CmpSci 187: Programming with Data Structures Spring 2015

Lecture #23, More Sorting Algorithms

John Ridgway

April 23, 2015

1 Efficient Sorting Algorithms

Merge Sort Review

Code for `mergeSort`

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

Code for `merge`

```
void merge(int first, int middle, int last) {
  int[] tempArray = new int[SIZE];
  for (int i = first; i <= last; i += 1) {
    tempArray[i] = values[i]; }
  int left = first;
  int right = middle + 1;
  int resultIndex = first;
  while ((left <= middle) && (right <= last)) {
    if (tempArray[left] < tempArray[right]) {
      values[resultIndex] = tempArray[left];
      left += 1;
    } else {
      values[resultIndex] = tempArray[right];
      right += 1; }
    resultIndex += 1;
  }

  while (left <= middle) {
    values[resultIndex] = tempArray[left];
    left += 1;
    resultIndex += 1;
  }
```

```
    while (right <= last) {
      values[resultIndex] = tempArray[right];
      right += 1;
      resultIndex += 1;
    }
  }
```

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`).

```
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

```
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; }
```

Analyzing Quick Sort

- The number of comparisons used by Quick Sort is more complicated to analyze, and you won't see the real answer until CMPSCI 311.
- First note that the worst-case behavior is terrible; $O(n^2)$ comparisons. This is because if the pivot is the first or last element, the split step does nothing.
- If the pivot were always the median element, on the other hand, the divide-and-conquer analysis would be the same as for Merge Sort, $O(n \log n)$.
- The actual behavior is somewhere in between, but in CMPSCI 311 you'll show that the average-case number is $O(n \log n)$, and that the constant in the big-O running time is smaller than that for Merge Sort or Heap Sort.
- Thus the sort methods in commercial code (such as Java's Arrays.sort) usually use Quick Sort. The algorithm rewards effort on optimizing tricks — see "Engineering a Sort Function" by Bentley and McIlroy.

Clicker Question #1

If the input list is random, the first element is as good a pivot as any other. For which of these input lists would this choice of pivot not lead to $O(n^2)$ running time for Quick Sort?

A. $\{1, 2, 3, ..., n\}$

B. $\{n, n-1, n-2, ..., 1\}$

C. $\{n/2, n/2-1, ..., 1, n, n-1, ..., n/2+1\}$

D. **Trick question – all three are $O(n^2)$.**

Heap Sort

- Our last $O(n \log n)$ sort starts with the idea of our first one; use a heap to get $O(\log n)$ enqueueing and dequeuing for a priority queue, then just put all the elements in and take them out. **Heap Sort** improves this with a few tricks.
- Remember that our Heap method `reheapDown` took a value as parameter and placed that value in the heap in place of whatever was at the root, preserving the heap property. Here we use a variant that places the value in the first suitable place within the range given by two parameters.

This algorithm works in two phases. We first form the list into a max-heap. Then we take the maximum element from the root and put it where it belongs, at the end of the array. Then we take the maximum of the remaining heap and put it in the next-to-last position of the array, and so on.

```
void heapSort() {
  int index;
  for (index=SIZE/2-1; index>=0; index-=1) {
    reheapDown(index, SIZE); }
  for (index=SIZE-1; index >= 1; index-=1) {
    swap(0, index);
    reheapDown(0, index); } }
```

- In this way we build up a sorted list at the end of the array, and the elements up to the sorted list remain a (decreasing) heap.
- Of course we could form the list into a heap by successively inserting all n elements, but the way used here turns out to be faster.

```
void heapSort() {
  int index;
  for (index=SIZE/2-1; index>=0; index-=1) {
    reheapDown(index, SIZE); }
  for (index=SIZE-1; index >= 1; index-=1) {
    swap(0, index);
    reheapDown(0, index); } }
```

Heapifying an Unsorted List

- The first loop of the Heap Sort code forms the original list into a heap in a **bottom-up** fashion.
- We consider each node x of the heap that is not a leaf. Assuming that the subtrees under each of x's children are heaps, but that the value at x itself may violate the heap property, we want to make the subtree under x into a heap.
- Once we have done this for the root, we have a heap.

Clicker Question #2

Why does the first loop start at $SIZE/2 - 1$ rather than at $SIZE - 1$ like the second loop?

A. we only form half the elements into a heap

B. heapifying the left subtree also heapifies the right

C. **the larger values are leaves of the heap**

D. we need $SIZE/2$ - 1 to make the time $O(n)$

```
void heapSort() {
  int index;
  for (index=SIZE/2-1; index>=0; index-=1) {
    reheapDown(index, SIZE); }
  for (index=SIZE-1; index >= 1; index-=1) {
    swap(0, index);
    reheapDown(0, index); } }
```

Code for `reheapDown`

```
private void reheapDown(int i, int size) {
  int l = 2 * i + 1;
  int r = 2 * i + 2;
  if (l >= size) { return; }
  int big = l;
  if (r < size && values[r] > values[l])
    {
      big = r;
    }
  if (values[i] < values[big]) {
    swap(big, i);
    reheapDown(big, size);
  }
}
```

Heapifying an Unsorted List

- We take at most $O(\log n)$ time to reheap at each of $n/2$ nodes, which would be $O(n \log n)$. But note that for most nodes we only take a few steps, the subheap has only one level for $n/4$ nodes, only two for $n/8$, only three for $n/16$, and so on. A careful analysis shows that we need only $O(n)$ to heapify the list.

How Long to Heapify a List?

- The total time to heapify a list with this method is thus a sum. We take the times to create heaps at each node, and add them together.
- Rather than look at the exact, more complicated sum that comes up here, let's look at a simpler, similar case.
- We want to add up n numbers, where n is a power of two, and the numbers have a similar distribution to the numbers in heapifying.

A Math Trick

- If half the nodes of the heap take one step to reheap, and a quarter of them take two, an eighth of them take three, and so on, we get a sum like $1 + 2$, $1 + 1 + 2 + 3$, $1 + 1 + 1 + 1 + 2 + 2 + 3 + 4$, or $1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 2 + 2 + 2 + 2 + 3 + 3 + 4 + 5$.

- These sums happen to add to 3, 7, 15, and 31. As computer scientists, we should be trained to recognize sequences with powers of two, so we should see that the sum of $n = 2^k$ terms in each case adds up to $2^{k+1} - 1$ or $2n - 1$. So the sum is $O(n)$.

Clicker Question #3

Suppose $n = 2^k$ is a power of two and we want to add up n numbers as on the last slide: half the numbers are 1, a quarter of them 2, an eighth 3, and so forth. What is the largest number in the set?

A. n

B. $n - 1$

C. $k + 1$

D. k

2 The Selection Problem

- The selection problem is to take a collection of elements from some ordered type, and an index k, and to return the kth smallest element in the collection.

- Sorting the collection is equivalent to solving the selection problem for all possible indices, but we will consider the difficulty of individual selection problems for unsorted collections. If $k = 1$ we are finding the minimum, if $k = n$ the maximum, and if $k = n/2$ the median element.

- As with comparison-based sorting, we will measure the difficulty of selection by the number of comparisons needed to determine the kth smallest object. Clearly we don't need more than $O(n \log n)$ for any k, since that is enough to sort the collection completely.

- The median element would be the best possible pivot for Quick Sort, but even with the best selection algorithms the extra effort to find the median outweighs the advantage from using it.

Finding Maxima or Minima

- Finding the maximum or minimum can be likened to a sports tournament, where we have n teams and want to find an undisputed champion.

- We could hold a binary tournament (as in college basketball) where we group the items into pairs, compare each pair, group the winners from each pair into new pairs, compare them, and so on, halving the number of remaining competitors each time until there is only one.

- A ladder tournament compares two items, compares the winner against a third, the winner of that against a fourth, and so on until all items have been compared. Both the binary and ladder are single elimination tournaments.
- It's easy to see that either of these methods uses exactly $n-1$ comparisons, because every item except for the eventual winner has been the loser in exactly one comparison.

Lower Bound for Max/Minima

- The existence of these tournaments shows that $n-1$ comparisons are sufficient to find the maximum or minimum. This is an upper bound on the worst-case number needed.
- It is not hard to prove that $n-1$ comparisons are also necessary to find the maximum or minimum, giving a lower bound on the number needed.
- Suppose we are finding the maximum and have done only $n-2$ comparisons.

Clicker Question #4

Suppose I have done $n-2$ comparisons among my n elements, and I claim that element x is the maximum. Why must I be wrong in the worst case?

A. **another element y must also be undefeated**

B. x must have lost at least one comparison

C. there must be an element that was never compared to any other element

D. only a binary tournament can find the max

Adversary Arguments

- This is our third example of an adversary argument in this course.
- We showed that no comparison-based search of a sorted list with 2^k elements could succeed with k or fewer comparisons in the worst case.
- We showed that any sorting algorithm with fewer than $\log(n!) = O(n \log n)$ comparisons must fail in the worst case.
- Here, if there are two undefeated elements, in the worst case you pick the wrong one and are wrong.

Finding Maxima and Minima

- Suppose we want to conduct comparisons to determine both the maximum and the minimum of the same collection.
- Clearly we could use $n-1$ comparisons to find the maximum and another $n-1$ to find the minimum, for $2n-2$ total.
- But there is a better way, that uses some of the same comparisons to serve both goals.
- Group the n elements into pairs (assume that n is even) and compare each pair.
- The maximum must be among the $n/2$ winners and the minimum among the $n/2$ losers.
- We can use one single elimination tournament to find the maximum with $n/2-1$ comparisons, and another tournament to find the minimum with another $n/2-1$, making $n/2+(n/2-1)+(n/2-1)=3n/2-2$ total comparisons.
- It turns out that $3n/2-2$ comparisons are needed.
- In fact this number of $3n/2-2$ cannot be improved using only comparisons. Proving this takes a more sophisticated adversary argument.
- We can design a system to provide answers to any algorithm's comparisons, so that until $3n/2-2$ have been used, there are either two undefeated items or two winless items, and the algorithm cannot give an answer correct in the worst case.

The Quick Select Algorithm

- What if we want the kth smallest item for arbitrary k? The basic idea of Quick Sort gives us an algorithm called Quick Select, which is the best known in the average case. (Like Quick Sort, in the worst case it needs $O(n^2)$ time.)
- Suppose we pick a pivot and compare it against all the other items, determining that it is the pth smallest element. This gives us some information about the kth smallest.
- Our pivot has proved to be pth smallest.
- If $k<p$, I now need to select the kth smallest item from the elements smaller than the pivot. If $k=p$, of course, I am done. If $k>p$, I need to select the $(k-p-1)$st element from those larger.

- So we make a recursive call to Quick Select, on a new list that is smaller by at least one element.
- If the pivot were always in the middle, we would spend at most $(n-1)+(n/2-1)+(n/4-1)+...+1$ comparisons, which sums to about $2n$. In CMPSCI 311, you may show that the average-case time is $O(n)$. But note that if the pivot is always the maximum or minimum of the remaining elements, you use $O(n^2)$.
- Also in CMPSCI 311, you may see a more sophisticated algorithm that always selects the kth smallest element with $O(n)$ comparisons, for any k.

Clicker Question #5

The worst case number of comparisons to find the median with Quick Select is $n(n-1)/2$. What is the best case number of comparisons to find the median? (Assume that n is odd.)

A. $O(1)$

B. $n-1$

C. n

D. $3n/2-2$