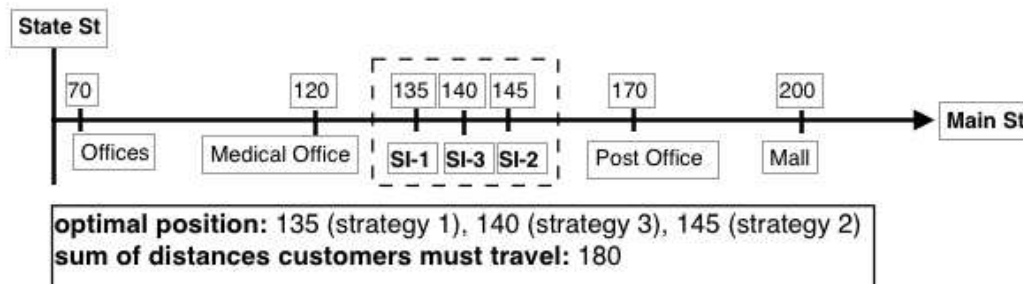


### 1 Problem

You have been hired to help determine the placement for a new Donut Delite store. There are several store fronts available along Main Street. Developers are trying to determine the best placement of a store based on the distance to several large office buildings. They want to locate the store so that it minimizes sum of the distances from the store to the offices on the street. Additionally, they would like to use the application to determine the location of any new store. (Ignore the fact that each building could have a different number of people.) The diagram below shows locations chosen by several strategies. Numbers indicate distances to the intersection of Main and State. The “SI-N” text means store locations identified by the numbered strategies.



You were asked to consider the following strategies to determine the best location of the new store (sometimes there might be more than one best location).

#### 1. Midpoint Distance Strategy

Let  $x$  be the distance of the business closest to State Street. Let  $y$  be the distance of the business farthest from State Street. Locate the new store at the mid-point between  $x$  and  $y$ ; that is, at location  $(x + y)/2$ . For the above example, this strategy produces the location 135 and the sum of the distances to this location is  $65 + 65 + 15 + 35 = 180$ .

#### 2. Median Distance Strategy

Sort the distances and choose the location of the middle element. If the number of elements is even, choose the midpoint between the two middle elements. For the above example, this strategy produces 140, and the sum of the distances to this location is  $75 + 55 + 25 + 25 = 180$ .

#### 3. Average Distance Strategy

Locate the store at the average of the distances. For this example, this strategy produces 140, and the sum of the distances to this location is  $70 + 60 + 20 + 30 = 180$ .

The problem is that only one of these strategies *always* provides the *best* possible sum of distances.

## 2 Problem-solving Session - (15%)

In a team of students, do the following:

1. Create a test case of 5 business office locations between 0 and 10 miles from State St. The test case should demonstrate that only one of the strategies is always optimal. Work through each strategy and indicate the calculated store location and the sum of the distances to each business, versus the optimal choice.  
Hint: If you think of the offices as distributed on a see-saw, how would you make it unbalanced?  
For each strategy your team should work in parallel to:  
(a) identify the optimal new store location;  
(b) compute the sum of the distances from this location to each office.  
Combine your answers, make a table of the results, and **identify the optimal strategy**. Team members should collect their answers, compare results, and list the results for each strategy.
2. Develop code that implements the optimal strategy. Given a list of the unsorted locations, write a function that returns the best location. (Assume that there is an already-written list sort function, and call it.)
3. Write another function that, given a list of unsorted locations and the best location, calculates and returns the sum of the distances between the best location and each other location.
4. Study the **quick\_select** algorithm on the next page.  
Run the algorithm for  $k := 3$  on the following list.

[5, 2, 9, 22, 1, 3, 17, 12, 4, 13]

Show all the intermediate steps and the final solution.

## 2.1 Quickselect Algorithm

There is an algorithm, known as quickselect, that runs fast without sorting.

The quickselect approach is more efficient than sorting techniques seen so far. Quickselect finds the  $k^{th}$  **smallest number** in an *unsorted* list of numbers. Study the algorithm below and consider how you could use quickselect to compute the median and solve the store location problem.

```
Function quick_select( a_list, k ):
  If a_list is not empty:
    pivot <- Choose the element at position ( len( a_list ) // 2 )
    smaller_list <- All elements of a_list smaller than pivot
    larger_list <- All elements of a_list larger than pivot
    count <- the number of occurrences of pivot value in a_list
    m <- the size of smaller_list
    If k >= m and k < m + count then:
      return pivot
    If m > k:
      return quick_select( smaller_list, k )
    Otherwise:
      return quick_select( larger_list, k - m - count )
```

It is a pre-condition that  $k$  is in the range 0 to  $\text{len}(\text{a\_list})-1$ .

### 3 Post-PSS - (10%)

For the Post-PSS complete the following tasks on *your own*.

The goal is to complete the first program that solves the problem.

1. Create a project for this assignment. There will be multiple source files.
2. Extract and copy the quicksort module from the lecture's zip file. Here is the link to the lecture materials:  
<https://www.cs.rit.edu/~csci141/Lectures/08/>
3. Create a Python file named `tools.py`. This module will contain functions that will be shared by other program modules.
4. Implement these reusable functions in `tools.py`:
  - A function to read the file and return the list of business locations; and
  - The function from problem-solving that, given a list of unsorted locations and the best location, calculates and returns the sum of the distances between the best location and each other location. The `abs()` function is useful to get the absolute value of the distance between two locations.

An example of input file content is shown below. Return only the list of numbers.

```
Offices 70
MedicalOffice 120
PostOffice 170
Mall 200
```

5. Create the main program file named `store_location.py`. This module will contain a main and a function to compute the optimal location using quicksort. **You may not use either the builtin `list.sort()` function or the `sorted()` function.** Import the `quick_sort.py` module into `store_location.py` (do NOT copy the content). Then import the `tools.py` module into `store_location.py`, and write the function to compute the optimal location using quicksort. Finally, write the main function so that it reads the file, computes the optimal location, calculates the sum of differences, and prints the results as shown in the example below. The `store_location.py` must use the functions you wrote in the `tools.py` file.

Here is example of the output for the post-PSS:

```
Enter data file: test_data_writeup.txt
Optimum new store location: 145.0
Sum of distances to the new store: 180.0
```

Zip `store_location.py` module and the `tools.py` module into `postpss5.zip` and submit the zip to the post-PSS dropbox in MyCourses by the deadline. Read the rest of this assignment carefully to understand and then implement the remaining requirements.

## 4 Implementation - (75%)

The assignment involves two programs that solve the problem different ways. The *two programs* implement the same strategy and are instrumented so that they measure and report their time performance.

First, add instrumentation to the `store_location.py` program that you wrote in the post-PSS. The instrumentation will measure time of execution.

The operations to measure are the calculation of the optimal location, plus the sum of distances. Do not include the time to read the file and make the data list.

After completing instrumentation of the `store_location.py` program, copy that file to a new file named `select_median.py`. In this module, you will implement the **quick\_select** algorithm to solve the problem, and replace the sorted approach.

Here is a summary of *what you will complete and deliver* for this assignment:

1. `tools.py`: the module you wrote in post-PSS containing the functions that are reusable by both main program modules.
2. `store_location.py`: the application you wrote in the post-PSS implementing the sorted approach, to which you added instrumentation;
3. `select_median.py`: an instrumented program implementing the *quickselect* algorithm; and
4. `report.txt`: a file containing *your name* and answers to the questions at the end of this section.

### 4.1 Program Operation and Instrumentation

Each program has similar structure, processes the same input files, and produces the same kind of results. Both implementations prompt for the input file name. Each program calculates and prints the optimal location of the store and the sum of the distances that people would have to travel to reach the store from each given location.

Each implementation must have instrumentation that reports the elapsed time used to sort and select the location. The Python `time` module has a `time.perf_counter()` function that returns the current number of fractions of a second used by the process as it runs. Use this time module function to capture a start time and a finish time; the difference is the elapsed time.

To test performance, download the set of large test cases and unzip to get and use the cases:

<http://www.cs.rit.edu/~csci141/Labs/05/TestData.zip>

Run each implementation several times with this data file. Measure how much time it takes to find the median *after* reading the file and *before* printing results. List your time comparisons for each implementation. Be patient; these runs may take some time.

Remember that, when a data set has an even number of elements, the optimal location is midway between the two middle numbers. Computing the midpoint requires identifying both middle values of the collection, as the strategy stated in the beginning.

**You may not use either the builtin `list.sort()` function or the `sorted()` function.**

## 4.2 A Small Example

Below is an example of running the sorted dataset program that solves a small version of this problem. This example is really too small to reveal anything however.

```
Enter data file: test_data_writeup.txt
Optimum new store location: 145.0
Sum of distances to the new store: 180.0

elapsed time : 3.845399999979904e-05
```

## 4.3 Larger Examples

This example uses a 100,000 element data set.

```
Enter data file: test_dataset100K.txt
Optimum new store location: 50100.5
Sum of distances to the new store: 2508114488.0

elapsed time : 0.29255081399999994
```

And this example uses the `test_dataset1M.txt` data set of 1 million business locations.

```
Enter data file: test_dataset1M.txt
Optimum new store location: 50128.5
Sum of distances to the new store: 25020348525.0

elapsed time : 3.2213172912597656
```

## 4.4 Questions

Answer the following questions on quickselect and program performance as you design, implement and test your programs. This will show that you understand how quickselect works and how to measure execution time in a program. (hint: to analyze the algorithm, apply it by hand to a small, unsorted data sequence.)

1. What is the purpose of the `pivot`? Do you really need to choose the middle item as the pivot? Would the algorithm have behaved differently if you had selected the first item as the pivot each time?
2. Quickselect is versatile because it can find the  $k^{th}$  **smallest number**. What test cases are needed to fully exercise quickselect? List several cases that exercise all aspects of the quickselect function. Remember: *a test case identifies specific input values and expected outputs*.
3. What is the elapsed time performance of your quickselect program when it processes a large data set? (See link below). What is the elapsed time performance of your median program when it processes the same large data set? Enter the elapsed times of *several runs* of each program in your report.

#### 4.5 Grading

Your grade will be based on the following *individually-developed work*:

1. 15%: The utility module, `tools.py`;
2. 20%: The sorting median program, `store_location.py`;
3. 20%: The quickselect median program, `select_median.py`; and
4. 20%: The `report.txt` file containing *your name* and answers to the questions reporting each program's performance.

#### 4.6 Submission

Submit all four required components in a zip file called `lab05.zip` and submit the zip file to the MyCourses dropbox for this assignment. **Use zip only. Do not use another compression mechanism.**