5.1 Sorting

Sections (like this one) which marked with a vertical line on the left side are commentary — discussions about the algorithm development process that wouldn't be part of a writeup.

Establish the problem.

Defining — and understanding — the precise task to be solved is an essential prerequisite for algorithm development.

- *Specifications.*

  State complete specifications for the problem. What is the problem? What do you start with (input) and what is the end result (output)? What are the legal input instances and the required output for each?

  Task: sort an array of numbers in increasing (non-decreasing) order
  Input: array A of n numbers
  Output: A, sorted

- *Examples.*

  If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

  The notions of sorted order and non-decreasing are common enough that examples aren't going to add anything to the task as already described.

Identify avenues of attack.

Determine the algorithmic approach(es) to try.

- *Targets.*

  Divide-and-conquer algorithms often seek to improve on a polynomial-time iterative brute-force running time. If there aren't other targets, identify the brute force algorithm and its running time.

  A brute-force approach for sorting: repeatedly find the smallest remaining element and append it to the list of sorted elements. (i.e. selection sort) This is $O(n^2)$.

- *Patterns.*

  Consider the divide-and-conquer patterns.

  There are two main patterns: easy split and easy merge. The goal in this step is to think about how this task might fit into the patterns — we're not trying to come up with an algorithm, just to identify what pieces need to be sorted out to complete an algorithm in a particular vein.

  Easy split is potentially applicable when the input is a collection of elements — split the elements into two groups (first half, second half). Easy merge is potentially applicable when the output is a collection of elements — have one friend produce the first part of the solution and the other friend produce the second part.

  Easy split: The input is an array of elements, so the easy split is to split the array in half. Then have friends sort each half, with the final step being to combine the two sorted lists into one.

  Easy merge: The result is a sorted array, so the easy merge is to have one friend produce the first part of the sorted array and the other friend produce the rest. Since the smaller elements are first in a sorted array, this means we need to split the input elements into smaller and larger with one friend sorting the smaller ones (first part of the result) and the other sorting the larger ones (the rest of the result).

> If more than one pattern seems applicable, pick one that seems easiest to proceed with. (And come back to a different one if the first attempt doesn't pan out.) Splitting into smaller values and bigger values seems easier than combining sorted lists, so let's try easy merge.

Define the algorithm.

> Assemble the pieces of the algorithm according to the template for the particular paradigm and pattern, culminating in a statement of the algorithm itself.

- *Size.*

  > What is the size of the problem, and what constitutes a smaller problem? What is the simplest/smallest instance of the problem?

  The size is the number of elements to sort in the array.

- *Generalize / define subproblems.*

  > Define the subproblem — the task, the input, and the output. Write this as a function definition where parameters and return values are explicitly identified.

  > The friends always have the same task as we do — if we are sorting, they are sorting. But it is a generalized version of the original task — if the original task is to sort the whole array, a generalized version is to sort some portion of the array. This is generalized because if you can sort a portion of the array, you can also sort the whole array — but if you can only sort the whole array, there's no way to only sort a portion.

  Subproblem task: sort a portion of an array of numbers in increasing (non-decreasing) order
  Subproblem input: array A of n numbers; values $low \leq high$
  Subproblem output: $A[low..high]$ (inclusive), sorted

- *Base case(s).*

  > Address how to solve the smallest problem(s). This is often trivial, or is solved via brute force.

  > The smallest meaningful size of problem would be $n = 0$ (nothing to sort) or $n = 1$ (one element). Since $n = 0$ is arguably a bit silly, let's choose $n = 1$ as the smallest problem.

  For $n = 1$ i.e. $low = high$, do nothing — a single element is sorted.

- *Main case.*

  > Address how to solve a typical large problem instance (one that is not a base case). Specify how many friends are needed and what subproblem is handed to each friend, as well as how the results the friends hand back are combined to solve the problem.

  > The easy merge pattern gives the outline of the main case: split the input into smaller elements and bigger elements, sort each group, and then just append one sorted list to the other. Splitting into smaller and bigger begs the question of smaller/bigger than what? We could pick the first element and use that to partition the rest.
  >
  > Our result should be that $A[low..high]$ is sorted, so if the friend that gets the smaller elements sorts them into $A[low..p-1]$ and the friend that gets the larger elements sorts them into $A[p+1..high]$ (where the element used to partition the other elements ends up in $A[p]$).

  ```
  // split the input into smaller and bigger
  pivot ← A[low]
  partition A[low..high] so that
    A[low..p − 1] contains the elements of A[low..high] < pivot,
  ```

$A[p]$ `contains` *pivot*`, and`
$A[p+1..high]$ `contains the elements of` $A[low..high] > pivot$

```
// recursively sort the two parts
sort(A,low,p-1)
sort(a,p+1,high)

// A[low..high] is now sorted
```

> The algorithm should be specified in enough detail to be able to establish correctness, but not more detail than that. The partition step above is not fully specified in terms of the process of how to carry out the partition, but the end result of the partition is understood and that's all that is needed to continue on with the algorithm — what is necessary for correct functioning is how the elements of A are arranged after the partition step, not the process by which they got arranged that way.

- *Top level.*

  > The top level puts the context around the recursion.

  - *Initial subproblem.*

    > Specify the inputs and parameters for the initial subproblem — the one whose solution solves the original problem.

    ```
    sort(A,0,n-1)
    ```

  - *Setup.*

    > Whatever must happen before the initial subproblem is solved.

    Nothing to do — sort A as given.

  - *Wrapup.*

    > Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

    Nothing to do — A being sorted is the goal.

- *Special cases.*

  > Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).

  > Typical special cases to consider for collections are empty collections and collections with a single element. Typical special cases to consider for ordering tasks include duplicate elements.

  $n = 0$ — Having an empty array for the initial input is possible and could be discounted as illegal input, but when the elements are partitioned in the main case it is possible that the pivot picked is the smallest or largest and thus one of the friends gets a subproblem with no elements.

  $n = 1$ — Already handled as a base case.

  Duplicates — Non-decreasing order already accounts for the possibility of duplicates. The partitioning needs to be updated to put values equal to the pivot into either the smaller elements group or the larger elements group.

- *Algorithm.*

> Assemble the algorithm from the previous steps and state it.
>
> There shouldn't be new elements here, instead bring together the base case(s), main case, and top level with any handling needed for special cases and state the whole algorithm.

```
algorithm sort(A)
 Task: sort an array of numbers in increasing (non-decreasing) order
 Input: array A of n numbers
 Output: A, sorted

 sort(A,0,n-1)
```

```
algorithm sort(A,low,high)
 Task: sort a portion of an array of numbers in increasing (non-decreasing) order
 Input: array A of n numbers; values low ≤ high
 Output: A[low..high] (inclusive), sorted

 if low > high           // n=0
   return

 else if low == high     // n=1
   return

 else
   // split the input into smaller and bigger
   pivot ← A[low]
   partition A[low..high] so that
     A[low..p − 1] contains the elements of A[low..high] ≤ pivot,
     A[p] contains pivot, and
     A[p + 1..high] contains the elements of A[low..high] > pivot

   // recursively sort the two parts
   sort(A,low,p-1)
   sort(a,p+1,high)

   // A[low..high] is now sorted
```

Show termination and correctness. Show that the algorithm produces a correct solution.

- *Termination.*
  > Show that the recursion — and thus the algorithm — always terminates.

  - *Making progress.*
    > Explain why what each of your friends get is a smaller instance of the problem.

    The pivot is not included in the elements handed off to the friends to sort, so, since the total number of elements passed to friends is one less than the number of elements we got, both friends have to fewer elements than we got.
  - *The end is reached.*
    > Explain why a base case is always reached.

    For $n \geq 2$ (the main case), if the pivot is the smallest or largest element, the elements will be split into problems of size 0 and $n - 1 \geq 1$. Between the base cases and the main case, every value ≥ 0 is covered so it's not possible to miss a base case.

- *Correctness.*

Show that the algorithm is correct.

- *Establish the base case(s).*

  Explain why the solution is correct for each base case.

  If $n = 0$, there are no elements to sort and nothing to do. If $n = 1$, there is one element and a single element is sorted no matter what it is.

- *Show the main case.*

  Assume that the friends return the correct results for their subproblem, and explain why the correct answer is then produced from those results.

  Everything in $A[low..p]$ is less than or equal to the pivot and everything in $A[p+1..high]$ is greater than the pivot, so as long as those two portions are themselves in sorted order (which they are if the friends do their jobs correctly), everything in $A[low..high]$ is sorted.

- *Final answer.*

  Explain why the top level — the setup plus a correct solution to the initial subproblem followed by the wrapup — means that the final result is a correct answer to the problem.

  The algorithm sorts $A[low..high]$ inclusive, so $sort(A, 0, n-1)$ includes both $A[0]$ and $A[n-1]$ and that covers everything in A.

Determine efficiency.

Evaluate the running time and space requirements of the algorithm.

- *Implementation.*

  Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic details that are needed in order to establish the running time.

  The only step that needs clarification as to process is the partition step — everything else in the algorithm is either a simple statement or a method call. Consider a brute force solution: every element in A[low..high] must be compared to *pivot* in order to determine which group it goes into, so that's the main process: for each element in $A[low..high]$, compare it to *pivot* and add it to the smaller group or the larger group.

  Handling the in-place part of the partition is potentially tricky, but since we're already going through every element $A[low..high]$, adding each element to separate smaller and larger collections on the first pass (possible in $O(1)$ time per element) and then traversing each of those collections (possible in $O(high - low + 1)$ time) to put the elements back into $A[low..high]$ doesn't change the big-Oh time or space requirements.

- *Time and space.*

  Assess the running time and space requirements of the algorithm given the implementation identified.

  For a recursive algorithm, write a recurrence relation for the running time. The challenge here is that we split into two subproblems, but the number of elements handed to each friend depends on the pivot that is chosen. So we consider best and worst cases... Use the table discussed in class to find a big-Oh for each recurrence relation.

  The best case is splitting in half — only $\log n$ steps to get down to a problem of size 1: $T(n) = 2T(n/2) + O(n)$ and thus $T(n) = \Theta(\log n)$

  The worst case is the most uneven split — one friend gets the smallest feasible problem ($n = 0$) and the other friend gets the larger problem ($n = n - 1$). Then: $T(n) = T(0) + T(n-1) + O(n)$ and thus $T(n) = \Theta(n^2)$

- *Room for improvement.*

> Are the targets met? Is it necessary to do better? If improvements in running time and/or space are needed, identify possible avenues for improvement: Can you leverage the work the friends are doing and have them hand back more in order to reduce your computation? Do you need to pass all of the elements off to friends or can some be eliminated?

The best case beats the brute force approach identified, but the worst case does not. If a worst-case runtime of $O(\log n)$ is required, then steps need to be taken to choose a pivot that achieves an even (or close enough to even) split of elements since it is the subproblem size and not the additional work that leads to the $O(n^2)$ running time.

This is where randomized quicksort comes in — choosing a random element as the pivot mitigates the worst case of sorted or reverse sorted order because with every input ordering there is the same likelihood of picking a pivot with a good split vs a bad one. There's still only a $1/n$ chance of picking the exact middle element as the pivot, but the exact middle element isn't needed — for example, what if the pivot results in a 1/4, 3/4 split? In that case $T(n) = T(n/4) + T(3n/4) + O(n)$. This recurrence relation doesn't fit our tables, but observe that there is a total of n elements between the two subproblems and $O(n)$ additional work — so there is a total of n elements and $O(n)$ work between all of the subproblems at a given level. Repeated division of input by some factor leads to $O(\log n)$ levels to reach a base case, resulting in a total of $O(n \log n)$ work if every pivot gives no worse than a 1/4, 3/4 split. What if you get a bad pivot? Pick another one. There's a 50% chance of a pivot resulting in no worse than a 1/4, 3/4 split so on average only two pivots would be needed in order to get a good one. Picking (and partitioning) for two pivots is twice the work of one, but $2n = O(n)$ so we get an expected behavior that is still $O(n \log n)$. This doesn't erase the still-possible worst case of $O(n^2)$ but that requires never picking a good pivot and the odds of that are vanishingly small.