Lab 2: Benchmarking Binary Heaps

Overview

In class, we reviewed a data structure called a binary heap. Binary heaps dynamically track the smallest (or largest) item in a collection as items are added and removed. Heaps have applications in partial sorting, scheduling problems, and caches.

In this lab, you're going to benchmark the operations of the Python heapq library. Implement a min heap in this lab.

Instructions

1. Design Benchmarks

All the following operations should be implemented from scratch without using the heapq library. Plan out how you intend to benchmark the following operations:

- heapify: create a heap from a collection. Note that this is not the same procedure as min-heapify but rather the build min heap procedure.
- heappush: push an element to the heap and restore the heap property if necessary.
- Implement a function that creates a heap by adding each element one by one, i.e. invoke heappush n times for n elements.
- heappop: pop and return the smallest item from the *heap*, maintaining the heap invariant.
- heapsort: sort a list of numbers using the heapsort algorithm.

2. Perform the Benchmarks

Implement and run the benchmarks.

3. Validating Formal Run Times

Use the regression techniques from lab 1 to estimate the big-o complexity for each of the heapq operations.

4. Reflection Questions

1. Do your empirically determined run times match the theoretical run times?

2. Does it matter whether you create the heap in one pass using heapify or add elements one-by-one?

3. Based on the theoretical run times, describe two ways you could use heaps to find the k smallest items in a collection of n items. Provide the pseudocode and calculate the formal run times for each approach.

Submission Instructions

Save the notebook as a HTML or PDF and upload it to Canvas. The pseudocode and analysis for question 3 of part 4 can be submitted as a separate PDF.