philosophy because the subject matter of this field is a major organizing principle for the ideas represented by human knowledge. Trends like this, where a collection of information conceives a web of emerging relationships, can give enormous insight into the structure of human knowledge and understanding.

One fun game to play is Wikiraces, where any number of participants race to get to a target Wikipedia page by using links to travel from page to page. The start and end page can be anything but are usually unrelated to make the game harder. Before the timer starts, you are allowed some time to read the target page to get a better understanding of it. If you want to have a try, there is an online version here!

Although usually just a fun pastime, looking at different Wikipedia ladders can give a lot of interesting insights into the relationship and semantic similarity between different pages. In this assignment, you are going to implement a bot that will intelligently find a Wikipedia link ladder between two given pages. In the process you will get practice working with iterators, algorithms, templates, and special containers like a priority queue.

A broad pseudocode overview of our algorithm is as follows:

```
To find a ladder from startPage to endPage:
    Make startPage the currentPage being processed.

    Get set of links on currentPage.

    If endPage is one of the links on currentPage:
        We are done! Return path of links followed to get here.

    Otherwise visit each link on currentPage in an intelligent way
    and search each of those pages in a similar manner.
```

To simplify the implementation and allow for some testing, we will do this in two parts. In part A, you will write the function that gets the valid set of links from the current page (i.e. the greyed out step in the pseudocode). Then in part B, you will write the intelligent search algorithm that actually finds the ladder.

Preliminary Task:

There are two preliminary steps you'll need to do to set up your program. Only the screenshots are due next week, but we highly encourage you to finish the file reading by next week as well to keep on track.

Screenshots

First is a really quick part that just involves taking a few screenshots and emailing them to us. This will let us test how your computer is dealing with internet connectivity, which you can imagine is going to be important!

The starter code contains three projects: InternetTest, WikiRacerLinks, and WikiRacer. For this step, just open the InternetTest project in Qt Creator and run it. This should prompt you with a console with a bunch of text. Every time the console asks you to "take screenshot and press enter to continue", take a screenshot, and press enter! There should be 4 screenshots in total. If you get any compiler errors, or anything strange, please screenshot those too. At the end, **please fill out this form** (if the link doesn't work: https://forms.gle/M1ztFtLzsSUbTEwB7) with all of your screenshots, and that's it :)

NOTE: This form is due on Wednesday, April 29! This is so that if any issues come up, we will have enough time to patch them up.

File Reading

In any program, a key part of programming is testing your code. In this assignment, testing will require reading in input files and outputting the corresponding results, so that you can compare them with the sample results.

You may have noticed that the starter code contains both WikiRacerLinks and WikiRacer as projects. In Part A, you will be writing a function `findWikiLinks` that you will subsequently use as part of the `findWikiLadder` function in Part B. Even though `findWikiLinks` is technically just a helper function for `findWikiLadder`, we want you to write Part A in WikiRacerLinks so that you can test it individually. Once you are satisfied that your function from Part A is working correctly, you can then copy it into your actual WikiRacer program and test your complete algorithm in Part B. We've hence provided you with some example input files to test on for both programs:

In WikiRacerLinks (i.e. test files for Part A):

- There are some test files in the res folder that you can find the links of and compare with the expected output files. Start with the smaller files like simple.txt and simple-invalid.txt before moving up to the bigger ones!

In WikiRacer (i.e. test files for Part B), under the res folder you'll see:

- input-small.txt - contains only one test pair of words. Should return almost immediately in your final program.
- input-big.txt - contains a more robust set of test word pairs. Might take a few minutes to run in your final program.
- sample-outputs.txt - contains the expected outputs for some sample runs (including for the above inputs) to compare against your program's output.
- random.txt - a file where you can write your own input test cases if you'd like.

Your preliminary task is to implement file reading for both WikiRacerLinks and WikiRacer (in their respective main.cpp files). **Both exercises should require <10 lines of code each.** We have already provided the code to read in a filename from the user in both programs. Your task is to create a filestream from the filename, and then process the file data appropriately for the given program as follows:

WikiRacerLinks: Concatenate the contents of the file into a single string of data, and pass that string into findWikiLinks as the page_html parameter. Finally, you will need to print the result of findWikiLinks so that you can compare your output with the sample output.

WikiRacer: The input files are formatted as follows: the first line contains the number of input pairs in the file, and each subsequent line consists of one input pair (i.e. two words, a start_page and end_page, separated by a space). See input-small.txt or input-big.txt for an example. You can assume files will be formatted correctly. For each input pair, call findWikiLadder (which returns a dummy value for now, but which you will implement later) and append its result to outputLadders. Our starter code will then print out the contents of outputLadders.

For this preliminary part, findWikiLadder will always return the vector {"File reading works!", start_page, end_page}. If you've read the file, called findWikiLadder, and appended to outputLadders correctly, then you should see all of the file's input pairs appear in the vectors printed to the console.

Implementation tips:

- Remember to avoid mixing cin and getline!
- Depending on how you implement reading in values, you may end up needing to convert a string to an integer. To do so, you can use the `stoi(line)` function, which takes in a string `line` and returns the integer it represents.
- For this assignment, you don't need to handle the case that the user inputs an invalid filename. (Since you are your own user in this case, make sure that the filenames you type in are valid!)
- [Optional extension] Implement error checking! (It's the same code for both WikiRacerLinks and WikiRacer.) If the user types in an invalid filename (i.e. the filestream fails on opening it), then re-prompt the user until they type in a valid filename.

If you have any questions, definitely let us know on Piazza. You got this! :)

Part A:

Note: *This part should be done in the main.cpp file in the WikiRacerLinks project. Don't write this in the InternetTest project - that part is only for the screenshots.*

Congratulations on completing file reading! Let's take a step back now and return to our algorithm.

As you can imagine, a really important part of our code for this assignment will involve taking a Wikipedia page's HTML and returning a set of all links on this page. Part A of the assignment will be to implement a function

```
unordered_set<string> findWikiLinks(const string& page_html);
```

that takes a page's HTML in the form of a string as a parameter, and returns an unordered_set<string> containing all the ***valid Wikipedia links*** in the page_html string. For those who don't remember, an unordered_set behaves exactly like a set but is faster when checking for membership (in the Stanford library it is called a HashSet).

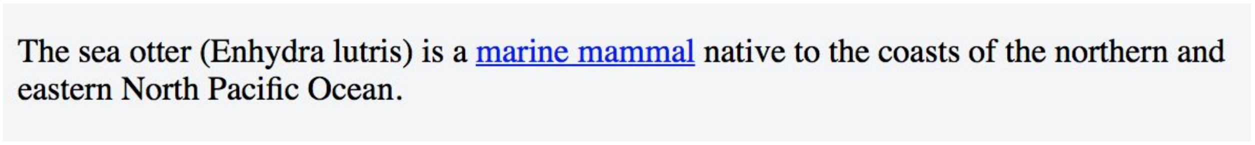
HTML:

First a quick overview of HTML. HTML (Hypertext Markup Language) is a language that lets you describe the content and structure of a web page. When a browser loads a webpage, it is interpreting the content and structure described in the HTML of the page and displaying things accordingly.

We won't go into exceptional detail over how HTML works but we will discuss how hyperlinks are formatted. Here is how a section of HTML on a Wikipedia page might look:

```
<p>
The sea otter (Enhydra lutris) is a
<a href="/wiki/Marine_mammal">marine mammal</a> native to the
coasts of the northern and eastern North Pacific Ocean.
</p>
```

which would display the following:



The sea otter (Enhydra lutris) is a marine mammal native to the coasts of the northern and eastern North Pacific Ocean.

Here we can see that to specify a piece of text as hyperlink, we have to surround it with the <a href="target"></a> tag. This would look like this:

```
<a href="link_path">link text</a>.
```

Valid Links:

For our intents and purposes, a valid Wikipedia link is any link satisfying these two properties:

1. The link is of the form `"/wiki/PAGENAME"`.
2. The PAGENAME doesn't contain any of the disallowed characters (# or :).

The aim of this method will be to find all valid Wikipedia links of this form in the input html string and return an unordered_set containing just the PAGENAME part of the link.

A quick example might clarify this. Let's say we pass in the following string to our function:

```
<p>
In <a href="/wiki/Topology">topology</a>, the <b>long line</b> (or
<b>Alexandroff line</b>) is a
<a href="/wiki/Topological_space">topological space</a> somewhat
similar to the <a href="/wiki/Real_line">real line</a>, but in a
certain way "longer". It behaves locally just like the real line, but
has different large-scale properties (e.g., it is neither
<a href="/wiki/Lindel%C3%B6f_space">Lindelöf</a> nor
<a href="/wiki/Separable_space">separable</a>). Therefore, it serves
as one of the basic counterexamples of topology
<a href="http://www.ams.org/mathscinet-getitem?mr=507446">[1]</a>.
Intuitively, the usual real-number line consists of a countable
number of line segments [0,1) laid end-to-end, whereas the long line
is constructed from an uncountable number of such segments. You can
consult
<a href="/wiki/Special:BookSources/978-1-55608-010-4">this</a> book
for more information.
</p>
```

In this case, our function would return an unordered_set containing the following strings:

```
    Topology
    Topological_space
    Real_line
    Lindel%C3%B6f_space
    Separable_space
```

Note two things of interest here. Firstly, the function would not return the links highlighted as red because they are not valid Wikipedia links. In the first case it is not of the form /wiki/PAGENAME and in the second case, the link contains the invalid character `':'`. Secondly, note that the `Lindelöf` link seems to have weird percentage signs and letters in its hyperlink. This is how the HTML handles non-standard ascii characters like `'ö'` so you should just leave it as it is. Don't worry about cleaning it up!

Additional Rules:

For this assignment, we're going to mandate that you **don't** use indices or the builtin string methods like string.find to do the searching through the text. Instead, look at algorithms like **std::search** and **std::find** that operate on iterators.

The reason for this is twofold. Firstly, we want you to get practice using iterators and algorithms and since the string methods deal with indices, you will be missing out on this learning opportunity. Secondly, arguments are made that the string methods exist only for legacy reasons. Few other containers provide their own specialised find function and string only does this because it existed before the STL was worked into the C++ standard library.

Implementation Tips:

- The code we'll be learning over the next few STL lectures will be really helpful for this part of the assignment. Specifically, pay attention to the countOccurrences method from lecture, which will sequentially look through a text for instances of a string. You'll use a similar approach here.

- Don't forget to test your function using the test files in the res folder! See the File Reading section in the Preliminary Task section for more information on how to test.

- Our solution uses the following algorithms:

    - std::search   // find the start of a link
    - std::find   // find the end of a link
    - std::all_of   // see if link contains invalid characters

  We would definitely recommend you try your best to see where you can leverage these. The comments above indicate how we used them.

If you want to discuss your plan of attack or discuss where an algorithm might be useful, definitely make a private post on Piazza! We look forward to seeing what you come up with :)

Note: by the end of Part A, you should've completed everything in the **WikiRacerLinks** program, and been able to test your findWikiLinks function.

Part B:

Note: *This part should be done in the main.cpp file in the WikiRacer project, not the WikiRacerLinks project! (You'll also be copying your code from Part A into the wikiscraper.cpp file in the WikiRacer project - see instructions later down.)*

Congratulations on finishing the first part of the assignment! You have now implemented a function

```
unordered_set<string> findWikiLinks(const string& page_html);
```

that takes the html of a page in the form of a string as a parameter and returns an `unordered_set<string>` containing all the valid Wikipedia links in the page_html string.

In this next part, we are going to write the code to actually find a Wikipedia ladder between two pages. We will be writing a function:

```
vector<string> findWikiLadder(const string& start_page,
                                        const string& end_page);
```

that takes a string representing the name of a start page and a string representing the name of the target page and will return a `vector<string>` that will be the link ladder between the start page and the end page. For example, a call to `findWikiLadder("Mathematics", "American_literature")` might return the vector that looks like `{Mathematics, Alfred_North_Whitehead, Americans, Visual_art_of_the_United_States, American_literature}` since from the Mathematics wikipedia page, you can follow a link to the Alfred_North_Whitehead page, then follow a link to the American page, then the Visual art of the United States page, and finally arrive at the American_Literature page.

We are going to break the project into steps:

Designing the Algorithm

We want to search for a link ladder from the start page to the end page. The hard part in solving a problem like this is dealing with the fact that Wikipedia is *enormous.* We need to make sure our algorithm makes intelligent decisions when deciding which links to follow so that it can find a solution quickly.

A good first strategy to consider when designing algorithms like these is to contemplate how you as a human would solve this problem. Let's work with a small example using some simplified Wikipedia pages.

Suppose our start page is **Lion** and our target page is **Barack_Obama**. Let's say these are the links we could follow from the Lion page:

- Middle_Ages
- Federal_government_of_the_United_States
- Carnivore
- Cowardly_Lion
- Subspecies
- Taxonomy_(biology)

Which link would you choose to explore first? It is fairly clear that some of these links look more promising than others. For example, the link to the page titled Federal_government_of_the_United_States looks like a winner since it is probably really close to the Barack_Obama page. On the other hand, the Subspecies page is less directly related to a page about a former president of the United States and will probably not lead us anywhere helpful in terms of finding the target page.

In our algorithm, we want to capture this idea of following links to pages "closer" in meaning to the target page before those that are more unrelated. How can we measure this similarity?

One idea to determine "closeness" of a page to the target page is to count *the number of links in common between that page and the target page*. The intuition is that pages dealing with similar content will often have more links in common than unrelated pages. This intuition seems to pan out in terms of the links we just considered. For example, here are the number of links each of the pages above have in common with the target Barack_Obama page:

Pages	**Links in common with Barack_Obama page**
Middle_Ages	0
Federal_government_of_the_United_States	5
Carnivore	0
Cowardly_Lion	0
Subspecies	0
Taxonomy_(biology)	0

This makes sense! Of course the kind of links on the Barack_Obama page will be similar to those on the Federal_government_of_the_United_States page; they are related in their content. For example, these are the links that are on both the Federal_government_of_the_United_States page and the Barack_Obama page:

- Democratic_Party_(United_States)
- United_States_Senate
- President_of_the_United_States
- Donald_Trump
- Vice_President_of_the_United_States

Thus, our idea of following the page with more links in common with the target page seems like a promising metric. Equipped with this, we can start writing our algorithm.

The Algorithm

In our code, we will be doing something very similar to the old 106B WordLadder assignment (if you took the class previously), except instead of using a normal queue, we will use a priority queue. A priority queue is a data structure where elements can be enqueued (just like a regular queue), but the element with the highest priority (determined by a priority function) is returned on a request to dequeue. This is useful for us because we can enqueue each possible page we could follow and define each page's priority to be the number of links it has in common with the target page. Thus, when we dequeue from the queue, the page with the highest priority (i.e. the most number of links in common with the target page) will be dequeued first.

In our code, we will use a vector<string> to represent a "link ladder" between pages, where pages are represented by their links. Our pseudocode looks like this:

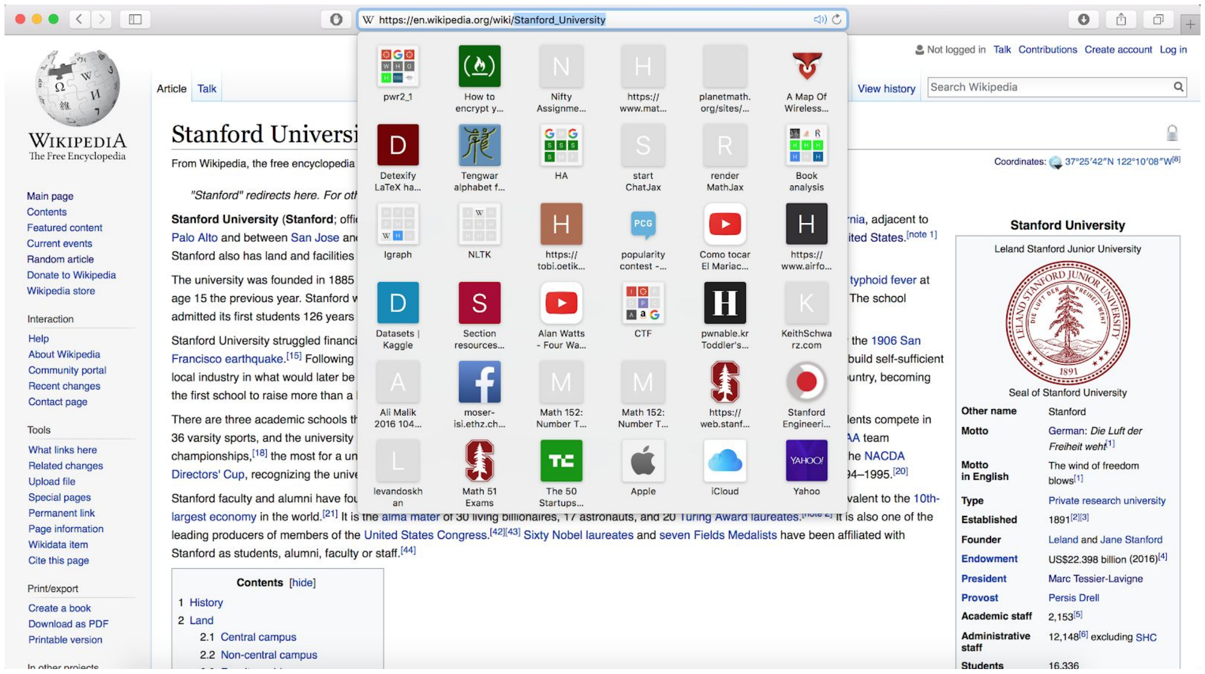
```
Finding a link ladder between pages start_page and end_page:

    Create an empty priority queue of ladders (a ladder is a vector<string>).

    Create/add a ladder containing {start_page} to the queue.

    While the queue is not empty:

        Dequeue the highest priority partial-ladder from the front of the queue.

        Get the set of links of the current page i.e. the page at the end of the
          just dequeued ladder.

        If the end_page is in this set:
            We have found a ladder!
            Add end_page to the ladder you just dequeued and return it.

        For each neighbour page in the current page's link set:

            If this neighbour page hasn't already been visited:

                Create a copy of the current partial-ladder.

                Put the neighbor page string at the end of the copied ladder.

                Add the copied ladder to the queue.

    If while loop exits, no ladder was found so return an empty vector<string>
```

Using the WikiScraper Class

To assist you with connecting to Wikipedia and getting the page html, we have provided a WikiScraper class. It exports the following public method:

```
unordered_set<string> WikiScraper::getLinkSet(const string& page_name);
```

which takes a string representing the name of a Wikipedia page and returns a set of all links on this page. Throughout this assignment, we will take the name of a Wikipedia page to be what gets displayed in the url when you visit that page on your browser. For example, the name of the Stanford University page would be **Stanford_University** (note the _ instead of spaces):



Stanford Universi

From Wikipedia, the free encyclopedia

"Stanford" redirects here. For ot

Stanford University (**Stanford**; offi ... nia, adjacent to Palo Alto and between San Jose an ... ited States.[note 1] Stanford also has land and facilities

The university was founded in 1885 ... typhoid fever at age 15 the previous year. Stanford w ... The school admitted its first students 126 years

Stanford University struggled financi ... the 1906 San Francisco earthquake.[15] Following ... build self-sufficient local industry in what would later be ... untry, becoming the first school to raise more than a

There are three academic schools th ... lents compete in 36 varsity sports, and the university ... AA team championships,[18] the most for a un ... he NACDA Directors' Cup, recognizing the unive ... 94–1995.[20]

Stanford faculty and alumni have fou ... valent to the 10th-largest economy in the world.[21] It is the alma mater of 30 living billionaires, 17 astronauts, and 20 Turing Award laureates.[note 2] It is also one of the leading producers of members of the United States Congress.[42][43] Sixty Nobel laureates and seven Fields Medalists have been affiliated with Stanford as students, alumni, faculty or staff.[44]

Contents [hide]
1 History
2 Land
2.1 Central campus
2.2 Non-central campus

Coordinates: 37°25′42″N 122°10′08″W[8]

Stanford University

Leland Stanford Junior University

Seal of Stanford University

Other name	Stanford
Motto	German: *Die Luft der Freiheit weht*[1]
Motto in English	The wind of freedom blows[1]
Type	Private research university
Established	1891[2][3]
Founder	Leland and Jane Stanford
Endowment	US$22.398 billion (2016)[4]
President	Marc Tessier-Lavigne
Provost	Persis Drell
Academic staff	2,153[5]
Administrative staff	12,148[6] excluding SHC
Students	16,336

1

In part A of this assignment, you wrote most of the code that implements the functionality of the `getLinkSet` method. The WikiScraper class adds a bit more functionality on top of the `findWikiLinks()` method you wrote to avoid redundant work, but otherwise completely relies on `findWikiLinks()` to work properly. Your first task is to copy your code from the `findWikiLinks()` method you wrote for part A and replace the unimplemented `findWikiLinks()` method in the wikiscraper.cpp file. Once this is done, the WikiScraper class is complete and you can use it for the rest of the assignment.

To use the class, you will first need to make a single WikiScraper object in the `findWikiLadder()` method that you are implementing in main.cpp. This would look something like this:

```
vector<string> findWikiLadder(const string& start_page,
                                        const string& end_page)
                                        {

    // creates WikiScraper object
    WikiScraper scraper;

    // gets the set of links on page specified by end_page
    // variable and stores in target_set variable
    auto target_set = scraper.getLinkSet(end_page);

    // ... rest of implementation
}
```

Note: Do not create more than one WikiScraper object; doing so will slow your code down. Just create one at the start of the function and pass it around wherever it is needed.

Creating the Priority Queue

The next task in the assignment is to make a priority queue using a constructor from the standard library. Although this sounds like a simple task in theory, it will require you to really understand how to use lambdas and variable capture.

The first thing to do is read the documentation for `std::priority_queue`. The format of a `std::priority_queue` looks like this:

```
template<
   class T,
   class Container = std::vector<T>,
   class Compare = std::less<typename Container::value_type>
> class priority_queue;
```

Let's break this down. This is telling us the `std::priority_queue` needs three template types specified to be constructed. We can read the documentation further to see what each one represents:

Template parameters

T	-	The type of the stored elements. The behavior is undefined if T is not the same type as Container::value_type. (since C++17)
Container	-	The type of the underlying container to use to store the elements. The container must satisfy the requirements of SequenceContainer, and its iterators must satisfy the requirements of RandomAccessIterator. Additionally, it must provide the following functions with the usual semantics: ■ front() ■ push_back() ■ pop_back() The standard containers std::vector and std::deque satisfy these requirements.
Compare	-	A Compare type providing a strict weak ordering

So we essentially need to specify the type of thing the priority_queue will store (T), the container the priority_queue will use behind the scenes (which will be a vector<T> for our purposes), and finally, the type of our comparison function that will be used to determine which element has the highest priority.

Since we want a `priority_queue` of ladders, where we represent a ladder as a `vector<string>`, we will take `T` to be `vector<string>`, and so the Container, which is of the form `vector<T>`, will be a `vector<vector<string>>`. Lastly, we need to determine what comparator function we want to use. Remember, we want to order the elements by how many links the page at the very end of its respective ladder has in common with the target_page. To make the `priority_queue` we will need to write this comparator function:

```
To compare ladder1 and ladder2:
    page1 = word at the end of ladder1
    page2 = word at the end of ladder2
    int num1 = number of links in common between set of links on
                page1 and set of links on end_page
    int num2 = number of links in common between set of links on
                page2 and set of links on end_page
    return num1 < num2
```

The thing to keep in mind is that this function will need to have access to the WikiScraper object you made in your findWikiLadder() method so that it can get the set of links on page1 and page2. Think about how you can write this comparison function as a lambda that can access the WikiScraper object in the function where the lambda is declared.

Equipped with this, we can create the priority_queue, which takes three template parameters: the thing to store (ladder), the container to use behind the scenes (vector<ladder>), and the ***type*** of the comparison function we will use. We hit a little snag here, since if we write our comparison function as a lambda, we have no idea what its type is. To deal with this, C++ provides the `decltype()` method which takes an object and returns its type. Then, to the constructor of the priority_queue, we actually just pass the comparison function. All in all, we can create our priority_queue like this:

```
vector<string> findWikiLadder(const string& start_page,
                                    const string& end_page)
                                    {
    // creates WikiScraper object
    WikiScraper scraper;

    // Comparison function for priority_queue
    auto cmpFn = /* declare lambda comparator function */;

    // creates a priority_queue names ladderQueue
    std::priority_queue<vector<string>, vector<vector<string>>,
                        decltype(cmpFn)> ladderQueue(cmpFn);

    // ... rest of implementation

}
```

Implementation Tips:

- We would *strongly* suggest you print the ladder as you dequeue it at the start of the while loop so that you can see what your algorithm is exploring.

- Don't forget to test your code using the test files in the res folder! You can compare your output with the sample runs in the sample-outputs.txt file. See the File Reading section in the Preliminary Task section for more information on the test files.

- For convenience, the following lists the expected output of the pairs in the input-big.txt file in the res folder:

    - **Start page**: Fruit
    - **End page**: Strawberry
    - **Expected return**: {Fruit, Strawberry}
    - **Notes**: This should return almost instantly since it is a one link jump.

    - **Start page**: Milkshake
    - **End page**: Gene
    - **Expected return**: {Milkshake, Barley, Gene}
    - **Notes:** This ran in less than 60 seconds on our computers.

    - **Start page**: Emu
    - **End page**: Stanford_University
    - **Expected return***:
      {Emu, Encyclop%C3%A6dia_Britannica_Eleventh_Edition, Cornell_University, Stanford_University}
      or
      {Emu, Beetle, University_of_Florida, Florida_State_University, University_of_California,_Berkeley, Stanford_University}
    - **Notes**: This ran in less than 60 seconds on our computers.

  *These two ladders result from differences in the order of valid links in the unordered set from Part A. If you generate ladders that are longer than the solution ladder, but the number of common links matches the first step of the solution given here, you should be fine! Feel free to post to Piazza if you have any questions or want to double-check.

  You don't need to match these ladders exactly (as long as your ladder is the same length as the sample) but your code should run in less than the times specified. If not, double-check: are you using references where needed? Can you remove any redundant variables? Are you missing any steps in the ladder algorithm? Most often, slowness tends to come from either a small algorithm error or inefficient design choices in the lambda.

- Definitely consult the lecture on lambdas to see how you can make the comparator function on the fly. In particular, you will need to leverage a special mechanism of lambdas to capture the WikiScraper object so that it can be used in the lambda.

If you want to discuss your plan of attack, definitely make a private post on Piazza! The code for this assignment is not long at all, but it can be hard to wrap your head around. Ask questions early if things don’t make sense; we will be more than happy to talk through ideas with you.

Good luck! :)