CmpSci 187: Programming with Data Structures Spring 2015

Lecture #22, More Graph Searches, Some Sorting, and Efficient Sorting Algorithms

John Ridgway

April 21, 2015

1 Review of Uniform-cost Search

Uniform-Cost Search

- This method from DJW makes a priority queue of `Flight` objects, which each contain a start vertex, an end vertex, and a total distance.
- The `compareTo` method of the `Flight` class treats objects with smaller distances as "larger", so that they will be dequeued from the priority queue first.
- It takes a bit of reasoning to prove that this is correct, as you'll see in CMPSCI 250.

Code for `UCSEntry`

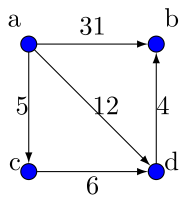
```
import java.lang.Comparable;

public class UCSEntry<V>
    implements Comparable<UCSEntry<V>> {
  private V vertex;
  private Double cost;
  public UCSEntry(V vertex, Double cost) {
    this.vertex = vertex;
    this.cost = cost;
  }
  public V getVertex() { return vertex; }
  public Double getCost() { return cost; }
  public int compareTo(UCSEntry<V> other) {
    return Double.compare(other.cost, this.cost);
  }
}
```

Code for Uniform-Cost Search

```
public double ucs(
   WeightedGraphInterface<V> g, V from, V to) {
  PriorityQueue<UCSEntry<V>> queue =
             new PriorityQueue<UCSEntry<V>>();
  GraphMarker<V> marker = g.getMarker();
  queue.add(new UCSEntry<V>(from, 0.0));
  do { UCSEntry<V> e = queue.remove();
    V v = e.getVertex();
    Double c = e.getCost();
    if (v == to) { return c; }
    if (! marker.isMarked(v)) {
      marker.mark(v);
      for (V neighbor : g.getNeighbors(v)) {
        queue.add(new UCSEntry<V>(neighbor,
          c + g.getEdgeWeight(v, neighbor))); } }
  } while (! queue.isEmpty());
  return Double.POSITIVE_INFINITY; }
```

Clicker Question #1

Suppose we perform a UCS on this directed graph starting at vertex a, with b as the goal. When we first put b on the priority queue, what is its distance?



A. 4

B. 15

C. 16

D. **31**

2 Sorting

The Sorting Problem

- We've seen that searching, for example, can be performed much more quickly on a sorted list than on an unsorted one.
- When we have data in a spreadsheet, sorted using one definition of "smaller", we often want to see what it looks like sorted by some other definition.
- It's thus interesting to look at algorithms for sorting a list of objects.
- We'll present a number of sorting algorithms in this lecture and the next.
- All will be **comparison-based**, meaning that the only operation they perform on the items is the `compareTo` method that all `Comparable` objects must have.

- There are also non-comparison-based sorting algorithms, that use particular properties of the objects they sort.
- We'll look at the asymptotic running time of each sort, and also at other considerations that might lead us to prefer one algorithm over another.
- These include memory usage, requirements on access to the data, and behavior in the average case, best case, or typical case.
- You'll see more about these algorithms, with more mathematical proofs, in CMPSCI 311.

A Sorting Test Harness

- DJW present a driver program called a **test harness** that can be used to demonstrate each sorting method and evaluate the number of comparisons and swaps that it uses on a randomly chosen input.
- Swaps are our basic means of moving items around within the space they originally occupy. An **in-place sort** has the property that the objects never move outside of this space.
- The harness will sort sequences of 50 integers, each randomly chosen to be any number in the range from 0 to 99 with equal probability.
- We will thus usually have sets of **equal elements** in the sequence.
- Since the sorting algorithms are comparison-based, they would take the same number of comparisons and swaps on any set of data of the same **order type**.

What's an Order Type?

- Consider the two input lists $\{3, 7, 1, 4, 6, 4\}$ and $\{12, 15, 2, 13, 14, 13\}$. If we take any comparison in the first list, such as item 2 versus item 5, the corresponding comparison in the second list has the same result.
- This correspondence will stay true as the same comparison-based sorting algorithm works on both lists.
- The two lists have **the same order type** in this case.

A Sorting Test Harness

- The harness has a method `initValues` to populate the array, a method `isSorted` to test whether the array is already sorted, a method `swap` that switches the values at two given indices, and a method `printValues` that gives the 50 values in a table of 5 rows with 10 values each.
- In the book, DJW show sample outputs from the harness for each of their sorting methods.

Selection Sort

- A natural way to sort is to find the element that should come first, put it in the first position, and continue by finding the element that should come next.
- We find the next element by simple linear search, then swap it into its correct position. This gives us an in-place sort, that avoids using memory for both the old and new list.
- This would also be easy to code recursively.

Code for Selection Sort

```
static int minIndex(int startIndex,
                    int endIndex) {
  int indexOfMin = startIndex;
  for (int index = startIndex + 1;
       index <= endIndex; index += 1) {
    if (values[index] < values[indexOfMin]) {
      indexOfMin = index; } }
  return indexOfMin; }

static void selectionSort() {
  int endIndex = SIZE - 1;
  for (int current = 0;
       current <= endIndex; current += 1)
    swap(current, minIndex(current, endIndex));
```

Analyzing Selection Sort

- The `minIndex` method with parameters x and y always takes $O(y-x)$ time, since the comparision is done with each item to find the one that belongs at the front. In this case it compares item x against $y-x-1$ other elements.
- The total time can't be more than $O(n^2)$, since we have a main loop that we go through $n-1$ times, and $O(y-x)$ is at worst $O(n)$. But we can figure out the exact number of comparisons.

Clicker Question #2

What is the worst case for selection sort? That is, what input order on n inputs causes selection sort to take the greatest number of comparisons?

A. the input is already sorted

B. the input is exactly backward

C. the order $\{n, 1, 2, 3, ..., n-1\}$

D. **selection sort always takes the same number of comparisons on n inputs**

Analyzing Selection Sort

- The total number of comparisons is actually $S(n) = (n-1) + (n-2) + ... + 2 + 1$ which evaluates to $n(n-1)/2$, the n'th "triangle number". (Perhaps the easiest way to see this is to add $S(n)$ to $1 + 2 + ... + (n-1)$ term by term, to get the fact that $2S(n)$ is $n-1$ terms, each of which is n.)
- If swaps are much more expensive than comparisons, selection sort is actually rather good, because it uses only $n-1$ swaps in the worst case.

Bubble Sort

- Bubble sort is a variant of selection sort where we find the next element in sorted order by first comparing the last two elements, then comparing the lesser of those two with the next to next to last, then comparing the smallest so far with the one before that, and so on.
- The minimum element in the range that we check "bubbles" to the top, and some other elements also move.

Code for Bubble Sort

This time we use $n(n-1)/2$ swaps in the worst case. But they are adjacent swaps; we could implement bubble sort on a linked list as well as on an array.

```
static void bubbleUp(int startIndex,
                     int endIndex) {
  for (int index = endIndex;
       index > startIndex; index -= 1) {
    if (values[index] < values[index - 1]) {
      swap(index, index - 1); } } }
static void bubbleSort() {
  for (int current = 0;
       current < (SIZE - 1); current += 1) {
    bubbleUp(current, SIZE - 1); } }
```

Insertion Sort

- Another natural way to sort is to gradually build up a sorted list by successively inserting new elements.
- After k rounds of this process, the first $k+1$ elements of the list are sorted with respect to one another, though not with the others.
- This is a bit like our reheaping up – we have a hole at the end of the list, and we move it up by swapping until the new element can go into it.

Code for Insertion Sort

```
static void insertElement(int startIndex,
                          int endIndex) {
  for (int current = endIndex;
       current > startIndex; current -= 1) {
    if (values[current] < values[current - 1]) {
      swap(current, current - 1); }
    else { return; } } }

static void insertionSort( ) {
  for (int count = 1; count < SIZE;
       count += 1) {
    insertElement(0, count); } }
```

Analyzing Insertion Sort

- The insertion step with parameters x and y also takes $y-x-1$ comparisons in the worst case, and it takes that many swaps as well. Again the worst case is a backwards list. The total number of comparisons is again $n(n-1)/2$, and the total running time is $O(n^2)$.
- Insertion sort has a good **best-case** behavior; if the list is already sorted it still does n insertion steps, but each one then takes no swaps and only one comparison, so the total time is $O(n)$.
- One measure of how far from being sorted a list might be is the number of **reversals**, the number of pairs of elements that are in the wrong relative order.
- Both bubble sort and insertion sort perform a number of swaps that is equal to the number of reversals, since each adjacent swap fixes exactly one reversal.
- But if the number of reversals is r, insertion sort takes only $O(n+r)$ total time, which is very good if the list is relatively sorted.

Clicker Question #3

How many reversals are there in the list (A, M, H, E, R, S, T)? (We use ordinary alphabetical order on the letters.)

A. 2

B. **3**

C. 4

D. more than 4

3 Efficient Sorting Algorithms

Links: Sorting Videos

- One of the first and most famous animated videos in computer science education is Sorting Out Sorting from the University of Toronto, at `http://www.youtube.com/watch?v=SJwEwA5gOkM`
- Another comprehensive set of sorting animations is at `http://www.sorting-algorithms.com`
- Several sorting algorithms can be seen at `http://youtube.com/user/AlgoRythmics`, performed by folk dancers.

$O(n \log n)$-Comparison Sorts

- In tomorrow's discussion, we will present an argument that any comparison-based sorting algorithm must use at least $\log(n!)$ comparisons.
- This is because any such algorithm can be represented as a decision tree, and such a tree must have at least $n!$ leaves in order to be able to give any of the $n!$ possible correct answers. If it cannot give one of the answers, it will be wrong if that answer is correct.
- How do we compute $\log(n!)$, in big-O terms?
- The log of $n!$, as a real number, is equal to $\log(n) + \log(n-1) + \log(n-2) + ... + \log(2) + \log(1)$.
- These are n terms, each at most $\log n$, so the sum is at most $n \log n$. But the first half of those terms are also at least $\log(n/2) = \log(n) - 1$, so the sum is at least $(n/2)(\log(n) - 1)$ which is about $(1/2)(n \log n)$. Thus $\log(n!) = O(n \log n)$.
- Can we achieve a sort with only $O(n \log n)$ comparisons?
- We've actually seen one way to do this – enqueue each of the n elements into a heap-based priority queue, then dequeue them one by one. Each enqueue and dequeue operation takes only $O(\log n)$ comparisons.
- In this lecture we'll see three other ways to sort in $O(n \log n)$ comparisons, which are a bit better than this.

Merge Sort

- The first two of our three $O(n \log n)$ sorts use a divide and conquer strategy. We divide our list into two pieces, recursively sort each piece, then combine the two pieces.

- In Merge Sort, we divide the list arbitrarily and work to combine the two sorted lists.
- In Quick Sort, we work to divide the list into one sublist of larger elements and one of smaller elements, a large and a small sublist, which we can combine by just concatenating.
- DJW's method takes a range of the array, which may be entirely unsorted, and leaves it sorted.
- It does nothing if the range has size 1, of course.
- The method it calls to do all the work takes two ranges, each assumed to be sorted, and merges them into a single sorted range.

Code for Merge Sort

Note that we have a base case of a one-element list with `first == last`. Our method is correct as long as the `merge` method is.

```
static void mergeSort(int first, int last) {
  if (first < last) {
    int middle = (first + last)/2;
    mergeSort(first, middle);
    mergeSort(middle + 1, last);
    merge(first, middle,
          middle + 1, last); } }
```

Analyzing Merge Sort

- Suppose for convenience that our array size n is a power of 2, so that whenever a range is split, it is split exactly in half. How many comparisons does Merge Sort use to sort this array?
- To merge an array of size x with an array of size y, it should be clear that the merge method uses $O(x + y)$ comparisons. The exact number is very close to $x+y$, as for the most part each comparison leads to an item being copied into the temporary array.
- To count the total comparisons, we can divide the entire set of merge operations into phases, where the operations in each phase take a total of $O(n)$ time. The first phase is the merging of each single element into a pair. The second is the merging of pairs into sets of size 4, the third merges those into size 8, and so on until phase number $\log n$ creates the final sorted list.
- There are thus $\log n$ phases of $O(n)$ comparisons each for $O(n \log n)$ in all.

Space Usage of Merge Sort

- If we use Merge Sort on an array, we cannot sort in place as we did with the three $O(n^2)$ sorts.
- The last merge operation, for example, takes two array segments of length $n/2$ each and merges them into an array of length n. Since all three arrays exist at the same time, we use $2n$ locations.
- The up-side is that Merge Sort does not require random access to the data, and so can be used with linked lists or files.
- Actually, when using linked lists, we only need $O(\log n)$ extra space rather than $O(n)$.

Clicker Question #4

Suppose we alter Merge Sort so that if the list size is 20 or less, we use Insertion Sort rather than recursing. How will this affect the big-O running time of the Merge Sort?

A. It will improve to $O(n)$

B. It will take exactly as many steps as before.

C. **It will remain $O(n \log n)$ though the constant factor may change.**

D. It will become $O(n^2)$ because Insertion Sort is $O(n^2)$.

Quick Sort

- **Quick Sort** also divides the list in two, sorts each piece recursively, and combines the pieces; but here we make sure that every item in the first piece is less than or equal to every item in the second piece.
- We do this by taking a pivot element and comparing each other element to the pivot. Items smaller than the pivot go in the first piece, and items larger than it go in the second. We could put elements equal to the pivot in either; we'll put them in the first.
- We could use a temp array as in Merge Sort, but Quick Sort is able to sort pretty much in place.
- We'll create a method `split` that will choose a pivot, do the comparisons, and leave the pivot in its new correct place.
- Every element before the pivot will be less than or equal to it, and everything after the pivot will be greater than it.

Code for Quick Sort

Again our base case will be a list of size 1, which needs no sorting. (We will also call `quickSort (first, last)` in cases where `last < first`).

```
static void quickSort(int first, int last) {
  if (first < last) {
    int splitPoint = split(first, last);
    quickSort(first, splitPoint - 1);
    quickSort(splitPoint + 1, last); } }
```

The Splitting Method

```
static int split(int first, int last) {
  int splitV = values[first];
  int saveF = first;
  first += 1;
  do {
    while (first<=last && values[first]<=splitV){
      first += 1; }
    while (first<=last && values[last]>=splitV) {
      last -= 1; }
    if (first < last) {
      swap(first, last);
      first += 1;
      last -= 1; }
  } while (first <= last);
  swap(saveF, last);
  return last; }
```