CHAPTER

THIRTYNINE

LECTURE 21 — SORTING

39.1 Overview

- Sorting is a fundamental operation
- Provides good practice in implementing and testing small functions
- Leads to a better understanding of algorithm efficiency
- Allows us to consider the fundamental notion of a merge of two sorted sequences.
- During testing, we will see an example of the important notion of passing functions as arguments.

39.2 Algorithms to Study

- Insertion sort
- Merge sort
  - This is our primary focus.
- Python's built in sort

39.3 Insertion Sort

- Idea:
  - If we already have a sorted list and we want to insert a new value, we can shift values one location higher until we find the proper location for the new value
  - Insert the new value
  - Start with a just a list of length 1 and repeat until all values have been inserted
- Algorithm

```
for each index i in the list, starting at 1 do
    Save the value stored at location i in variable x
    Initialize j at location i-1
    while j is non-negative and the location to insert x has not been found do
        Shift the value at location j up to location j+1
        Decrement j
    Insert the value stored in x in location j+1
```

- Code (in-class exercise):

```
def ins_sort(v):
```

39.4 Steps to Testing

1. Re-read and mentally simulate
2. Insert print statements and/or view with debugger to see what it is actually doing.
3. Run on "test cases" that capture challenging conditions:
   - Empty list
   - Singleton list
   - List of repeated values
   - List where the largest value is at the beginning or the smallest value is at the end.

39.5 Rough Analysis of Time Required

- For any particular value of `i` in the outer `for` loop, there can be up to `i-1` comparisons/shifts.
  - When `i==1` this is not much, but
  - When `i==N-1`, this is a lot.
- Adding across the different values of `i`, this results in at most (roughly) $N^2/2$ comparisons.
- We write this as $O(N^2)$ because (informally) the number of comparisons done is proportional to N^2.

39.6 Experimental Analysis

- The code `lec21_test_sort.py` posted on the Piazza site, and attached to the end of these notes, will be used to do timing experiments on all the sorts we write.
- Makes use of the `random` module
- Includes two main functions:
  - `run_and_time`
  - `generate_local_perm`

  We will discuss each of these in turn.
- The sorting functions themselves are functions in the module `sort`
- Notice that the sorting function is *passed as an argument* to `run_and_time`:
  - First time that we have passed a function as an argument to another function!
- We will start with experiments to analyze selection sort (see textbook) and insertion sort.

39.7 Breaking the N-Squared Barrier

- The fundamental problem with both selection sort (discussed in the textbook, but not in these notes) and insertion sort:
  - We need to do up to N comparisons by scanning through the list to find the proper location of the next value in the sorted list.
  - For insertion sort, we could use binary search to find the insertion location, but we would still have up to N shifts of values.
- Do better than selection sort and insertion sort by using algorithms that don't scan the entire list to assign one value.
- Examples
  - Quick sort
  - Heap sort
  - Merge sort
- We'll study merge sort, in part because it is the easiest of these to understand and in part because of the importance of the idea of a merge.

39.8 Merging Two Sorted Lists

- Given two lists that are each already sorted, our problem is to generate a new sorted list containing all of the items from both lists.
- For example,

```
L1 = [ 9, 12, 17, 25 ]
L2 = [ 3, 5, 11, 13, 16 ]
```

  must be merged into a new list containing

```
[ 3, 5, 9, 11, 12, 13, 16, 17, 25 ]
```

- Idea:
  - Since both lists are sorted, the first item in the new list must be the first item in one of the lists!
  - If we "remove" the smallest item (3 in `L1` in this case), the next item will again be the first non-copied item in one of the two lists!
  - We repeat this process until one of the lists has no more items to copy.
  - Then, copy the remainder of the other list to the back of our new list.
- We don't actually remove the items from `L1` or `L2`. Instead we keep an index to the next location of `L1` and `L2` that has not yet been copied.
- We'll write the code in class, starting from here

```
def merge(L1, L2):
    i1 = 0
    i2 = 0
    L = []
```

```
    return L
```

- Studying the solution:
  1. Write the values of the index variables, `i1` and `i2`, each time through the loop for lists `L1` and `L2` above.
  2. What are the values of `i1` and `i2` when the loop terminates?

39.9 Merge Sort

- Key observation: all lists of length 1 are sorted
- Therefore, for a list of length N that is to be sorted:
  - Create N lists of length 1 from the values in the list
  - Start to merge these "singleton" lists in pairs to create longer, sorted lists.
  - Repeat on pairs of longer lists in succession
- Requires
  - Keeping a list of sorted sublists, initialized with each singleton list
  - Rather than deleting the sorted sublists, just keep track of which we need to work on.
- Code (in class):

```
def merge_sort(v):
    if len(v) <= 1:
        return
```

39.10 Analysis of Merge Sort

- Check for correctness
- We'll give an informal analysis explaining why there are only $O(N \log N)$ comparisons.
- Experimental timings
- Can you think of ways to improve our implementation of the merge sort idea?

39.11 Final Comparison Across All Sorts

- Selection sort and insertion sort are dramatically slower than merge sort, which in turn is dramatically slower than Python's built-in sort, a highly optimized, C language implementation of merge sort.
- Shows
  - the difference between $O(N^2)$ sorting and $O(NlogN)$ sorting, and
  - the difference between a straight-forward Python implementation and a careful, optimized implementation of the same algorithm.

  Both of these are important!
- Final question: what happens when values are "almost" sorted?
  - Experimentally, we can explore this using the `generate_local_perm` function in `test_sort.py`.
  - Insertion sort becomes much faster, far outstripping selection sort. Why?

39.12 Practice Questions

1. For our insertion sort code, show the contents of the following list after each iteration of the outer `for` loop

   ```
   v = [ 12, 4, 11, 2, 6, 18, 9 ]
   ```

   While you can and should use the implementation to test your answers, you should start by manually generating the answers on your own.

2. Show the contents of the `lists` list at the end of the `merge_sort` implementation developed in class when it is called with

   ```
   v = [ 17, 15, 29, 66, 31, 19, 9, 33 ].
   ```

3. Consider the following function

   ```
   def extract( comp, v ):
       x = v[0]
       for i in range(1,len(v)):
           if comp(v[i],x):
               x = v[i]
       return x
   ```

   Note that `comp` is a function that has been passed to `extract`.

   (a) Write a function called `compare_lower(a,b)` such that if `L` is a list then the call

   ```
   extract(compare_lower,L)
   ```

   returns the smallest value in `L`.

   (b) Write a function called `compare_upper(a,b)` such that if `L` is a list then the call

   ```
   extract(compare_upper,L)
   ```

   returns the largest value in `L`.

4. Write a version of `merge` that does all of the work inside the `while` loop and does not use the `extend`. This is a good test of your logic skills.

5. Based on your previous solution write a function to merge three sorted lists. This is an even greater challenge to your logic skills.

Note that when it comes to the Final, you will not be required to have memorized the code of the sorting functions, but you should know the algorithms!

Sort Testing Code

```
'''
Testing code for Computer Science 1, Lecture 21 on sorting. This
assumes that the sort functions are all in file lec21_sorts.py, each taking
one list as its only argument, and that their names are sel_sort
ins_sort merge_sort

All tests are based on random permutations of integers.

. In most of our tests, these permutations are completely random,
  meaning that a value is equally likely to end up anywhere in the
  list.

. In the final test we will explore the implications of working
  with lists that are "almost sorted" by only moving values a small
  distance from the correct location.  You can see that insertion sort
  is very fast in this case by removing the # char in front of
  generate_local_perm
'''

import lec21_sorts as sorts
import time
import random


def run_and_time(name, sort_fcn, v, known_v):
    '''
    Run the function passed as sort_fcn, timing its performance and
    double-checking if it correct.  The correctness check is probably
    not necessary.
    '''
    print("Testing " + name)
    t0 = time.time()
    sort_fcn(v)
    t1 = time.time()
    print("Time: {:.4f} seconds".format(t1-t0))
    # print("Is correct?", v==known_v
    print()


def generate_local_perm(v,max_shift):
    '''
    This function modifies a list so values are only a small amount
    out of order.  Each one  Generate a local permutation by randomly moving each
    value up to max_shift locations in the list.
    '''
    for i in range(len(v)):
        min_i = max(0,i-max_shift)
        max_i = min(len(v)-1, i+max_shift)
        new_i = random.randint( min_i, max_i )
        v[i], v[new_i] = v[new_i], v[i]
```

```
##################################################

if __name__ == '__main__':
    n = int(raw_input("Enter the number of values ==> "))
    print("----------")
    print("Running on {:d} values".format(n))
    print("----------")

    v = range(n)
    v1 = v[:]
    random.shuffle(v1)
    # generate_local_perm(v1, 10)
    v2 = v1[:]
    v3 = v1[:]
    v4 = v1[:]

    run_and_time("merge sort", sorts.merge_sort, v3, v )   # passing functions as an arg to a fcn
    run_and_time("python sort", list.sort, v4, v )
    run_and_time("selection sort", sorts.sel_sort, v1, v )
    run_and_time("insertion sort", sorts.ins_sort, v2, v )
```