Searching and Sorting Algorithms

CS117, Fall 2004
Supplementary Lecture Notes
Written by Amy Csizmar Dalal

1 Introduction

How do you find someone's phone number in the phone book? How do you find your keys when you've misplaced them? If a deck of cards has less than 52 cards, how do you determine which card is missing?

Searching for items and sorting through items are tasks that we do everyday. Searching and sorting are also common tasks in computer programs. We search for all occurrences of a word in a file in order to replace it with another word. We sort the items on a list into alphabetical or numerical order. Because searching and sorting are common computer tasks, we have well-known algorithms, or recipes, for doing searching and sorting. We'll look at two searching algorithms and four sorting algorithms here. As we look at each algorithm in detail, and go through examples of each algorithm, we'll determine the *performance* of each algorithm, in terms of how "quickly" each algorithm completes its task.

Before we begin though, we'll take a look at how we can measure the speed of one algorithm against the speed of another algorithm.

2 Algorithm performance measurement

If we have two computer programs that perform the same task, how can we determine which program is "faster"? One way we can do this is to "time" each program, either using a stopwatch or some other timing mechanism. This works if we run the programs on the same computer under the exact same conditions, but what if we run the two programs on different computers? What happens if some other program starts running while we're trying to time one of the programs? You can see that using "clock time" as a measure of program performance has many drawbacks. But what else can we use?

We can compare the speed of two programs without using time by counting the number of instructions or operations in the two programs. Typically, the faster program has fewer operations. Often, the number of operations is directly proportional to the number of data items that the program operates on. The advantage of this measure of performance is that it is independent of the type of computer we use (processor speed, memory, disk space) and also of the "load" on the computer (what other programs are running at the same time as our program).

When comparing the performance of two search algorithms or two sorting algorithms, we concentrate on two types of operations: *data movements*, or *swaps*, and *comparisons*. Data movements occur when we replace one item in a list with another item in the list. Data comparisons occur when we compare one item in a list with either another item in the list, or an item outside the list.

Additionally, it's often sufficient to determine the *approximate* number of swaps and comparisons, rather than the exact number of swaps and comparisons, for a search or sorting algorithm. In fact, it's often sufficient to know the order of magnitude of the number of swaps (i.e., is it in the 10's or 100's or 1,000,000's) and comparisons. For example, if one algorithm requires 100 swaps, and another requires 50 swaps (100/2), then we say that these two algorithms require the same number of swaps, since both of them are on the order of 100.[1]

Now let's take a look at some common search algorithms.

[1]We say that 50, 100, 500, 75, etc. are all on the order of 100, because all of these values can be expressed as $100c$, where c is some positive constant. For example, $50 = (100)(0.5)$ and $500 = (100)(5)$.

3 Search algorithms

There are two types of search algorithms: algorithms that don't make any assumptions about the order of the list,and algorithms that assume the list is already in order. We'll look at the former first, derive the number of comparisons required for this algorithm, and then look at an example of the latter.

In the discussion that follows, we use the term *search term* to indicate the item for which we are searching. We assume the list to search is an array of integers, although these algorithms will work just as well on any other primitive data type (doubles, characters, etc.). We refer to the array elements as *items* and the array as a *list*.

3.1 Linear search

The simplest search algorithm is *linear search*. In linear search, we look at each item in the list in turn, quitting once we find an item that matches the search term or once we've reached the end of the list. Our "return value" is the index at which the search term was found, or some indicator that the search term was not found in the list.

3.1.1 Algorithm for linear search

```
for (each item in list) {
     compare search term to current item
     if match,
          save index of matching item
          break
}
return index of matching item, or -1 if item not found
```

3.1.2 Java implementation of linear search

```
public int sequentialSearch(int item, int[] list) {
    // if index is still -1 at the end of this method, the item is not
    // in this array.
    int index = -1;

    // loop through each element in the array.  if we find our search
    // term, exit the loop.
    for (int i=0; i<list.length; i++) {
        if (list[i] == item) {
            index = i;
            break;
        }
    }
    return index;
}
```

3.1.3 Performance of linear search

When comparing search algorithms, we only look at the number of comparisons, since we don't swap any values while searching. Often, when comparing performance, we look at three cases:

- Best case: What is the fewest number of comparisons necessary to find an item?
- Worst case: What is the most number of comparisons necessary to find an item?
- Average case: On average, how many comparisons does it take to find an item in the list?

For linear search, our cases look like this:

- Best case: The best case occurs when the search term is in the first slot in the array. The number of comparisons in this case is 1.
- Worst case: The worst case occurs when the search term is in the last slot in the array, or is not in the array. The number of comparisons in this case is equal to the size of the array. If our array has N items, then it takes N comparisons in the worst case.
- Average case: On average, the search term will be somewhere in the middle of the array. The number of comparisons in this case is approximately N/2.

In both the worst case and the average case, the number of comparisons is proportional to the number of items in the array, N. Thus, we say in these two cases that the number of comparisons is *order N*, or $O(N)$ for short. For the best case, we say the number of comparisons is *order 1*, or $O(1)$ for short.

3.2 Binary search

Linear search works well in many cases, particularly if we don't know if our list is in order. Its one drawback is that it can be slow. If N, the number of items in our list, is 1,000,000, then it can take a long time on average to find the search term in the list (on average, it will take 500,000 comparisons).

What if our list is already in order? Think about looking up a name in the phone book. The names in the phone book are ordered alphabetically. Does it make sense, then, to look for "Sanjay Kumar" by starting at the beginning and looking at each name in turn? No! It makes more sense to exploit the ordering of the names, start our search somewhere near the K's, and refine the search from there.

Binary search exploits the ordering of a list. The idea behind binary search is that each time we make a comparison, we eliminate half of the list, until we either find the search term or determine that the term is not in the list. We do this by looking at the middle item in the list, and determining if our search term is higher or lower than the middle item. If it's lower, we eliminate the upper half of the list and repeat our search starting at the point halfway between the first item and the middle item. If it's higher, we eliminate the lower half of the list and repeat our search starting at the point halfway between the middle item and the last item.

3.2.1 Algorithm for binary search

```
set first = 1, last = N, mid = N/2
while (item not found and first < last) {
     compare search term to item at mid
     if match
          save index
          break
     else if search term is less than item at mid,
          set last = mid-1
     else
          set first = mid+1
     set mid = (first+last)/2
}
return index of matching item, or -1 if not found
```

3.2.2 Java implementation of binary search

```
public int binarySearch(int item, int[] list) {
    // if index = -1 when the method is finished, we did not find the
    // search term in the array
    int index = -1;

    // set the starting and ending indexes of the array; these will
    // change as we narrow our search
    int low = 0;
    int high = list.length-1;
    int mid;

    // Continue to search for the search term until we find it or
    // until our ``low'' and ``high'' markers cross
    while (high >= low) {
        mid = (high + low)/2;// calculate the midpoint of the current array
        if (item < list[mid]) { // value is in lower half, if at all
            high = mid - 1;
        } else if (item > list[mid]) {
            // value is in upper half, if at all
            low = mid + 1;
        } else {
            // found it!  break out of the loop
            index = mid;
            break;
        }
    }
    return index;
}
```

3.2.3 Performance of binary search

The best case for binary search still occurs when we find the search term on the first try. In this case, the search term would be in the middle of the list. As with linear search, the best case for binary search is $O(1)$, since it takes exactly one comparison to find the search term in the list.

The worst case for binary search occurs when the search term is not in the list, or when the search term is one item away from the middle of the list, or when the search term is the first or last item in the list. How many comparisons does the worst case take? To determine this, let's look at a few examples.

Suppose we have a list of four integers: {1, 4, 5, 6}. We want to find 2 in the list. According to the algorithm, we start at the second item in the list, which is 4.[2] Our search term, 2, is less than 4, so we throw out the last three items in the list and concentrate our search on the first item on the list, 1. We compare 2 to 1, and find that 2 is greater than 1. At this point, there are no more items left to search, so we determine that 2 is not in the list. It took two comparisons to determine that 2 is not in the list.

Now suppose we have a list of 8 integers: {1, 4, 5, 6, 9, 12, 14, 16}. We want to find 9 in the list. Again, we find the item at the midpoint of the list, which is 6. We compare 6 to 9, find that 9 is greater than 6, and thus concentrate our search on the upper half of the list: {9, 12, 14, 16}. We find the new midpoint item, 12, and compare 12 to 9. 9 is less than 12, so we concentrate our search on the lower half of this list (9). Finally, we compare 9 to 9, find that they are equal, and thus have found our search term at index 4 in the list. It took three comparisons to find the search term.

If we look at a list that has 16 items, or 32 items, we find that in the worst case it takes 4 and 5 comparisons, respectively, to either find the search term or determine that the search term is not in the list. In all of these examples, the number of comparisons is $\log_2 N$.[3] This is much less than the number of comparisons required in the worst case for linear search! In general, the worst case for binary search is *order log N*, or $O(logN)$.

The average case occurs when the search term is anywhere else in the list. The number of comparisons is roughly the same as for the worst case, so it also is $O(logN)$.

In general, anytime an algorithm involves dividing a list in half, the number of operations is O(log N).

[2]Since we have an even number of terms in our list, there is no clear midpoint. When this occurs, we arbitrarily pick one of the items on either side of the actual midpoint as our midpoint. Typically, since we find the midpoint by integer division, we take the term to the left of the actual midpoint. However, our algorithm would work just as well if we selected 5 as our middle item rather than 4 in this example.

[3]The mathematical expression $y = log_2 x$ means "y is the power of 2 such that $2^y = x$."

3.3 Discussion

From the analysis above, binary search certainly seems faster and more efficient than linear search. But binary search has an unfair advantage: it assumes the list is already in sorted order! If we sort the items in a list before running the linear search algorithm, does the performance of linear search improve? The answer is no. If we know that the list is sorted, then we can modify linear search so that it stops searching the list once the items in the list are larger than the search term. But, on average, we will still end up searching roughly half of the list (N/2 comparisons), and in the worst case we will still have to search the entire list before declaring defeat (N comparisons). So, our modified linear search is still $O(N)$ for the average case and worst case.

Figure 1 illustrates why a $O(\log N)$ algorithm is faster than an $O(N)$ algorithm. Some other common functions are also illustrated for comparison purposes. As you can see, when N is small there is not much difference between an $O(N)$ algorithm and an $O(\log N)$ algorithm. As N grows large, the differences between these two functions become more pronounced.

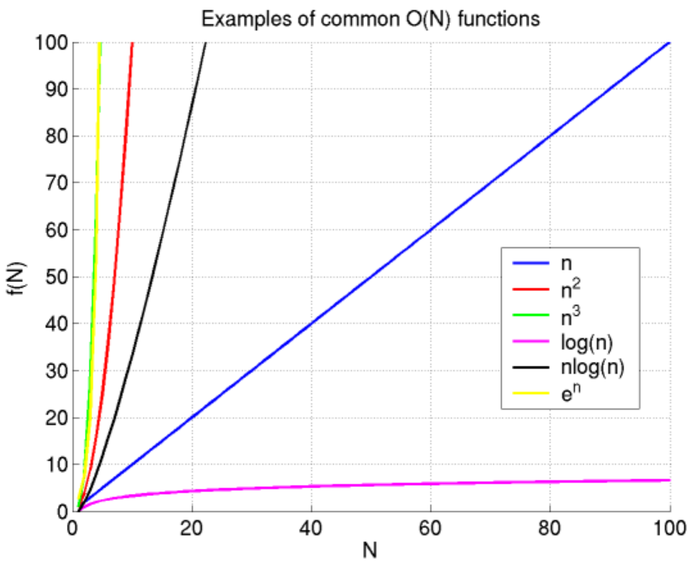

Figure 1: Some common O(N) functions

Ideally, when talking about any algorithm, we want the number of operations in the algorithm to increase *as slowly as possible* as N increases. The best-performing algorithm is $O(1)$, which means that the algorithm executes in constant time. There are very few algorithms for which this is true, so the next best algorithm is $O(logN)$.

4 Sorting algorithms

How do we put the items in a list in order? There are many different mechanisms that we can use. In this section, we'll look at four such algorithms in detail, and determine the performance of each algorithm in terms of the number of data comparisons and the number of times we swap list items.

4.1 Selection sort

The idea behind selection sort is that we put a list in order by placing each item in turn. In other words, we put the smallest item at the start of the list, then the next smallest item at the second position in the list, and so on until the list is in order.

4.1.1 Algorithm for selection sort

```
for i=0 to N-2 {
     set smallest = i
     for j=i+1 to N-1 {
          compare list[smallest] to list[j]
          if list[smallest] == list[j]
               smallest = j
     }
     swap list[i] and list[smallest]
}
```

4.1.2 Java implementation of selection sort

```
public void selectionSort(int[] list) {
    int minIndex;  // index of current minimum item in array
    boolean newMinFound = false;  // indicates if we need to swap
                                  // items or not on this pass

    for (int i=0; i<list.length-1; i++) {
        minIndex = i;
        for (int j=i+1; j<list.length; j++) {
            if (list[j] < list[i]) {
                // new minimum found, update minIndex
                minIndex = j;
                newMinFound = true;
            }
        }
        if (newMinFound) {
            swap(list,i,minIndex);
        }
        newMinFound = false;  // reset our ‘‘swap’’ flag
    }
}
```

The swap method looks like this:

```
public void swap(int[] list, int index1, int index2) {
    // Save a copy of the number at the first index, so that
    // we don’t overwrite it
    int temp = list[index1];
    // Copy number at second index into first index slot
    list[index1] = list[index2];
    // Copy number that was originally in the first index slot
    // into the second index slot.
    list[index2] = temp;
}
```

Notice that we don’t return the array after we swap the two items. This is one feature of arrays, and of objects in general: because we refer to an array by its address, if we change an item in the array within a method, the change holds outside of the method. We can do this because we are not changing the array itself; we are just changing one of the values stored in the array. So, in short, this is legal.

4.1.3 Performance of selection sort

The best case for selection sort occurs when the list is already sorted. In this case, the number of swaps is zero. We still have to compare each item in the list to each other item in the list on each pass through the algorithm. The first time through, we compare the first item in the list to all other items in the list, so the number of comparisons is (N-1). The second time through, we compare the second item in the list to the remaining items in the list, so the number of comparisons is (N-2). It turns out that the total number of comparisons for selection sort in the best case is $(N-1)+(N-2)+...+2+1$. This equation simplifies to $N(N+1)/2-1$, which is approximately N^2. Thus, even in the best case, selection sort requires $O(N^2)$ comparisons.

The worst case for selection sort occurs when the first item in the list is the largest, and the rest of the list is in order. In this case, we perform one swap on each pass through the algorithm, so the number of swaps is N. The number of comparisons is the same as in the best case, $O(N^2)$.

The average case requires the same number of comparisons, $O(N^2)$, and roughly N/2 swaps. Thus, the number of swaps in the average case is $O(N)$.

In summary, we say that selection sort is $O(N)$ in swaps and $O(N^2)$ in comparisons.

4.2 Bubble sort

The idea behind bubble sort is similar to the idea behind selection sort: on each pass through the algorithm, we place at least one item in its proper location. The differences between bubble sort and selection sort lie in how many times data is swapped and when the algorithm terminates. Bubble sort performs more swaps in each pass, in the hopes that it will finish sorting the list sooner than selection sort will.

Like selection sort, bubble sort works by comparing two items in the list at a time. Unlike selection sort, bubble sort will always compare two consecutive items in the list, and swap them if they are out of order. If we assume that we start at the beginning of the list, this means that at each pass through the algorithm, the largest remaining item in the list will be placed at its proper location in the list.

4.2.1 Algorithm for bubble sort

```
for i=N-1 to 2 {
     set swap flag to false
     for j=1 to i {
          if list[j-1] > list[j]
               swap list[j-1] and list[j]
               set swap flag to true
     }
     if swap flag is false, break.  The list is sorted.
}
```

Notice that in each pass, the largest item "bubbles" down the list until it settles in its final position. This is where bubble sort gets its name.

4.2.2 Java implementation for bubble sort

```
public void bubbleSort(int[] list) {
    // this variable will indicate if the list is sorted on this pass
    boolean isSorted;

    for (int i=0; i<list.length-1; i++) {
        isSorted = true;
        for (int j=0; j<list.length-i-1; j++) {
            if (list[j+1] < list[j]) {
                // two items are out of order, so swap them
                swap(list,j,j+1);
                isSorted = false;
            }
        }
        if (isSorted) {
            // we didn't find anything to swap, so we're done.
            break;
        }
    }
}
```

4.2.3 Performance of bubble sort

The best case for bubble sort occurs when the list is already sorted. In this case, bubble sort makes one pass through the list, performing no swaps and N-1 comparisons.

The worst case for bubble sort occurs when the list is in reverse order. In this case, every item will have to be swapped with every other item on every pass through the algorithm. As in selection sort, there will be $O(N^2)$ comparisons. The number of swaps in the worst case is greater than in selection sort: each comparison results in a swap, so there are $O(N^2)$ swaps in bubble sort!

The average case looks like the worst case: $O(N^2)$ comparisons and $O(N^2)$ swaps. The tradeoff is that we may be able to do half as many iterations, on average, as we do in selection sort. From a mathematical standpoint, however, bubble sort performs worse than selection sort in terms of the number of swaps, and the same as selection sort in terms of the number of comparisons.

The next two sorts we will look at behave much differently than selection sort and bubble sort. Selection sort and bubble sort had many similarities in terms of the number of comparisons and the number of swaps. We will see the same thing for quicksort and merge sort. We will also answer the age-old question: is quicksort really all that quick?

4.3 Quicksort

Quicksort is in fact a very fast sorting algorithm. We will see just how fast it is later on in this section. The algorithm itself is a bit tricky to understand, but it works very well.

The basic idea behind quicksort is this: Specify one element in the list as a "pivot" point. Then, go through all of the elements in the list, swapping items that are on the "wrong" side of the pivot. In other words, swap items that are smaller than the pivot but on the right side of the pivot with items that are larger than the pivot but on the left side of the pivot. Once you've done all possible swaps, move the pivot to wherever it belongs in the list. Now we can ignore the pivot, since it's in position, and repeat the process for the two halves of the list (on each side of the pivot). We repeat this until all of the items in the list have been sorted.

Quicksort is an example of a *divide and conquer* algorithm. Quicksort sorts a list effectively by dividing the list into smaller and smaller lists, and sorting the smaller lists in turn.

4.3.1 Example

It's easiest to see how quicksort operates using an example. Let's sort the list {15, 4, 23, 12, 56, 2} by quicksort. The first thing we need to do is select a pivot. We can select any item of the list as our pivot, so for convenience let's select the first item, 15.

Now, let's find the first item in the list that's greater than our pivot. We'll use the variable *low* to store the index of this item. Starting with the first item beyond the pivot (4), we find that the first item that is greater than 15 is 23. So we set *low* = 2, since 23 is at index 2 in the list.

Next, we start at the end of the list and work back toward the beginning of the list, looking for the first item that is less than the pivot. We'll use the variable *high* to store the index of this item. The last item in the list, 2, is less than our pivot (15), so we set *high* = 5, since 2 is at index 5 in the list.

Now that we've found two items that are out of place in the list, we swap them. So, now our list looks like this: {15, 4, 2, 12, 56, 23}. We also increment *low* and decrement *high* by 1 before continuing our search. Now, *low* = 3 and *high* = 4, and these correspond to 2 and 56, respectively, in the list.

Starting at index *low*, we try to find another item that's greater than our pivot. In fact, there's no other element that's greater than our pivot. We can determine this by looking at the values of *low* and *high*, and stopping when *low* becomes equal to *high*. At this point, we're done swapping; now we just have to put the pivot into the correct position. The proper position for the pivot is the position below where *low* and *high* meet: position 3, which is currently occupied by 12. So, we swap 15 and 12.

At the end of the first pass, the list looks like this: {12, 4, 2, 15, 56, 23}. Clearly, the elements are not in order, but at least they are all on the correct side of the pivot, and 15 is in fact in place. We can now safely ignore 15 for the rest of the algorithm.

Now, let's repeat the procedure on the two halves of the list: {12, 4, 2} and {56, 23}. Let's start with the second half, since there are fewer elements there.

Again, we pick the first element, 56, as our pivot. We set our *low* and *high* markers to 1 and 1, respectively. But wait a minute. Right off the bat, *low* and *high* are equal! In fact, this means that there's nothing to swap except for the pivot, and so we swap 56 and 23. This means our list is 23, 56, which is sorted, so we can safely ignore both of these for the rest of the algorithm.[4]

At the end of the second pass, the list is {12, 4, 2, 15, 23, 56}, but we still need to sort the first half of the list.

We pick 12 as our pivot point, set *low* to 1 and *high* to 2. We increment *low* and find that now *low* = *high*, which means we cannot swap anything except for the pivot. We swap the pivot with the last element in the list, which gives us {2, 4, 12}. 12 is now in the proper position, so at the end of this pass our list is {2, 4, 12, 15, 23, 56}, which is sorted.

4.3.2 Java implementation of quicksort

The Java implementation of quicksort is a bit more complex than our previous sort algorithms. In fact, we will use two different methods in order to implement quicksort. The first method is the primary method: it sets up the algorithm and then calls the other method, which does the actual sorting.

```
public void quicksort(int[] list, int low, int high) {
    if (low < high) {
        int mid = partition(list,low,high);
        quicksort(list,low,mid-1);
        quicksort(list,mid+1,high);
    }
}
```

Before we get to the partition method, let's take a close look at the quicksort() method. quicksort() actually calls itself, twice! A method that calls itself is called a *recursive* method. Recursive methods are useful when we need to repeat a calculation or some other series of tasks some number of times with a parameter that varies each time the calculation is done, such as to compute factorials or powers of a number. Recursive methods have three defining features:

- A *base case* or *stop case*, that indicates when the method should stop calling itself
- A test to determine if recursion should continue
- One or more calls to itself

In other words, each time we call the method, we change the parameters that we send to the method, test to see if we should call the method again, and provide some way to ultimately return from the method.

In the quicksort() method, the conditional statement serves as the test. The method is called twice, once for the lower half of the list and once for the upper half of the list. The "stop case" occurs when *low* is greater than or equal to *high*: in this case, nothing is done because the condition in the conditional statement is false.

The quicksort() method calls another method that selects the pivot and moves the elements in the list around as in the example. It returns the index of the pivot point in the list, which is the item in the list that is now fixed in place.

[4]In reality, we only "fix" 56 in position at this point, and we apply the algorithm to the list 23. But since this list's size is 1, there is nothing to swap, so we declare it sorted right away.

```
public int partition(int[] list, int low, int high) {
    // The pivot point is the first item in the subarray
    int pivot = list[low];
    // Loop through the array.  Move items up or down the array so that they
    // are in the proper spot with regards to the pivot point.
    do {
        // can we find a number smaller than the pivot point?  keep
        // moving the ``high'' marker down the array until we find
        // this, or until high=low
        while (low < high && list[high] >= pivot) {
            high--;
        }
        if (low < high) {
            // found a smaller number; swap it into position
            list[low] = list[high];
            // now look for a number larger than the pivot point
            while (low < high && list[low] <= pivot) {
                low++;
            }
            if (low < high) {
                // found one!  move it into position
                list[high] = list[low];
            }
        }
    } while (low < high);

    // move the pivot back into the array and return its index
    list[low] = pivot;
    return low;
}
```

As you can see, partition() performs most of the actual work of the quicksort algorithm.

4.3.3 Performance of quicksort

The best case of quicksort occurs, obviously, when the list is already sorted. For this algorithm, the best case resembles the average case in terms of performance. The average case occurs when the pivot splits the list in half or nearly in half on each pass. Each pass through the algorithm requires N comparisons. But how many passes does quicksort take? Recall that every time we divide something in half repeatedly, as in binary search, the number of operations is approximately $\log_2 N$. So, the number of passes through the algorithm is approximately $log_2 N$, and thus the number of comparisons in the average and best cases is $O(NlogN)$. The number of swaps differs in the best and average cases: for the best case, we have no swaps, but for the average case, by the same reasoning, we have $O(NlogN)$ swaps.

The worst case for quicksort occurs when the pivot is always the largest or smallest item on each pass through the list. In this instance, we do not split the list in half or nearly in half, so we do N comparisons over roughly N passes, which means that in the worst case quicksort is closer to $O(N^2)$ in performance. For the same reason, the number of swaps in the worst case can be as high as $O(N^2)$.

4.4 Merge sort

Merge sort is a neat algorithm, because it's "the sort that sorts itself". This means that merge sort requires very few comparisons and swaps; it instead relies on a "divide and conquer" strategy that's slightly different from the one that quicksort uses.

Merge sort starts by dividing the list to be sorted in half. Then, it divides each of these halves in half. The algorithm repeats until all of these "sublists" have exactly one element in them. At that point, each sublist is sorted. In the next phase of the algorithm, the sublists are gradually merged back together (hence the name), until we get our original list back — sorted, of course.

4.4.1 Example

Let's use the same example from the quicksort section, and sort this list by merge sort this time. The list is {15, 4, 23, 12, 56, 2}.

First, we divide the list in half: {15, 4, 23}, {12, 56, 2}

Then, we divide each of these in half again: {15}, {4, 23}, {12}, {56, 2}.

Now we have two lists that have one element each, and two that have two elements each. We divide the size-2 lists so that all of our lists have a single item: {15}, {4}, {23}, {12}, {56}, {2}.

In the second phase, we combine all of the one-element lists into two-element lists. Starting with {15} and {4}, we compare these values and see that 4 needs to go into our "new" list first. We copy 4 into our new list, and then we copy 15 into this list. Our new list is {4, 15}. Repeating this for {23} and {12}, we get the new list {12, 23}. Finally, repeating this for {56} and {2}, we get the new list {2, 56}.

Next, we combine {4, 15} with {12, 23}. We first look at 4 and 12, see that 4 should go into our new list first, and place 4 in location 0 in the new list. Next, we compare 15 and 12, and place 12 in location 1 in the new list. Next, we compare 15 and 23, and place 15 in location 2 in the new list. 23 is left over, so we copy this into location 3 in the new list. The list is now {4, 12, 15, 23}.

Finally, we combine {4, 12, 15, 23} with {2, 56}. We compare 4 and 2 and place 2 in location 0 in our (final) new list. We then compare 4 and 56 and place 4 in location 1 in our final list. Next, we compare 12 and 56 and place 12 in location 2 in our final list. We repeat this for the remaining elements of each list, giving us a final, sorted list that looks like this: {2, 4, 12, 15, 23, 56}.

4.4.2 Java implementation of merge sort

The merge sort implementation is similar to the quicksort implementation in two respects. First, it consists of two methods: one that "sets up" the algorithm, and one that does the actual work of the algorithm. Second, the primary or "set up" algorithm calls itself recursively.

The primary algorithm looks like this:

```
public void mergeSort(int[] list, int low, int high) {
    if (low < high) {
        // find the midpoint of the current array
        int mid = (low + high)/2;

        // sort the first half
        mergeSort(list,low,mid);

        // sort the second half
        mergeSort(list,mid+1,high);

        // merge the halves
        merge(list,low,high);
    }
}
```

This method calculates the midpoint of the array that's passed in, uses the midpoint to divide the array in half, and then calls mergeSort() for each of these halves. Again, the test is found in the conditional statement, and the stop case occurs when *low* is greater than or equal to *high*, which occurs when the size of the array is 1.

mergeSort() will be recursively called until we have arrays of size 1. At that point, the second method, merge(), is called to reassemble the arrays:

```
public void merge(int[] list, int low, int high) {
    // temporary array stores the ‘‘merge’’ array within the method.
    int[] temp = new int[list.length];

    // Set the midpoint and the end points for each of the subarrays
    int mid = (low + high)/2;
    int index1 = 0;
    int index2 = low;
    int index3 = mid + 1;

    // Go through the two subarrays and compare, item by item,
    // placing the items in the proper order in the new array
    while (index2 <= mid && index3 <= high) {
        if (list[index2] < list[index3]) {
            temp[index1] = list[index2];
            index1++;
            index2++;
        }
        else {
            temp[index1] = list[index3];
            index1++;
            index3++;
        }
    }

    // if there are any items left over in the first subarray, add them to
    // the new array
    while (index2 <= mid) {
        temp[index1] = list[index2];
        index1++;
        index2++;
    }

    // if there are any items left over in the second subarray, add them
    // to the new array
    while (index3 <= high) {
        temp[index1] = list[index3];
        index1++;
        index3++;
    }

    // load temp array's contents back into original array
    for (int i=low, j=0; i<=high; i++, j++) {
        list[i] = temp[j];
    }
}
```

4.4.3 Performance of merge sort

Since we repeatedly divide our list in half, we expect that the performance of merge sort will have a $\log_2 N$ term in it. But, is merge sort faster than quicksort, slower than quick sort, or about the same in terms of performance?

Regardless of best or worst case, the number of swaps in this algorithm is always $2(\mathit{last}\text{-}\mathit{first}+1)$ per merge, where *first* and *last* are the indexes of the first and last items in the (sub)list, respectively. In the best case, where the list is sorted, the number of comparisons per pass is $(\mathit{first}+\mathit{last})/2$. In the worst case, where the last item of one sublist precedes only the last item of the other sublist, the number of comparisons is *last-first*, which approaches N.

Thus, for the worst and average cases, the number of comparisons and number of swaps for merge sort is $O(NlogN)$. In theory, merge sort is as fast as quicksort. In practice, however, merge sort performs a bit more slowly, because of all the data copying the algorithm requires.

4.5 Summary

We've looked at four different sorting algorithms: two $O(N^2)$ sorts (selection sort and bubble sort), and two $O(NlogN)$ sorts (quicksort and merge sort). Each sorting algorithm has its advantages and disadvantages:

- *Selection sort*'s advantage is that the number of swaps is $O(N)$, since we perform at most one data swap per pass through the algorithm. Its disadvantage is that it does not stop early if the list is sorted; it looks at every element in the list in turn.
- *Bubble sort*'s advantage is that it stops once it determines that the list is sorted. Its disadvantage is that it is $O(N^2)$ in both swaps and comparisons. For this reason, bubble sort is actually the least efficient sorting method.
- *Quicksort* is the fastest sort: it is $O(NlogN)$ in both the number of comparisons and the number of swaps. The disadvantages of quicksort are that the algorithm is a bit tricky to understand, and if we continually select poor pivots then the algorithm can approach $O(N^2)$ in the worst case.
- *Merge sort* is as fast as quicksort: it is $O(NlogN)$ in both swaps and comparisons. The disadvantage of merge sort is that it requires more copying of data from temporary lists back to the "full" list, which slows down the algorithm a bit.