Lecture 12: Sorting Techniques

Merge Sort

Let's look at the recursive Divide-and-Conquer paradigm. We will first describe how it is used in Merge Sort, a popular sorting algorithm.

Suppose we have a list as shown here that needs to be sorted in ascending order:

a	b	c	d

We divide it into two equal-sized sublists:

a	b

c	d

Then, we sort the sublists recursively. At the base case of the recursion, if we see a list of size 2, we simply compare the two elements in the list and swap if necessary. Let's say that a < b, and c > d. Since we want to sort in ascending order, after the two recursive calls return we end up getting:

a	b

d	c

Now we come back to the first (or topmost) call to sort and our task is to merge the two *sorted* sublists into one sorted list. We do this using a *merge* algorithm, which simply compares the first two elements of the two sublists, repeatedly. If a < d, then it first puts a into the merged (output) list and effectively removes it from the sublist. It leaves d where it was. It then compares b with d. Let's say that d is smaller. Merge puts d into the output list next to a. Next, b and c are compared. If c is smaller, c is placed next into the output list, and finally b is placed. The output will be:

a	d	c	b

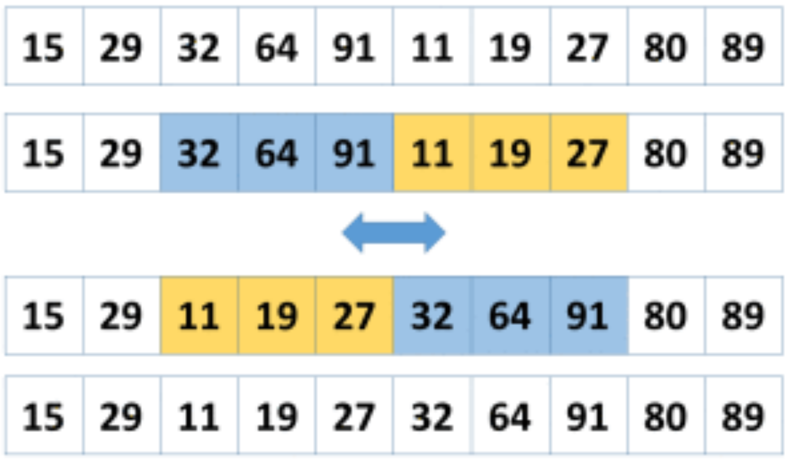
Note that if the number of elements is odd, the subarrays will not be equal in size but differ in size by 1.

Code for Merge Sort that can be found in **mergesort.py**. As you can see in the code during the `merge` step we need to create a new list `result` (Line 2) and return that. The amount of intermediate storage required for Merge Sort therefore grows with the length of the list to be sorted.

In Place Merging

It is possible to perform an *in place* merge sort, where only a fixed amount of intermediate storage is required, and a copy of the list need not be made. Such a merge routine is given in **mergesort-inplace.py**. *The algorithm for merge is itself recursive and is described below.*[1]

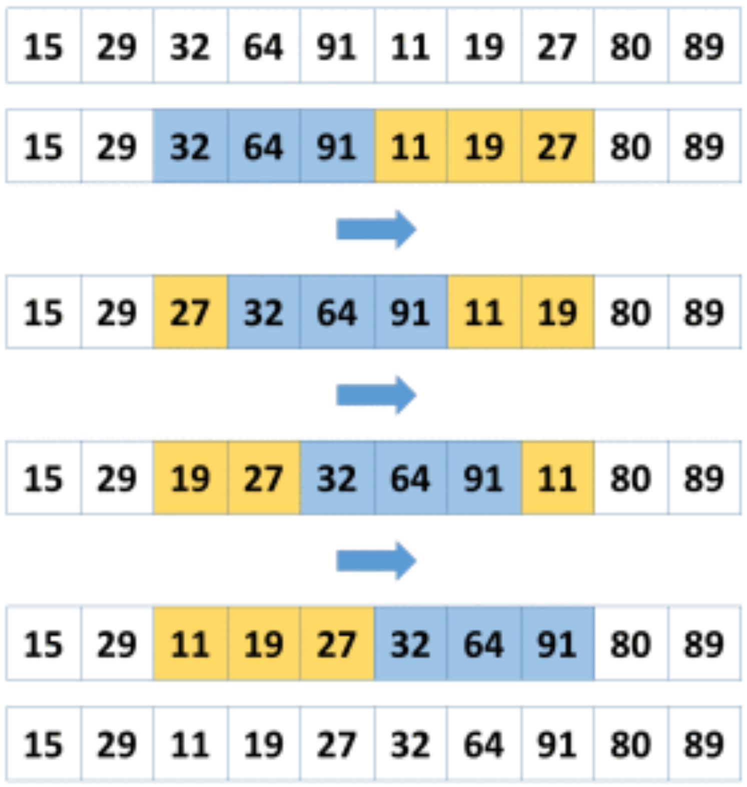
We'll first describe rotations as they are used heavily in the in place merging process. For in-place merging, rotations are used to swap the elements in two **adjacent** sublists, which are **equal in length**. As you will soon see, this technique is used to move a large number of elements at once. See the example below where we have performed a swap (via a rotation as you will see).



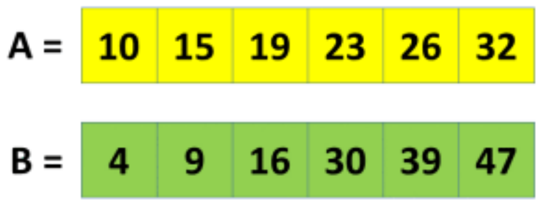
The above is the basic operation of in-place merging. To rotate elements in a sublist means to shift the elements some number of places to right and elements at the end wrap around to the beginning (and vice versa). So if you have the list [1,2,3,4,5], shifting it one place to the right will give you [5,1,2,3,4].

The fact that the two ranges being swapped are **adjacent** is important in calling this operation a rotation. In the following graphic, the elements are shifted one place to the right three times, achieving the same result as before.

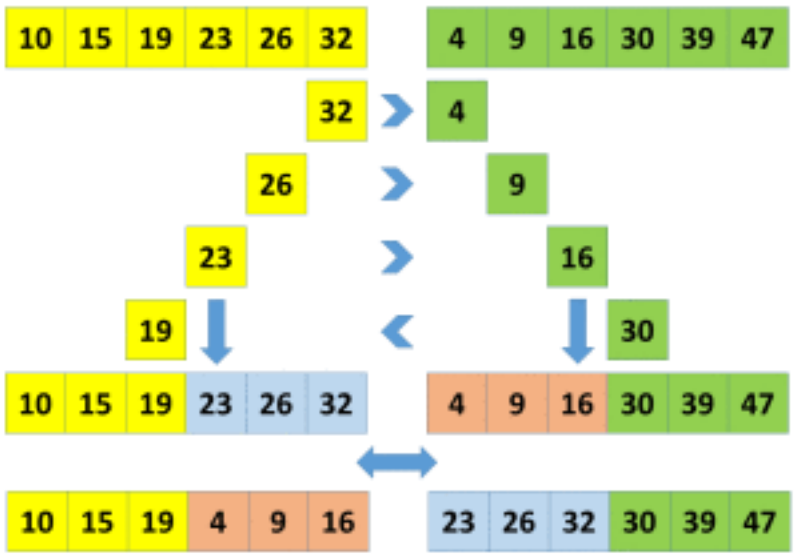
[1] https://xinok.wordpress.com/2014/08/17/in-place-merge-sort-demystified-2/



Now we can explain how to use rotations for in-place merging. Let us define two sorted sublists, A and B that we want to merge in-place.

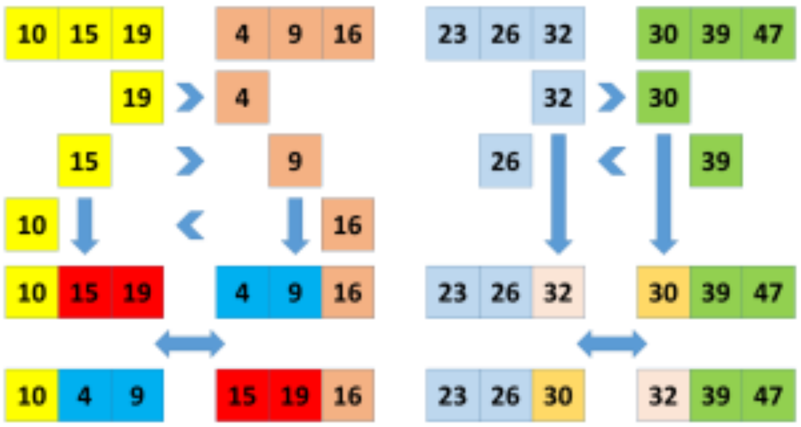


The trick is to swap the **largest** elements in A for the **smallest** elements in B. A linear search can find the range of elements to rotate. Start from the middle and extend outwards, until you find an element in A which is less than an element in B. Rotate the elements between these bounds.



That's the basic operation. However, something special happened. **Of the two ranges, the smallest elements are in A and the largest elements are in B, albeit not in order.** *This means you can continue to merge A and B independently*, so all you're left with is two smaller merges. This also means that in-place merging is a recursive function in itself.

By **recursively** applying rotations in this manner, eventually you'll merge the two ranges. Let's apply this process once more and see what happens:

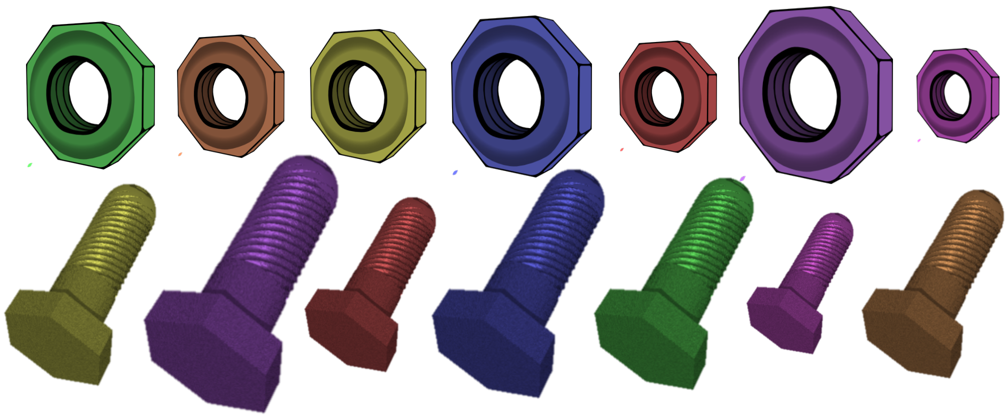


Just like that, the right-hand side is already sorted and the left-hand side is almost sorted. Two more rotations on the left-hand side and the merge procedure would be complete.

While this is an elegant algorithm and worth understanding, it turns out in place merging is not particularly useful. This is because the number of swaps required can grow in the worst case as n^2 where n is the number of elements in the list. In place merge sort is therefore less efficient than allocating memory and using n comparisons for the merge ☹

Nuts and Bolts Puzzle

A handyman has a whole collection of nuts and bolts of different sizes in a bag. Each nut is unique and has a corresponding unique bolt, but the disorganized handyman has dumped them all into one bag and they are all mixed up. How best to "sort" these nuts and attach them to their corresponding bolts?



Given n nuts and n bolts, the handyman can pick a nut and try it with each bolt and find the one that fits the nut. Then, he can put away the nut-bolt pair, and he has a problem of size $n - 1$. This means that he has done n "comparisons" to reduce the problem size by 1. $n - 1$ comparisons will then shrink the problem size to $n - 2$, and so on. The total number of comparisons required is $n + (n - 1) + (n - 2) + \ldots + 1 = n(n+1)/2$. Can one do better? More concretely, can one split the nuts and bolts up into two sets, each of half the size, so we have two problems of size n/2 to work on? This way, if the handyman has a helper, they can work in parallel.

Unfortunately, simply splitting the nuts up into two equal sized piles A and B and the bolts into two equal sized piles C and D does not work. If we group nuts and bolts corresponding to A and C together into a nut-bolt pile, it is quite possible that a nut in A may not fit *any* bolt in C; the correct bolt is in D. However, what we can do is to pick a bolt, we will call it the **pivot bolt**, and use it to determine which nuts are smaller, which one fits exactly, and which nuts are bigger. We separate the nuts into three piles in this way, with the middle pile being of size 1 and containing the paired nut. Therefore, in this process we have discovered one nut-bolt pairing. Using the paired nut, that we will call the **pivot nut**, we can now split the bolts into two piles, the bolts that are bigger than the pivot nut, and those that are smaller. The bigger bolts are grouped with the nuts that were bigger than the pivot bolt, and the smaller bolts are grouped with the nuts that were smaller than the pivot bolt. We now have a pile of "big" nuts and "big" bolts, all together, and a pile of small nuts and small bolts all together. Depending on the choice of the pivot bolt, there will be a differing number of nuts in the two piles. However, we are guaranteed the same number of nuts and bolts in each pile, and moreover, the nut corresponding to any bolt in the pile is guaranteed to be in the same pile!

In this strategy, we had to make *n* comparisons given the pivot bolt to split the nuts into two piles. In the process we discover the pivot nut. We then make *n* – 1 comparisons to split the bolts up and add them to the nut piles. That is a total of 2*n* – 1 comparisons. Assuming we chose a pivot nut that was middling in size, we have two problems roughly of size *n*/2, which we can divide again using roughly *n* comparisons to problems of size *n*/4. The cool thing is the problem sizes halve at each step, rather than only shrinking by 1. For example, suppose *n* = 100. The original strategy requires 5,050 comparisons. In the new strategy, using 199 comparisons, we get two subproblems each roughly of size 50. Even if we use the original strategy for each of these subproblems, we will only require 1225 comparisons for each one, for a total of 199 + 1225 * 2 = 2649 comparisons. Of course, we can do a recursive Divide and Conquer. (The analysis to show that the comparisons in the new strategy applied recursively grows as *n* log *n* as compared to n^2 in the original strategy is beyond the scope of this class.)

Interestingly, this puzzle has a deep relationship with perhaps the most widely used sorting algorithm quicksort. The selection of an arbitrary bolt in the recursive Divide and Conquer strategy can be viewed as a selection of a **pivot element**.

Quicksort

Recall in merge sort, we divide the array up into two equal sized subarrays and this division may be such that there are elements in each subarray that are bigger or smaller than elements in the other subarray. Therefore, after the subarrays are sorted, we need a merge operation to find the correct locations for each element in the original array. Mergesort is not an in-place algorithm, since it needs different array storage for the merge step and subarray sorting. We did look at a version of Mergesort that was in-place, but the merge step was not efficient.

In quicksort, we will spend more time in partitioning the array so the merge step becomes trivial. In particular, suppose we have an array with *unique* elements:

a	b	c	d	e	f	g	h

that we wish to sort in ascending order. We will choose an **arbitrary** pivot element, say, g. Now we will partition the array into two subarrays where the left subarray has elements less than g, and the right subarray has elements greater than g. The two subarrays are not sorted. We can now represent the array as:

Elements less than g	g	Elements greater than g

We can sort the left subarray without affecting the position of g, and similarly for the right subarray. Once these two subarrays have been sorted, we merely concatenate the results; there is no sophisticated merge step as in the mergesort algorithm. A simplistic implementation of quicksort is shown in **quicksort.py**, which is not in-place. However, the main advantage of quicksort is that the partitioning step, i.e., going from the original array to the one with g's location fixed and the two subarrays unsorted but satisfying ordering relationships with g, can be done in place as shown in **quicksort-inplace.py**. This is quite clever code and worth reading carefully and understanding. This code assumes that the pivot is selected as the *last* element of the array, so in the above array, it would be h.

```
def partition(lst, start, end):
    pivot = lst[end]
    bottom = start - 1
    top = end
    done = False
    while not done:
        while not done:
           bottom += 1
            if bottom == top:
                done = True
                break
            if lst[bottom] > pivot:
                lst[top] = lst[bottom]
                break
        while not done:
            top -= 1
            if top == bottom:
                done = True
                break
            if lst[top] < pivot:
                lst[bottom] = lst[top]
                break
    lst[top] = pivot
    return top
```

The first thing to observe about this code is that it works exclusively on the input list `lst` and does not allocate additional list/array storage to store list elements other than the variable `pivot` that stores one list element. Furthermore, only list elements between the start and end indices are modified. This procedure uses *in-place* pivoting – the list elements exchange positions and are not copied from one list to another wholesale as in the first version of the procedure.

It is easiest to understand the procedure with an example. Suppose we want to sort the following list:

```
a = [4, 65, 2, -31, 0, 99, 83, 782, 1]
quicksort(a, 0, len(a) - 1)
```

How exactly is the first pivoting done in-place? The pivot is the last element `1`. When `partition` is called for the first time, it is called with `start = 0` and `end = 8`. This means that `bottom = -1` and `top = 8`. We enter the outer **`while`** loop and then the first inner **`while`** loop (Line 7). The variable `bottom` is incremented to `0`. We search rightward from the left of the list for an element that is greater than the pivot element `1`. The very first element `a[0] = 4 > 1`. We copy over this element to `a[top]`, which contains the pivot. At this point, we have element `4` duplicated in the list, but no worries, we know what the pivot is, since we stored it in the variable `pivot`. If we printed the list and the variables `bottom` and `top` after the first inner **`while`** loop completes, this is what we would see:

```
[4, 65, 2, -31, 0, 99, 83, 782, 4] bottom = 0 top = 8
```

Now, we enter the second inner **`while`** loop (Line 15). We search moving leftward from the right of the list at `a[7]` (the variable `top` is decremented before the search) for an element that is less than the pivot `1`. We keep decrementing `top` till we see the element `0`, at which point `top = 4`, since `a[4] = 0`. We copy over element `0` to `a[bottom = 0]`. Remember that `a[bottom]` was copied over to `a[8]` prior to this so we are not losing any elements in the list. This produces:

```
[0, 65, 2, -31, 0, 99, 83, 782, 4] bottom = 0 top = 4
```

At this point we have taken one element `4` that is greater than the pivot `1` and put it all the way to the right of the list, and we have taken one element `0` which is less than the pivot `1` and put it all the way to the left of the list.

We now go into the second iteration of the outer **`while`** loop. The first inner **`while`** loop produces:

```
[0, 65, 2, -31, 65, 99, 83, 782, 4] bottom = 1 top = 4
```

From the left we found `65 > 1` and we copied it over to `a[top = 4]`. Next, the second inner **`while`** loop produces:

```
[0, -31, 2, -31, 65, 99, 83, 782, 4] bottom = 1 top = 3
```

We moved leftward from `top = 4` and discovered `-31 < 1` and copied it over to `a[bottom = 1]`.

In the second outer **`while`** loop iteration we moved one element `65` to the right part of the list where beyond `65` all elements are greater than the pivot `1`. And we moved `-31` to the left of the list where all elements to the left of `-31` are less than the pivot `1`.

We begin the third iteration of the outer **`while`** loop. The first inner **`while`** loop produces:

```
[0, -31, 2, 2, 65, 99, 83, 782, 4] bottom = 2 top = 3
```

We discovered `a[bottom = 2] = 2` > 1 and moved it to `a[top = 3]`. The second inner **`while`** loop decrements `top` and sees that it is equal to `bottom` and sets `done` to **`True`**, and we break out of the second inner **`while`** loop. Since `done` is **`True`**, we do not continue the outer **`while`** loop.

We set `a[top = 2] = pivot = 1` (Line 23) and return the index of the pivot `1`, which is `2`. The list `a` now looks like:

```
[0, -31, 1, 2, 65, 99, 83, 782, 4]
```

We have indeed pivoted around the element `1`.

Of course, all we have done is split the original list `a` up into two lists that are of size 2 and size 6. We need to recursively sort these sublists. For the first sublist of 2 elements, we will pick `-31` as the pivot and produce `-31, 0`. For the second sublist, we will pick `4` as the pivot and the process continues.

It is important to note that `partition` assumes that the pivot is chosen to be the end of the list. So the assignment `pivot = lst[end]` (Line 2) is crucial to correctness!

Quicksort is hard to analyze for algorithmic complexity. This is partly because it is unclear what the sizes of the two subarrays are. If, in our example, g was the largest element in the array, then the left subarray would have $n - 1$ elements and the right subarray would have 0! Like in the Nuts and Bolts puzzle, we want to pick an element with medium size. Turns out that quicksort's worst-case complexity is worse than mergesort (the version that uses space for an additional list of n elements), but on average it performs better if a random pivot element is picked or the input is randomized. We will leave such analysis for when you take 6.046!