CSCI S-Q Lecture 7

Searching and Sorting

• Comparison-based searching
• Comparison-based sorting
  • Terms and definitions
  • Why are there so many sorting algorithms?
  • Simple sorting algorithms and their complexity bounds
  • Divide-and-conquer sorting algorithms
  • General complexity bounds of comparison-based sorting algorithms
Comparison-based Searching:

Rules of the Game

We will begin by considering a very constrained (but very useful and practical) type of search: searching for a key in an array of elements that has been sorted by that key.

We assume that:

- Accessing any element in the array can be accomplished in time that is \( O(1) \).
- The type of the key has well-defined less than, equal, and greater than comparison operations, and these operations are transitive.

What do we hope to accomplish?

The \texttt{FIND} operation on the array ADT is \( O(N) \) for ordinary arrays. However, if the array is sorted (by some well-defined and transitive relationship between all the keys in the array), the \texttt{FIND} can be \( O(\log N) \) (as we will see very shortly).

From this, and our previous experiences, we know that some operations can be done much more quickly on sorted data.
Binary Search

Binary search is a classic divide-and-conquer strategy.

• To search for a key K in a sorted array A of length N
  1. If N == 0, the search failed: return nil.
  2. If K == A[N/2], we have found what we are looking for. Return N/2.
  • If K > A[N/2], we need to search the "upper" half of the array. Call binary search recursively on the array starting at &A[N/2 + 1] and of length N - (N/2 + 1).
    If this recursive search is successful, we need to add N/2 + 1 to the return value from the recursive call.
  • If K < A[N/2], we need to search the "lower" half of the array. Call binary search recursively, again on the array starting at A but with length (N - N/2) - 1.

• Analysis

At each instance of the recursion, we divide the search space approximately in half. The search will take at most $\log_2(N) + 1$ steps to either find the key or to fail, so this search algorithm is $O(\log N)$. 
Comparison-based Sorting

We will treat \texttt{SORT} as an ADT operation on the array ADT, where the array elements are themselves an ADT that supports $<$, $=$, and $>$ operations (where $<$, $=$, and $>$ are defined in the usual way, and are transitive).

\texttt{SORT}(A, N) - sort the first N elements of array A in-place. (we are not creating a new array-- A itself is being modified)

- \textit{Why just the first N, and not the whole thing?}
  Because many times we’ll find it useful to sort only part of an array.

The first N elements of A are sorted if and only if for all i from 0 to N-2, \texttt{A}[i] <= \texttt{A}[i+1].

In most cases, we will assume that the array ADT is implemented by an ordinary C array, and so we can access any element of the array in O(1) time. (It is interesting to explore the impact it has on our analysis if this is not the case, however!)

We will measure complexity in terms of:

- \textbf{comparisons} between elements of the array
- \textbf{swaps} or \textbf{copy} operations that move elements of the array.
- The total amount of \textbf{memory} used.
Why so many sorting algorithms?

- Different algorithms are appropriate for different types of keys and storage of data:
  - trading computation for comparisons when comparisons are expensive
  - trading computation for swaps or data movement
  - difficulty in randomly accessing data
- Different algorithms are appropriate when the data has particular properties:
  - the data is “almost” sorted already
  - the data is randomly permuted
  - the keys have some useful properties (i.e. integers between 0 and 20)
- Some have theoretic interest.

Inversions

An *inversion* is a pair of elements that is "out of order": \(A[x] > A[y]\), but \(x < y\).

Theorem (from Weiss, 7.1):

The average number of inversions in a randomly permuted array of \(N\) unique values is \(N(N-1)/4\).

So, the average number of inversions in a randomly permuted array of \(N\) unique values is \(O(N^2)\).
Sorting Algorithms

The Brute-Force Algorithm

Try each permutation of the array until you find one that has the sorted property.

Unfortunately, an array of N distinct values has N! distinct permutations, so this algorithm is O(N!). If N = 70, N! > 10^{100}, and if N = 100, N! > 10^{160}.

This is a good example of where the Brute-Force approach is not good enough for even small cases.

The Ideal Algorithm

Examine each element in the array in turn, performing some small, O(1) computation for each. As a result of that O(1) computation, we discover exactly where the element should appear in the sorted array, and place it there.

Surprising as it may seem, for many useful types of data an ideal algorithm does exist. (We will encounter some of them later.)
Sorting Algorithms - Bubblesort

Principle: find neighboring elements that are out of order, and swap them:

```c
if (A[j] > A[j+1]) {
    SWAP(A, j, j+1);
}
```

Easily coded in a loop that repeatedly scans the array:

```c
for (i = 0; i < N - 1; i++) {
    for (j = 0; j < (N - 1) - i; j++) {
        if (A[j] > A[j+1]) {
            SWAP(A, j, j+1);
        }
    }
}
```

Notes:

- We only need to do N-1 passes, because if N-1 elements are in position, the N\textsuperscript{th} one must be as well.

- We don’t need to scan all the way down the array each time: after the i\textsuperscript{th} pass, the last i elements will be in the right places-- the largest remaining element "bubbles up" to the top during each scan.

- We can stop as soon as a scan doesn’t cause any swaps. (not implemented in this code)
Bubblesort Analysis

Worst case:

The smallest item begins in position N. It moves 1 position to the left with each scan, so N-1 scans are necessary, and each scan is O(N) so the total amount of work is O(N^2) comparisons (and as many as O(N^2) swaps).

Best case:

If the aforementioned optimization is made, and the array is already sorted, then the first scan shows that the array is sorted by not performing any swaps. The algorithm terminates after N-1 comparisons and 0 swaps.

"Expected" case:

O(N^2) comparisons and swaps.
A Parallel Bubblesort Algorithm

Imagine that each element of the array is also a processor, and that adjacent processors can communicate with each other. In addition, there is a boss processor, who all the other processors obey.

Each processor is represented here by a bare-headed person or a person wearing a hat.

The bosses instructions:

3. Bare-headed people-- look to your right. If there is someone there, and they have a number that is smaller than yours, exchange numbers with them.

4. People wearing hats-- look to your right. If there is someone there, and they have a number that is smaller than yours, exchange numbers with them.

5. Raise your hand if you performed a swap during that last two steps.

If anyone raises their hands, repeat. Otherwise, the array is sorted.

*Note that everyone follows the same instructions.*
Selectionsort

Strategy: scan the array looking for the largest element that we haven’t put into the right place yet, and then put it where it belongs (the largest goes last, second largest goes second to last, etc)

(Conversely, we could look for the smallest element and put it at the front of the array.)

Note:

- Every swap puts at least one element into exactly the right position. Unlike bubblesort, which can do $O(N^2)$ swaps, selection sort only does $O(N)$.
- It never moves anything "out" of position.
- Doesn’t have a nice optimization for detecting pre-sorted data, but this could be added as a separate step (at the cost of $O(N)$ more comparisons).

The big-O of this algorithm depends on how we select the largest (or smallest) element in the unsorted part of the array. If we use the "naive" algorithm, it requires $O(N)$ comparisons, on average.

(If we use a heap-based algorithm we can do much better-- but we’ll save that trick for another algorithm.)
**Insertionsort**

for (p = 1; p < N; p++) {
    scan to the left of index p, until we find the largest j such that (j < p) and A[j] > A[p]
    
    DELETE A[p] from position p, and INSERT it at position j
}

This is probably easiest to visualize (or implement) using a linked list instead of an array.

A crucial optimization: the deletion and insertion can performed "lazily": instead of shrinking the entire array when we do the delete, and then expanding it again when we do the insertion, we can combine the two operations so that only p - j elements actually.
Analysis of Insertionsort

Like bubblesort, insertionsort requires $O(N^2)$ comparisons and $O(N^2)$ data movements (or $O(N)$ if a linked list is used to do the insertion). It can potentially cause much more data movement than selectionsort.

So, why should we be interested in yet another $O(N^2)$ sorting algorithm?

Theorem:

If every element of A is within a constant $c$ positions of its index in the sorted array, then insertionsort requires $O(N)$ comparisons and data moves.

At worst, some element might be $N-1$ positions away from their sorted positions, in which case this is $O(N^2)$. However, if $c$ is bounded by a constant, then $cN$ is $O(N)$.

Proof: (sketchy)

If a particular element is within $x$ positions of its proper position, then it will be moved to its correct position after at most $x+1$ comparisons and $x$ moves.

If it starts to the left of its proper position, it will be compared against each element inserted to its left at most once, and moved to the right one position per insertion.

If it starts to the right of its proper position, it will moved to precisely where it belongs.
Insertionsort Analysis

Theorem:
If the number of inversions in array A is bounded by a constant c, then insertion sort is O(N).

Proof: (sketchy)
We only move an element in order to reduce the number of inversions: every time an element is moved x positions, it removes x inversions (without introducing any new inversions)

Once we have moved some unknown number of elements by a total of c positions, we are finished sorting the array.

Therefore, insertionsort is very efficient if the array is "almost" sorted (i.e. the number of inversions is proportional to N, or every element is "near" its proper position). This is useful, because "almost sorted" data is quite common.
Complexity Bounds on Sorting Algorithms

Can we do better than these algorithms? Yes, but not without taking a new approach.

**Theorem:**

Any algorithm that compares swaps only adjacent elements in the array is $\Omega(N^2)$ in the general case. (perhaps faster for some arrays, like insertion sort or bubblesort, but not for *all* arrays)

**Proof:**

Each comparison detects and each swap removes *at most* one inversion. There may be as many as $N(N-1)/2$ inversions, so this many comparisons or swaps might be necessary.
Analysis (Continued...)

Imagine that the array is partitioned into three sections:

\[ \alpha \quad \beta \quad \gamma \]

The total number of inversions in the array =

- total within \( \alpha \) + total within \( \beta \) + total within \( \gamma \) +
- total between \( \alpha \) and \( \beta \) + total between \( \beta \) and \( \gamma \) +
- total between \( \alpha \) and \( \gamma \)

If the two elements in \( \beta \) are equal, then there are no inversions in \( \beta \). Otherwise, there is either 1 or 0. If there is 1 inversion, then we can remove it by swapping the two elements.

The rest of the terms in this sum are unaffected by this swap, however, so swapping the elements of \( \beta \) can only remove one inversion from the entire array.

Since there are possibly as many as \( N(N-1)/2 \) inversions, we may need to perform that many swaps.
Analysis (continued 2...)

Bubblesort clearly compares and swaps only adjacent elements, but with insertionsort it is less obvious that this is what is happening. It is easier to visualize this by imagining that during each scan from right to left to find where element \( p \) should go be inserted, we actually swap elements \( p \) and \( p-1 \) at each step.

This proof can be generalized for a larger \( \beta \)-- any fixed-size \( \beta \) will have the same problem. In order to get a faster sorting algorithm, we will need to consider algorithms that compare/swap elements that are distant from each other in the array.

**Shell sort** is one such algorithm-- it is described in the Weiss text.
Mergesort - Divide and Conquer

The idea behind mergesort is very simple:

1. Divide the array into two (or possibly more) segments of equal (or approximately equal) size.

2. Sort each segment. (using mergesort, of course)

   The base case of the recursion is when the array is length 1 or 0-- or perhaps when the array is short enough so that it is more efficient to use some other sorting algorithm (i.e. insertion sort) that is more efficient for small arrays.

3. Merge the sorted segments together.

   The merge operation is $O(N)$: merging two (or more) sorted arrays into one sorted array is easily done:

   ```
i = 0; j = 0; count = 0;
while (A and B are not empty) {
    if (A [i] < B [j])
        Merged [count++] = A [i++];
    else
        Merged [count++] = B [j++];
}
```

   (and append the leftover elements of A or B to Merged)

Note that mergesort needs an extra array of length $N$ (named "Merged" in this pseudo-code) in order to merge the two sorted arrays. It is not "in-place" like the other algorithms we have seen.
Analysis of Mergesort

• Since the algorithm was stated recursively, the easiest way to describe its running time is also recursive: the time needed to sort an array of length \( N \) is the time needed to sort each of the parts plus the time needed to perform the merge.

Let \( T(N) \) be the time necessary to sort \( N \) elements, and \( S(N) \) be the maximum amount of time needed to merge two arrays containing a total of \( N \) elements:

\[
T(N) = 2T(N/2) + S(N)
\]

Since \( S(N) = \text{big-O}(N) \), we know that \( S(N) \) is bounded by \( cN \) (where \( c \) is some constant):

\[
T(N) = 2T(N/2) + cN
\]

We can expand this repeatedly by plugging in the expression for \( T(N) \):

\[
T(N) = 2(2T(N/4) + cN/2) + cN = 4T(N/4) + 2cN
\]

And every time we do we see that another \( cN \) term appears, while \( N \) is divided by a higher power of two. Eventually, we reach the base case when we have divided \( N \) by 2 so many times that it reaches 1:

\[
T(N) = NT(1) + cN \log_2 N = \text{O}(N \log N)
\]

• Merge sort always requires roughly the same number of steps (in some cases the merging may require fewer comparisons). Merge sort is always big-Theta \( N \log N \).
Quicksort

Quicksort is another divide-and-conquer algorithm:

Algorithm:
If N = 0 or N = 1, then return.
Otherwise:

• Choose a pivot element v.
  (v does not have to be an element in A, but this is the way it is usually selected)

• Partition A into $A_{<=}$ and $A_{>}$

  $A_{<=} = \text{all elements of } A \text{ that are } <= v$

  $A_{>} = \text{all elements of } A \text{ that are } > v.$

• Quicksort $A_{<=}$ and $A_{>}$ and return the concatenation of the results.
Partitioning

By swapping within the original array A, we can use A as our work space - there is no need to allocate a scratch array, as was needed for mergesort.

How the partitioning is actually accomplished varies from implementation to implementation. Quite complicated strategies have been used in order to increase performance.
Analysis - Choosing the Pivot

• If we choose the pivot well (at best, we choose a value such that it divides the array exactly in half) then quicksort is $O(N \log N)$.
  
  • In fact, all we need to do is ensure that the ratio of the lengths of $A_{\leq}$ and $A_{>}$ are bounded by constants, and we'll get $O(N \log N)$ - for example, if $A_{\leq}$ is always $c$ times longer than $A_{>}$, or vice versa, the $O(N \log N)$ bound holds.

• If we choose the pivot badly, then we have a glorified selectionsort.

• So it is worth some effort to try to pick a pivot well.

• Common methods:
  
  • Median of 3 (or 5, 7, ...)
  
  • Randomized
Finding the Median

• If we knew what the median was, then we could pivot perfectly. It’s worth a little work for us to compute it - but not more than O(N), because this will slow us down more than it generally helps...

• Randomized Median Finding
  • Like doing "half of quicksort"
  • Expected running time - O(N)
  • Worst case - O(N²)
  • Easy to understand, code.

  • For quicksort, a chicken-and-egg problem, however-- only runs well when it is able to randomly pick good estimates of the median. If we can pick good estimates of the median, we might as well just use these estimates in quicksort!

• Deterministic Median Finding
  • Very cute, not very practical
  • Always runs in O(N)

  • However, doing "part" of the work gives a usable pair of bounds on the median-- good enough to ensure that the partitioning will be effective.
Other Sorting Topics: External Sorting

If the array is too large to fit in memory then it must be kept on a larger storage device, such as disks or tape. The cost of accessing data stored on such devices has a profound effect on the choice of sorting algorithms!

- For disks, using a B-tree to store the data is often the best way, because it minimizes disk accesses.

- For tapes, algorithms that move the through the data strictly sequentially (similar to mergesort, or the radixsort algorithm we will see later) are the best. (see Weiss 7.11)

These algorithms were once the topic of much research, and they are still very important for very large data sets.

See Knuth’s Art of Computer Programming, vol 3 for more info.
Heapsort

- We will now extend our knowledge of heaps to create a sorting algorithm that has the following properties:
  
  - $O(N \log N)$ comparisons and swaps in the worst case (like mergesort)
  
  - Requires only constant space overhead (like the simple sorts). This property implies an iterative implementation (or recursion whose depth is bounded by a constant).
    
    - Note that quicksort requires $O(\log N)$ expected and $O(N)$ worst case space overhead, for the recursive implementations we have seen.
    
    - mergesort requires $\Theta(N)$ extra space.
Heapsort Algorithm

1. Heapify the array (but use a MAX heap-order, not MIN, and use a zero-based array).

This is slightly different from the heapify we saw earlier - the resulting heap has the largest element at the root. The algorithm identical, except that we must reverse the direction of the comparisons.

2. Perform the following iteration:

   ```
   for (i = 0; i < N-2; i++) {
       v = DeleteMax;
       Array[(N-1) - i] = v;
   }
   ```

How this works:

- Since we are using a zero-based array implementation of the heap, then a heap of length N occupies positions 0..N-1 in the array.
- When we do a DELETEMAX on a heap with N elements, the resulting heap occupies indices 0..N-2.
- This means that the element we just deleted, which belongs at location N-1 in the final sorted array, can be placed in this exact index of the heap array.
- Thus the heap and the sorted array can "share" the same array. No extra space is needed.
An Example of Heapsort

Array

1. Heapify in-place (MAX order)

2. Loop

a. DELETEMAX

v=7, resulting heap

b. Put v into the newly-vacated array position:
continuing...

a. **DELETE MAX**  \( v = 5 \)
   
   \[
   \begin{array}{cccc}
   4 & 1 & 2 & \_ \_ \_ \\
   \end{array}
   \]

   b. place the 5 in the newly-vacated array position
   
   \[
   \begin{array}{cccc}
   4 & 1 & 2 & 5 & 7 \\
   \end{array}
   \]

   \[
   \begin{array}{c c c c c c}
   \text{heap} & \text{sorted array} \\
   \end{array}
   \]

a. **DELETE MAX**  \( v = 4 \)

   \[
   \begin{array}{cccc}
   2 & 1 & \_ \_ \_ & 5 & 7 \\
   \end{array}
   \]

   b. place the 4 in the new "hole"

   \[
   \begin{array}{cccc}
   2 & 1 & 4 & 5 & 7 \\
   \end{array}
   \]

   \[
   \begin{array}{c c c c c c}
   \text{heap} & \text{sorted array} \\
   \end{array}
   \]
finally...

a. \texttt{DELETEMAX} \quad v = 2

\begin{array}{c}
1 \quad 4 \quad 5 \quad 7
\end{array}

b. place the 2 in the new "hole"

\begin{array}{c}
1 \quad 2 \quad 4 \quad 5 \quad 7
\end{array}

Even though we still have a "heap" of one element, we know that this last element is the smallest, and so it belongs right where it is. We are finished.

\textbf{In fact}

We would, in actual practice, probably hand off such a small array to insertion sort, instead of using heapsort on very small arrays.
Heapsort Analysis: Time

- Heapify is $O(N)$, as we showed earlier.
- The loop has $N-1$ iterations, each including:
  - one $\text{DELETEMAX}$, which is $O(\log N)$
  - one copy, which is $O(1)$

So the total is $O(N \log N)$

Heapsort Analysis: Space

- Everything is done "in place" and iteratively, so only a few extra variables are needed (for loop counters, $v$, etc), independent of $N$.

Unfortunately...

- In practice, heapsort is slower on average than a tuned quicksort. The "percolate down" triggered by each $\text{DELETEMAX}$ is costly.

- The cost of $\text{DELETEMAX}$ can be reduced (by using different data structures, similar to the heaps we have seen) but only at the cost of using more memory (and pointers), which eliminates this benefit.