Class Activity 2: Benchmarking Merge Sort

CS 3851 Algorithms

Overview

You are going to implement the merge sort algorithm. You will then benchmark the run time of merge sort under the best, worst, and average case scenarios. You will plot the run times and interpret the plot in relation to the asymptotic run time. Furthermore, you will compare the run times of merge sort and insertion sort for three different scenarios.

Implement Merge Sort

You have been provided the file `test_sorting.py`, which contains unit tests designed to help you check the correctness of your implementation. In the file `sorting.py`:

- Add the definition and implementation of the following function:
  - `void merge_sort(lst, p, r)` – The function takes a Python list and sorts it in place.
    - `p` is the start index of the incoming list, i.e. `0`
    - `r` is the end index of the incoming list, i.e. `len(lst) - 1`

You can test your code by running:

```
$ python test_sorting.py
```

Benchmark Merge Sort

Next, you are going to benchmark merge sort. Although we can derive the run time formally and express the run time in big-oh notation, benchmarking the algorithm can help us develop intuition. We are going to benchmark merge sort on lists of numbers. We will evaluate three cases: list is randomly permuted, list is already sorted, and list is already sorted in reverse order.

Use the following skeleton for your benchmarks:

```
times = []
for i in range(n_trials):
      # DO LIST SETUP HERE
      start = time.clock()
      merge_sort(lst)
      end = time.clock()
      elapsed = end - start
      times.append(elapsed)
```

Note that `time.clock()` returns in the time in seconds. You can use the `mean()` function from NumPy to calculate the average run times.

In a Jupyter notebook, import: `sorting`, `matplotlib`, `numpy`, `random`, and `time` modules.

Benchmark merge sort with lists of `100; 1,000; 10,000;` and `100,000` numbers. Create the lists from random numbers using `random.random()`. Use `random.shuffle()` to generate a random permutation of the list and the `list.sort()` method to sort the list in order and reverse order (using the `reverse` keyword). Run each benchmark 3 times (trials).

Use `Matplotlib` to plot the list size versus the run time using a line plot. There should be one line for each scenario (randomly permuted, sorted, and reverse sorted). Label each line and create a legend. You may need to take the $\log_{10}$ of the list sizes to get a nice graph.

Benchmark merge sort vs insertion sort with lists of `100; 1,000; 10,000;` and `100,000` numbers for three separate cases: list is randomly permuted, list is already sorted, and list is already sorted in reverse order. Create the lists from random numbers using `random.random()`. Use `random.shuffle()` to generate a random permutation of the list and the `list.sort()` method to sort the list in order and reverse order (using the `reverse` keyword). Run each benchmark 3 times (trials).

Use `Matplotlib` to plot the list size versus the run time using a line plot. Create three separate plots for each scenario (randomly permuted, sorted, and reverse sorted) displaying benchmarking comparisons between insertion sort and merge sort. Label each line and create a legend. You may need to take the $\log_{10}$ of the list sizes to get a nice graph.

Analyze Merge Sort Run Time

Big-oh notation (e.g., O(n)) gives an upper-bound on the run time of an algorithm relative to the size of its input. In this case, we will be sorting lists of numbers, and n refers to the number of items in the list. The run time is specified in terms of the number of instructions executed.

Algorithms are evaluated in three cases: the best, worst, and average cases. Each of the cases is associated with a property of the input data.

Answer the following questions in your notebook:

1. Were there any differences between the run times of the three cases for merge sort? What inputs gave the best-, worst-, and average-case runs time?

2. Why do you think the different inputs caused different run times for merge sort?

3. Look up the best-, worst-, and average-case run time of merge sort in big-oh notation and provide those. Are the formally determined run times consistent with your benchmarks?

4. Were there any differences between the run times of the three cases for insertion sort vs merge sort?

5. Why do you think the different inputs caused different run times for insertion sort vs merge sort? Why do you think the best-case run time of insertion sort is faster than merge sort?

6. Are the formally determined run times (in terms of big-oh) consistent with your benchmarks?