Sorting 4: Merge Sort and Quicksort

Instructors: Sam McCauley and Dan Barowy

March 14, 2022

Admin

- Remember to bring questions on Wednesday
- No class Friday
- Any questions?

Merge Sort

Merge Sort

- We can sort faster than $O(n^2)$!
  - Can sort in $O(n \log n)$ time
- Merge sort is a classic way to do that
- Very fast in practice; used as the basis of many state of the art sorting algorithms

Sorting Recursively?

- Can we use recursion to sort?
- Base case?
  - One-element array is always sorted.
- How can we create a smaller subproblem?
  - Could do one element smaller, but that gives us $O(n^2)$ (like in selection sort)
  - Can we divide the size by 2 like in Binary Search?

Merge Sort Hint

- Let's say I have two sorted arrays
- How fast can I sort the concatenation of these arrays?

-3	17	21	40

-4	10	11	13

Goal:

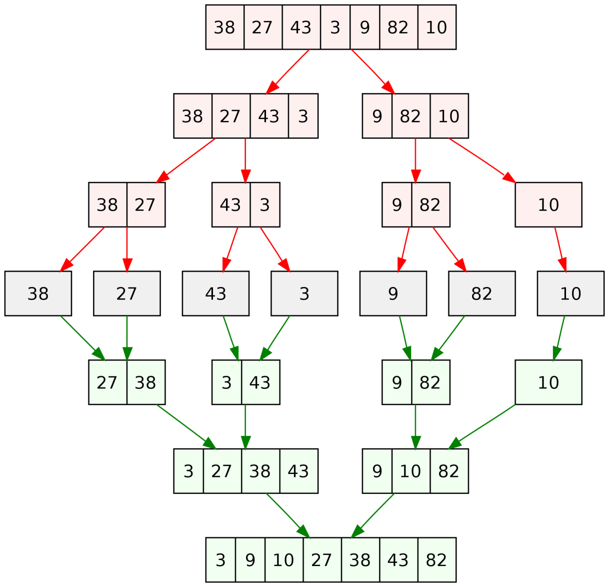
-4	-3	10	11	13	17	21	40

- Hint: where does the *smallest* element between the two arrays reside?
- Repeatedly take the smaller of the first remaining element in the two arrays. Takes $O(n)$ time

Merge Sort

- If array has size 1, just return
- Split array into two halves
- Sort each half
  - How?
  - Using Merge Sort!
- Then merge the resulting arrays together in $O(n)$ time by walking through them
- Let’s do an example on the board

Merge Sort Image



Merge Sort Code

- Goal of the code: efficient implementation
- One tricky part: implementation only uses two *total* arrays
    - One new array per recursive call would be $O(n)$ arrays and as much as $O(n)$ space—inefficient!
- Idea: pass around two arrays
- Each recursive call is passed two indices `low` and `high`. This recursive call is free to use any indices of either array between `low` and `high`
    - In other words: a postcondition of the method is that all array indices outside of `low..high` must remain the same

Merge Sort Code

```
void mergeSortRecursive(int data[], int temp[], int low, int high) {
   int n = high-low+1;
   int middle = low + n/2;
   int i;
   if (n < 2) return;
   // move lower half of data into temporary storage
   for (i = low; i < middle; i++) {
      temp[i] = data[i];
   }
   mergeSortRecursive(temp,data,low,middle-1);
   mergeSortRecursive(data,temp,middle,high);
   // merge halves together
   merge(data,temp,low,middle,high);
}
```

Merge `temp[low..middle-1]` and `data[middle..high]`

```
void merge(int data[], int temp[], int low, int middle, int high) {
   int ri = low; // result index
   int ti = low; // temp index
   int di = middle; // destination index
   // while two lists are not empty merge smaller value
   while (ti < middle && di <= high) {
      if (data[di] < temp[ti]) {
         data[ri++] = data[di++]; // smaller is in high data
      } else {
         data[ri++] = temp[ti++]; // smaller is in temp
      }
   }
   // possibly some values left in temp array
   while (ti < middle) {
      data[ri++] = temp[ti++];
   }
}
```

Merge Sort Discussion

- *All work* is done during merging or copying
- “Splitting” part of the algorithm just recurses down to arrays of size 1
- How can we prove that merge sort correctly sorts the arrays?
    - Induction!
    - Strong or weak?
    - What will our proof look like?

Merge Sort Running Time

- First: let's draw out the recursive calls that merge sort makes
- How much time do they take?
- Can we prove that merge sort takes $O(n \log n)$ time using induction?
- Idea: we'll prove that the *number of elements moved* is at most $\frac{3}{2}n \log_2 n$
- Do you agree that this bounds the running time? I.e. that the running time of merge sort is $O(\#\text{ number of elements moved})$?
- Note that calling `merge` on two arrays of total size n takes n movements, and copying an array of size n takes n movements.

Merge Sort Running Time: Induction

- Base case: merge sort on an array of size 1 takes $0 = \frac{3}{2}\log_2 1$ element movements
- Inductive hypothesis: number of elements moved calling merge sort on an array of size k is $\frac{3}{2}k\log_2 k$
- Inductive Step: assume I.H. for all $1 \leq i \leq k$ for some $k \geq 1$. (We'll prove the I.H. for $k+1$.)
- (This is a fairly difficult induction, but it demonstrates some interesting concepts)

Merge Sort Running Time: Inductive Step

- Let's merge sort an array of size $k + 1$.
- Copying the elements moves $\frac{k+1}{2}$ elements.
- How many elements do the recursive calls to Merge Sort move?
  - Merge Sort recursive calls are on arrays of size $\frac{k+1}{2}$, so each moves $\left(\frac{3}{2}\right)\left(\frac{k+1}{2}\right)\log_2 \frac{k+1}{2}$ elements by I.H.
- Merging the arrays moves $\leq k + 1$ elements.
- Total elements moved:

$$\frac{k+1}{2} + (k+1) + \frac{3(k+1)}{2} \log_2 \frac{k+1}{2}$$

Merge Sort Running Time: Inductive Step

- Total elements moved:

$$
\begin{aligned}
\frac{k+1}{2} + (k+1) + \frac{3(k+1)}{2} \log_2 \frac{k+1}{2} &= \frac{k+1}{2} + (k+1) + \frac{3(k+1)}{2} (\log_2(k+1) - 1) \\
&= \frac{3(k+1)}{2} + \frac{3(k+1)}{2} \log_2(k+1) - \frac{3(k+1)}{2} \\
&= \frac{3}{2}(k+1) \log_2(k+1)
\end{aligned}
$$

Phew

Wrapping Up Merge Sort

- $O(n \log n)$ time worst case
- Proof is a bit harder than what we've seen in the past. But only uses tools you're familiar with!
- Insertion sort is better for small arrays (size of 10–100 or thereabouts). Any ideas why?

QuickSort

QuickSort

- Last sorting algorithm we'll discuss in this class
- Probably the fastest overall
- Can be implemented *in-place* (without an extra array like MergeSort used)
- Java's built-in sorting method uses QuickSort, as do many other library sorting algorithms
    - (Not python; python uses TimSort which is based on Merge Sort)

Basic Quicksort idea

- Pick a *pivot* from the array (we'll use the leftmost value)
- Goal: can we put just this pivot in its correct place in the array?
- Followup: can we then place all elements around the pivot? I.e.: smaller things to the left of the pivot and larger things to the right.
- Let's say we've done all that. What do we need to do to finish sorting?
  - Recurse! On the elements to the left of the pivot, and the elements to the right of the pivot
  - Base case: a 1-element array is already sorted

Quicksort Example

- Let’s do an example on the board
- Now let’s look at the code. Let’s say there’s a `partition` method that puts the pivot in the correct place, places all other array elements to the left or right of the pivot, and returns the location of the pivot

Quicksort Code

```
public static void quickSort(int data[], int n) {
   quickSortRecursive(data,0,n-1);
}

// @post data[left..right] is in sorted order
private static void quickSortRecursive(int data[],int left,int right) {
   int pivot; // the final location of the leftmost value
   if (left >= right)
      return;
   pivot = partition(data,left,right); /* 1 - place pivot */
   quickSortRecursive(data,left,pivot-1); /* 2 - sort small */
   quickSortRecursive(data,pivot+1,right);/* 3 - sort large */
}
```

Partition

- The heart of the quicksort algorithm
- Partition is pretty straightfoward to implement if we can copy the array over (like we did in Merge Sort)
- Let's look at the `structure5` implementation, which does it all in place. This implementation is a bit involved!

Partition Implementation

```java
// post: data[left] placed in the correct (returned) location
private static int partition(int data[], int left, int right) {
   while (true) {
      // move right "pointer" toward left
      while (left < right && data[left] < data[right]) right--;
      if (left < right) swap(data,left++,right);
      else return left;
      // move left pointer toward right
      while (left < right && data[left] < data[right]) left++;
      if (left < right) swap(data,left,right--);
      else return right;
   }
}
```

Quicksort Performance

- Let's say that our pivot is always the median element. What is Quicksort's performance?
  - Basically the same as Merge Sort: do $O(n)$ work, and recurse on two arrays of size $n/2$
  - $O(n \log n)$ total time
- Let's say our pivot is always the *smallest* element. What is Quicksort's performance?
  - Basically the same as Selection Sort: each time our array only gets one smaller!
  - $O(n^2)$ time
- What if our elements are randomly jumbled? In short: we do about as well as Merge Sort on average; $O(n \log n)$ time

Quicksort: Some Practical Ideas

- $O(n^2)$ performance on an already-sorted array is terrible!
- Can we get $O(n \log n)$ performance on an already-sorted array? Can we get $O(n \log n)$ performance on almost all practical arrays?
- One idea: pick pivot randomly. Then we get $O(n \log n)$ *average* running time.
- Another idea: pick several different potential pivots, and choose the median of these pivots.
  - Java's Quicksort does this; works well in practice
  - Still $O(n^2)$ worst case, but this worst case is a very specific, very carefully structured array
  - Downside: takes lots of time to pick pivots
- Final idea: randomly permute the array before running quicksort
  - Surprisingly, has practical merit
  - But it's expensive!

Wrapping up Sorting

- Selection sort: $O(n^2)$, easy to understand the invariant. In place algorithm
- Insertion Sort: $O(n^2)$, better performance than Selection Sort in some case. In place algorithm
- Merge Sort: $O(n \log n)$ in all cases; requires an extra array of size n
- Quick Sort: $O(n^2)$ in the worst case, but often $O(n \log n)$. In place algorithm if `partition` is implemented in place