Week 5 Lab

Collaboration level 0 (no restrictions). Open notes.

1 Sorting lower bound and Counting Sort

1. Consider a comparison-based **searching** algorithm that takes as input an array of n elements and a key, and returns an index i where $A[i] = key$ if such an index exists, and -1 otherwise. Consider its corresponding decision-tree.

   (a) How many leaves does a decision tree have to have?

   (b) Using the same argument as the lower-bound for sorting in the decision-tree model, come up with a lower bound for (comparison-based) searching. Express it as a theorem and sketch the justification below.

   Theorem:

   Proof: *(We expect a couple of lines).*

2. Suppose that we were to rewrite the last for loop in Counting-sort as: for j=1 to A.length (instead of: for j=A.length down to 1). Would the algorithm sort properly? What would change?

3. Assume you have n elements in the range $\{-20, -19, ..., ..., 19, 20\}$. How would you modify Counting-sort to sort this array?

4. Assume you have $n = 1,000$ integers in the range $\{0, .., 50\}$ and you need to sort them. Would you use Counting-sort or Quicksort, and why?

5. Describe one scenario when Counting-sort is more efficient than Quicksort.

6. Describe one scenario when Quicksort is more efficient than Counting-sort.

7. Describe one scenario when you cannot use Counting-sort.

2 Selection

1. Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Spell out the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms on terms of n and i.

   (a) Sort the numbers, and list the i largest.

   (b) Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.

   (c) Use a SELECT algorithm to find the ith largest number, partition around that number, and sort the i largest numbers.

Sorting in practice

On the class website you will find three files, `python-insertionSort.ipynb, python-mergesort.ipynb` and `python-quicksort.ipynb`. They are not pure Python sources, but are in a format called a Jupyter Notebook. Jupyter Notebook is a web-based interactive computational environment for creating notebook documents. A Jupyter notebook contains *cells*, where cells can be Python code or text. Basically a notebook contains Python code interleaved with comments and plots, so the whole experience of programming is interactive.

How to run a Jupyter notebook:

- Go to JupyterLab (`https://jupyter.org`).On the website, you'll see two buttons: one that says *Install JupyterLab*: you could download and install JupyterLab on your computer, but don't install it just yet because it's slow (feel free to do it later); Another button says *Try it in your browser*. You can use this, and upload your notebook in the browser.

- Another option is to login in to Bowdoin (`login.bowdoin.edu`) and use Bowdoin's JupyterLab server.

- Another option is to VisualStudio has an extension for Jupyter Notebooks, which you could install.

Once you are able to run the notebooks step through them cell by cell: read the explanations, run the code and generate the plots. While you go through the notebook, make sure to use its features and interact with the code: make changes, re-run the cells and see what happens. After stepping through all notebooks you should have seen how the sorts that we discussed in class perform in practice and compared to each other, and how seemingly small changes in the implementation lead to significant differences in running time. Overall the results will validate everything we leant in class so far and the idea that once you know the theory of these algorithms you can usually predict practice.