Lecture 8

Selection Sort Review

Review recursive algorithm for selection sort?

- if the list is bigger than size 1:
  - find the largest item in the list
  - swap it with the item at the end of the list
  - recursively sort the list from [0 , lastIndex - 1]

We used induction on the size of the list to prove that this approach works. The key idea was this:

- put largest item at index lastIndex
- assume recursive call on list [0 , lastIndex - 1] works
- combine recursive sort of smaller list with information about largest item to conclude overall algorithm works

This is very typical of recursive algorithms. We typically solve some smaller version of the problem, then combine this solution with some other work to solve the larger problem.

It is no coincidence that this is what our inductive proofs often look like!

Merge Sort

If we have two sorted lists, can we easily merge them together into one sorted list? How?

Let's assume that we have a separate array where we are creating the merged list.

- repeat until one list is empty:
  - Examine first item on each sorted list, and choose the smallest
  - Move that item to our large list
- copy remainder of other list to new list

We can use this idea of merging to create a recursive algorithm to sort a list of items.

- recursively sort the left half of the list
- recursively sort the right half of the list
- merge the two halves together into a sorted list

Can we prove that this method is correct?

Proof by induction on size of list.

Base case: list of size 1, code returns original list. Original list of size 1 is sorted by definition. Holds!

Inductive case: Assume that mergeSort works on lists smaller than size n (That is, assume that recursive call on left side of list works. Assume recursive call on right side of list works.)

Now, need to combine these results. Just need to argue that merge works. This is not inductive proof, just direct proof.

- at each step, the first item on list a is smallest item in list a
- at each step, the first item on list b is smallest item in list b
- at each step, we choose smaller of first item on list a or first item on list b to move to results.
- since we are choosing the smallest item of all possible remaining items at each step to add to *results*, *results* is sorted.

Since left side is sorted, and right side is sorted, then merge returns a sorted list of all elements.

End of proof.

Now we want to look at the complexity of Merge Sort. Let's consider how the code works.

In merge, we have two lists. For simplicity, we'll just consider number of comparisons between elements.

For each comparison, we move one item to the results list. Sometimes, we don't need a comparison to move an item. If one list is empty, we'll just move the rest of the other list.

So, in terms of comparisons, at each level, we are doing $\leq n$, or $O(n)$ work.

How many levels do we have. We split the list in two each time, until we get n lists of size 1. We can look at how many lists we have.

- step 0: we have 1 list
- step 1: we have 2 lists
- step 2: we have 4 lists
- step 3: we have 8 lists
- ...
- step i: we have 2^i list

We want to know when $2^i = n$. Solving for i, we get $i = \log_2(n)$.

So, n work per level, times $\log_2(n)$ levels, gives $n \log_2(n)$ total work.

Again, this is a good guess, based on logical reasoning. But we really want to prove that this is true.

Prove by induction on size of list n that comparisons done by MergeSort is $\leq n * \log_2(n)$.

To simplify matters, let's assume that the size of the list is always a power of two. That is, $n = 2^m$, where m is some integer $m = \{0, 1, \ldots\}$.

Basis step: list of size 1 does 0 comparisons according to code.

$1 * log_2(1) = 1 * 0 = 0$ $0 \leq 0$, so basis step holds.

Inductive step. Assume that sorting left half does $\leq \frac{n}{2} \log_2(\frac{n}{2}) = 2^{m-1} * (m-1)$ comparisons, and the sorting the right half does $\leq \frac{n}{2} \log_2(\frac{n}{2}) = 2^{m-1} * (m-1)$ comparisons.

Then the complexity of sorting the whole list is

$$
\begin{aligned}
&\leq 2^{m-1} * (m-1) + 2^(m-1) * (m-1) + 2^m \\
&= 2 * 2^{m-1} * (m-1) + 2^m \\
&= 2^m * (m-1) + 2^m \\
&= 2^m * ((m-1) + 1) \\
&= 2^m * m \\
&= n \log_2(n)
\end{aligned}
$$

Quick Sort

Basic approach:

- choose a pivot value. In simplest case, we can choose first item on list to be the pivot.
- move all values less than pivot to the left of the pivot in list.
- move all values greater than pivot to the right of the pivot in list.
- recursively Quick Sort on the left half and the right half.

Similarities:

- Another Divide and Conquer algorithm. Cuts list in half, recursively calls itself on each half.
- In average case, same complexity as Merge Sort. Same number of levels, same work per level.

Differences:

- Merge Sort did the recursive call on each half first, then did $O(n)$ work to merge the lists. Quick Sort does $O(n)$ work to move the values to correct side, then recursively splits.
- Merge Sort complexity is always $O(n \log_2(n))$, even in worst case. Quick Sort complexity can reach $O(n^2)$ in worst case.

Quick Sort reaches $O(n^2)$ when it tries to sort a list that is already sorted. This happens because the number of levels becomes $O(n)$, since we need to recursively sort a list of size $n-1$ and a list of size 0 in the two recursive calls. Thus, we don't reach base case until $n-1$ calculation reaches size 1.

NOTE: Java uses both MergeSort (Arrays.sort(Object[] a)) and QuickSort(Collections.sort(List list)).