Sorting

Annalise Tarhan

April 23, 2021

1 The Grinch is given the job of partitioning $2n$ players into two teams of n players each. Each player has a numerical rating that measures how good he or she is at the game. The Grinch seeks to divide the players as unfairly as possible, so as to create the biggest possible talent imbalance between the teams. Show how the Grinch can do the job in $O(n \log n)$ time.

Once the players have been sorted by their numerical rating, the problem is trivial. The first n players form one team, the second n players form the second. The sorting can be accomplished in guaranteed $O(n \log n)$ time using heapsort or mergesort.

2 For each of the following problems, give an algorithm that finds the desired numbers within the given amount of time.

2.1 Let S be an unsorted array of n integers. Give an algorithm that finds the pair $x, y \in S$ that maximizes $|x - y|$. Your algorithm must run in $O(n)$ worst case time.

This is equivalent to finding the minimum and maximum elements in the array. Keep track of cur_max and cur_min, initialized to INT_MIN and INT_MAX, respectively, and traverse the array, updating cur_max and cur_min when appropriate.

2.2 Let S be a sorted array of n integers. Give an algorithm that finds the pair $x, y \in S$ that maximizes $|x-y|$. Your algorithm must run in $O(1)$ worst-case time.

$x = S[0]$ $y = S[n-1]$

2.3 Let S be an unsorted array of n integers. Give an algorithm that finds the pair $x, y \in S$ that minimizes $|x-y|$ for $x \neq y$. Your algorithm must run in $O(n \log n)$ worst-case time.

First, use heapsort or mergesort to sort the array. Then, traverse the array, tracking *cur_x*, *cur_y*, and *cur_min*. *cur_x* is initialized to 0, *cur_y* is initialized to 1, and *cur_min* is initialized to $S[1]-S[0]$. Each variable is updated whenever the difference between an adjacent pair of elements is less than *cur_min* but more than 0. After the traversal, return *cur_x* and *cur_y*.

2.4 Let S be a sorted array of n integers. Give an algorithm that finds the pair $x, y \in S$ that minimizes $|x-y|$ for $x \neq y$. Your algorithm must run in $O(n)$ worst-case time.

Do the same as in the previous problem, skipping the sorting step.

3 Take a list of $2n$ real numbers as input. Design an $O(n \log n)$ algorithm that partitions the number into n pairs, with the property that the partition minimizes the maximum sum of a pair.

First, use heapsort or mergesort to sort the numbers into an array S. The partition consists of the pairs $(S[0], S[2n-1])(S[1], S[2n-2])...(S[n-1], S[n])$. Sorting the numbers takes $O(n \log n)$ time and constructing the pairs takes $O(n)$, so the whole algorithm takes $O(n \log n)$.

4 Assume that we are given n pairs of items as input, where the first item is a number and the second item is one of three colors. Further assume that the items are sorted by number. Give an $O(n)$ algorithm to sort the items by color such that the numbers for identical colors stay sorted.

Initialize three arrays of size n, one for each color. Consider each pair in order and insert each one into the first open slot of the appropriate array. Then, choose the array with the most elements to be the base and insert the contents of the other two arrays into the remaining slots of the base array, preserving their ordering. This algorithm accomplishes the sorting with $3n$ extra memory in less than two passes. An alternative that only uses n extra memory but does three passes is as follows: Instantiate a single array of size n. Choose one of the colors and scan the original array for pairs with that color and insert them in order into the new array. Repeat for the other two colors.

5 The mode of a bag of numbers is the number that occurs most frequently in the set. Give an efficient and correct algorithm to compute the mode of a bag of n numbers.

Insert each number into a hashtable that stores key-value pairs consisting of the number and the number of occurrences so far. Use a pair of variables to track the current most frequent number and its count, initialized to the first number and one. Update when a different number's count exceeds the current one. When every number has been inserted, the current most frequent number will be the mode.

6 Given two sets S_1 and S_2, each of size n, and a number x, describe an $O(n \log n)$ algorithm for finding whether there exists a pair of elements, one from S_1 and one from S_2, that add up to x.

First, use heapsort or mergesort to sort each array. Then, traverse the sorted S_1 to find the first element, $S_1[m]$ such that $S_1[m] + S_2[0] > x$, or until $S_1[n-1]$ is reached. From here, the S_1 pointer will move backwards and the S_2 pointer will move forwards. Now, compare $S_1[m] + S_2[0]$ to x. (If $S_1[n-1]$ was reached, the sum might be less than x.) If the sum is greater than x, move the S_1 pointer back one element. If it is smaller, move the S_2 pointer forwards. Continue moving the pointers in this way until a pair of elements whose sum equals x is found. If the S_1 pointer reaches $S_1[0]$ or the S_2 pointer reaches $S_2[n-1]$ and an appropriate pair is not found, it does not exist.

7 Give an efficient algorithm to take the array of citation counts of a researcher's papers, and compute the researcher's h-index. By definition, a scientist has index h if h of his or her n papers have been cited at least h times, while the other $n-h$ papers each have no more than h citations.

First, sort the array so that $S[0]$ contains the highest number of citations. Traverse the array, looking for the first element such that $S[i] < i + 1$. Then the researcher's h-index equals i. If the end of the array is reached before that condition is met, the researcher's h-index equals n. If the array is empty, the h-index is zero.

8 Outline a reasonable method of solving the following problems. Give the order of the worst-case complexity of your methods.

8.1 You are given a pile of thousands of telephone bills and thousands of checks sent in to pay the bills. Find out who did not pay.

Separate the bills from the checks and sort each pile alphabetically by the customer's name, then by date if there is more than one bill per customer. This will take $O(n \log n)$ time if using mergesort, which is feasible to do by hand. Then, compare the first bill with the first check. If the names and dates (or amounts) match, move them aside. Continue until there is a bill without a matching check, then put unpaid bill in a different pile and continue. This takes $O(n)$ time, for a total of $O(n \log n)$.

8.2 You are given a printed list containing the title, author, call number, and publisher of all the books in a school library and another list of thirty publishers. Find out how many of the books in the library were published by each company.

First, rewrite the publisher list so it is sorted alphabetically. Then, go through the book list, identifying the publisher of each book, finding the publisher in the alphabetized list using a binary search, and putting a tally mark next to the publisher's name. If there are n books in the library, this will take $O(\log_2(30) * n) \sim O(5n) = O(n)$ time.

8.3 You are given all the book checkout cards used in the campus library during the past year, each of which contains the name of the person who took out the book. Determine how many distinct people checked out at least one book.

Depending on the size of the campus, the first step here would probably be to get out a computer and initialize a hashtable. If using a computer is cheating, opt for sorting instead. Sort alphabetically by the person's name and discard duplicates. At the end, count the number of cards remaining.

9 Given a set S of n integers and an integer T, give an $O(n^{k-1} log\ n)$ algorithm to test whether k of the integers in S add up to T.

First, sort the set and discard all elements that are definitely too large. That isn't just elements that are larger than T, but also those that are larger than T minus the sum of the smallest $k-1$ elements. Then, starting with the first $k-1$ elements, calculate their sum and subtract it from T. Perform a binary search on the remaining integers for the calculated difference. Assuming it wasn't found, continue with the first $k-2$ elements and the $k-2$nd element, calculating the difference between their sum and T and performing a binary search on the largest $k-3$ elements. Choosing each combination of $k-1$ elements is $O(n^{k-1})$ and the binary search for an appropriate kth element is $O(\log n)$ after the set has been sorted, for a total time complexity of $O(n^{k-1} \log n)$.

10 We are given a set of S containing n real numbers and a real number x, and seek efficient algorithms to determine whether two elements of S exist whose sum is exactly x.

10.1 Assume that S is unsorted. Give an $O(n \log n)$ algorithm for the problem.

Initialize a self-balancing binary search tree and insert each element in the set. At each insertion, also calculate the difference between the inserted element and x, and search the tree for that element. If one is found, return both elements. If every element is inserted without a pair being found, none exists.

10.2 Assume that S is sorted. Give an $O(n)$ algorithm for the problem.

Start with a pointer to the first element of the array and one to the last element of the array. If the sum of those two elements is less than x, move the first pointer forward. If it is greater than x, move the second pointer backward. Repeat the process until a suitable pair of elements is found or until the two pointers meet. As an optimization, make sure x is greater than or equal to the sum of the two smallest elements and is less than or equal to the sum of the largest two before traversing the entire array.

11 Design an $O(n)$ algorithm that, given a list of n elements, finds all elements that appear more than $n/2$ times in the list. Then, design an $O(n)$ algorithm that, given a list of n elements, finds all the elements that appear more than $n/4$ times.

Use a hash table as a supplementary data structure. Insert each element into the hash table, with the element serving as the key and the number of times it has been inserted as the value. Keep track of the maximum insertion count and the corresponding element. (Only the maximum needs to be tracked, because only at most one element can occur more than $n/2$ times in a set of size n.) After every element has been inserted, compare that maximum count size to $n/2$ and return the corresponding element if appropriate.

If n is known in advance, repeat the previous algorithm, adding an element to a return list when its count exceeds $n/4$. Otherwise, insert each element as before, calculate $n/4$ once n is known, and iterate through the list again, adding any element whose count exceeds $n/4$ to the return list.

12 Give an efficient algorithm to compute the union of sets A and B, where $n = max(|A|, |B|)$. The output should be an array of distinct elements that form the union of the sets.

12.1 Assume that A and B are unsorted arrays. Give an $O(n \log n)$ algorithm for the problem.

Choose A or B and use it to populate a self-balancing binary search tree. Do not allow duplicates. Then, iterate through the other set, searching the tree for each element. If the element is found, continue. Otherwise, insert the new element. At the end, traverse the tree and insert each element into the return array.

12.2 Assume that A and B are sorted arrays. Give an $O(n)$ algorithm for the problem.

Start with a pointer to the first element of A and one to the first element of B. If they are equal, add the element to a return array and increment each pointer. Otherwise, add the smaller value to the return array and increment that pointer. Repeat until both pointers have reached the ends of their arrays. Every time a pointer is incremented, temporarily store the previous value, and increment until the pointer points to a different value, to avoid inserting duplicates.

13 A camera at the door tracks the entry time a_i and exit time b_i for each of n persons p_i attending a party. Give an $O(n \log n)$ algorithm that analyzes this data to determine the time when the most people were simultaneously present at the party. You may assume that all entry and exit times are distinct.

Use whatever strategy is appropriate based on the input format to concatenate all the times into one list and sort it, tagging each time as an entry time or an exit time. Then, iterate through the list, incrementing a people counter for each entry and decrementing it for each exit. Maintain a reference to the current max along with the time it occurred, and update when a new maximum occurs. At the end, return the time associated with the current max.

14 Given a list I of n intervals, specified as (x_i, y_i) pairs, return a list where the overlapping intervals are merged. For $I = \{(1,3), (2,6), (8,10), (7,18)\}$, the output should be $\{(1,6), (7,18)\}$. Your algorithm should run in worst-case $O(nn)$ time complexity.

Instantiate two integer variables x and y and a return list L. Set x equal to x_0 and y equal to y_0. Then, iterate through the list: If $x_i \leq y$, set $y = y_i$. Else, insert (x, y) into L and set $x = x_i$ and $y = y_i$. At the end of the list, insert (x, y) and return L.

15 You are given a set S of n intervals on a line, with the ith interval described by its left and right endpoints (l_i, r_i). Give an $O(n \log n)$ algorithm to identify a point p on the line that is in the largest number of intervals. You can assume an endpoint counts as being in its interval.

Note: I've been switching between two approaches to 'merging' lists with different classes of items. In the first, the lists are actually merged. First, each item is converted into a pair of the item and a tag representing the list it came from. Then the lists are merged and sorted. I'll call this the merge with tags approach. In the second, which I'll call the double pointer method, the lists are sorted separately, and then iterated through using two pointers, each initially pointing to the head of each list and choosing the smaller of the two to visit first. The advantage of merge with tags is that the sorting work is done all at once and the results are preserved, so if the data will ever be reused, the correct ordering will be ready to use. The downside is the extra memory used for the new sorted list, including space for all the tags. The advantage of the double pointer method is simplicity, speed, and a smaller memory footprint, at the cost of not having the sorting preserved. For the purposes of many of these questions, they are equivalent, as they are both $O(n)$.

Returning to this question, use the merge with tags approach to insert all endpoints into the list and sort them, with the tag representing whether it is a left or right endpoint. Initialize a current max variable, initialized to zero, with a corresponding variable to hold an endpoint reference, *time*, as well as a count variable initialized to zero. Iterate through the sorted list, incrementing the count variable for left endpoints and decrementing it for right endpoints. When *count* exceeds *cur_max*, set *cur_max* to the current count and set *time* to the most recently visited endpoint. At the end, return *time*.

16 You are given a set S of n segments on a line, where segment S_i ranges from l_i to r_i. Give an efficient algorithm to select the fewest number of segments whose union completely covers the interval from 0 to m.

Sort the segments by l_i. Set $start = S_0$, $end = S_0$, and iterate through the rest of the list. When the first segment whose l_i is greater than *start*'s r_i is encountered, stop. Add *start* to the return set and set $start = end$. When a segment is encountered whose r_i is greater than *end*'s, make *end* point to the new node instead. Repeat until *end*'s $r_i == m$. At that point, add both start and end to the return set and return.

17 Devise an algorithm for finding the k smallest elements of an unsorted set of n integers in $O(n + k \log n)$ time.

Construct a min heap in $O(n)$ time and then extract the minimum element k times in $O(k \log n)$ time.

18 Give an $O(n \log k)$-time algorithm that merges k sorted lists with a total of n elements into one sorted list.

At any time, there are only k possibilities for the next smallest element. At first, the k possibilities are the heads of each of the k lists. Insert each into a min heap, tagged with a reference to which of the lists it came from. Then, remove the element at the top of the heap and replace it with the next element from its original list, which will be bubbled down to an appropriate place. Insert the removed element in the new sorted list and repeat until all elements have been sorted.

19 You wish to store a set of n numbers in either a max-heap or a sorted array. For each application below, state which data structure is better, or if it does not matter. Explain your answers.

19.1 Find the maximum element quickly.

It makes no difference, since the location of the maximum element is known. For a max heap it is the first element, and for the sorted array it is either the first or last, depending on the sort order.

19.2 Delete an element quickly.

The max-heap is better, since restoring the heap takes $O(\log n)$ while shifting all the elements in an array takes $O(n)$.

19.3 Form the structure quickly.

The heap requirements are less strict, so it can be constructed in $O(n)$ while the sorted array takes $O(n \log n)$. Use a heap.

19.4 Find the minimum element quickly.

The sorted array is the clear winner, since it can find the minimum or the maximum in $O(1)$. A max-heap, though, makes no guarantees about the location of the minimum element, except that it must be a leaf. Since there are approximately $n/2$ leaves, it takes $O(n)$ to find the minimum.

20

20.1 Give an efficient algorithm to find the second-largest key among n keys. You can do better than $2n - 3$ comparisons.

First, do $n/2$ comparisons, comparing each pair of elements and storing the larger of the pair in an array. Repeat, comparing each pair of elements and storing the results, until the largest element is found. At some point, the second largest element will have been compared to the largest element and lost, so use the stored arrays to find each of the $\log n$ elements the largest was compared to and find the maximum of those. The total number of comparisons is approximately $n + \log n$.

Credit and thanks to https://stackoverflow.com/questions/3628718/find-the-2nd-largest-element-in-an-array-with-minimum-number-of-comparisons

20.2 Then, give an efficient algorithm to find the third-largest key among n keys. How many key comparisons does your algorithm do in the worst case? Must your algorithm determine which key is largest and second-largest in the process?

Use the algorithm in the previous section to find the largest element. The third largest element would have lost a comparison to either the largest or second largest, so traverse their histories to find it in the same way as before. This takes at most $n + 2\log n$ comparisons, depending on how far down the tree the second largest element occurred. (If the second largest element was compared to the largest element in the first round, the third largest element would already have been found and the number of comparisons would be closer to $n + \log n$.)

21 Use the partitioning idea of quicksort to give an algorithm that finds the median element of an array of n integers in expected $O(n)$ time.

Begin the same was as quicksort, by picking the first element and partitioning around it. Instead of repeating on both sides, only repeat the partition on the subarray with more elements. (If they are equal, congratulations! You've already found the median element.) It will always be possible to determine which subarray contains the median element based on the size of the original array and the size of the resulting subarrays. Repeat the partitioning until the median element is found. The time complexity is linear because the first partition does n comparisons, the second does (on average) $n/2$, and so on, for an approximate total of $2n$ comparisons.

22 The median of a set of n values is the $\lceil n/2 \rceil$th smallest value.

22.1 Suppose quicksort always pivoted on the median of the current sub-array. How many comparisons would quicksort make then in the worst case?

$n \log_2 n$

22.2 Suppose quicksort always pivoted on the $\lceil n/3 \rceil$rd smallest value of the current sub-array. How many comparisons would be made then in the worst case?

$\frac{2n}{3}\log_{3/2} n + \frac{n}{3}\log_3 n$ (...maybe?)

23 Suppose an array A consists of n elements, each of which is red, white, or blue. We seek to sort the elements so that all the reds come before all the whites, which come before all the blues. The only operations permitted on the keys are $Examine(A, i)$, which reports the color of the ith element of A, and $Swap(A, i, j)$, which swaps the ith element of A with the jth element. Find a correct and efficient algorithm for red-white-blue sorting. There is a linear time solution.

This algorithm uses three pointers and accomplishes the sorting in a single pass. The first pointer is used to mark the end of the sorted red section, the second is used to mark the beginning of the sorted blue section, and the third is used to iterate through the array. The red pointer starts at the first non-red element in the array, discovered by calling *Examine* repeatedly, starting with $Examine(A, 0)$. The blue pointer starts at the last non-blue element, discovered by a similar process, starting with calling $Examine(A, n-1)$. The third pointer starts at the first element after the red pointer. As the iterating pointer reaches each element, it uses *Examine* to determine the element's color, then swaps it if it isn't white. If it is red, it calls *Swap* with the current element and the red pointer, which is then incremented until it finds a non-red element. (If the red pointer catches up to the iterating pointer, the iterating pointer should also be incremented until it points to the same (non-red) element as the red pointer.) If it is blue, it calls *Swap* with the element and the blue pointer, which is then decremented until it finds a non-blue element. Importantly, the swapped in element is also examined and if it is red or blue, it is swapped again. After the iterating pointer reaches the blue pointer, the array is sorted.

24 Give an efficient algorithm to rearrange an array of n keys so that all the negative keys precede all the non-negative keys. Your algorithm must be in-place, meaning you cannot allocate another array to temporarily hold the items. How fast is your algorithm?

I would use a similar strategy as in the last problem, but this time with two pointers. The first starts at the first non-negative element in the array and the second starts at the element immediately following that one. The second iterates through the array until it reaches a negative element, which it then swaps with the element pointed to by the first pointer. Both pointers are then incremented, the first until it reaches a non-negative element and the second until it reaches a negative one. At that point, the elements are swapped and the process repeats. When the second pointer reaches the end of the array, it is sorted, which will take $O(n)$ time.

25 Consider a given pair of different elements in an input array to be sorted, say z_i and z_j. What is the most number of times z_i and z_j might be compared with each other during an execution of quicksort?

Once. Elements are only compared to the partition element, and once an element has been used as a partition, it is never examined again.

26 Define the recursion depth of quicksort as the maximum number of successive recursive calls it makes before hitting the base case. What are the minimum and maximum possible recursion depths for randomized quicksort?

Even a randomized quicksort can be randomly unlucky, so the maximum recursion depth is $n - 1$. The minimum recursion depth is $\lceil \log_2 n \rceil - 1$.

27 Suppose you are given a permutation p of the integers 1 to n, and seek to sort them to be in increasing order $[1, ..., n]$. The only operation at your disposal is $reverse(p, i, j)$, which reverses the elements of a subsequence in the permutation.

27.1 Show that it is possible to sort any permutation using $O(n)$ reversals.

The first step is to locate the smallest element, 1, in the array, say at index j. Then, call $reverse(p, 0, j)$, which deposits 1 in the first slot. Then, find the second element, 2, at index k. Call $reverse(p, 1, k)$, which puts 2 in the second slot. Repeat $n - 1$ times until each element has been deposited in its place.

27.2 Now suppose the cost of $reverse(p, i, j)$ is equal to its length, the number of elements in the range $|j - i| + 1$. Design an algorithm that sorts p in $O(n \log^2 n)$ cost. Analyze the running time and cost of your algorithm and prove correctness.

The best algorithm I can come up with is the one above, with the possible optimization that each reversal can deposit an element at either end of the range, depending on which is closer to its intended place. So, for a permutation of elements from 1 to 10, the first reversal would place either 1 or 10. This would hopefully have the effect of moving the other element closer to its side as well. It doesn't even come close to solving the problem, though, because this is an $O(n^2)$ algorithm, as far as I can tell. I would think that because this is in the quicksort section and because it involves reversing around a point that the solution should involve some sort of partition, but I can't make the connection.

28 Consider the following modification to merge sort: divide the input array into thirds (rather than halves), recursively sort each third, and finally combine the results using a three-way merge subroutine. What is the worst case running time of this modified merge sort?

The recursion tree for this version of mergesort would have height $\lceil \log_3 n \rceil$ and $2n$ comparisons would be done to merge the tree. Each differs only by a constant factor from the original, so the complexity is still $O(n \log n)$.

29 Suppose you are given k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements. One approach is to use the merge subroutine repeatedly, merging the first two arrays, then merging the result with the third array, then with the fourth array, and so on until you merge in the kth and final input array. What is the running time?

Merging the first two arrays costs $2n$ comparisons, merging that array with the third costs another $3n$, and so on until merging the final $(k-1)n$ sized array with the last n sized array costs kn comparisons, for a total running time in $O(k^2 n)$.

30 Consider again the problem of merging k sorted length-n arrays into a single sorted length-kn array. Consider the algorithm that first divides the k arrays into $k/2$ pairs of arrays, and uses the merge subroutine to combine each pair, resulting in $k/2$ sorted length-$2n$ arrays. The algorithm repeats this step until there is only one length-kn sorted array. What is the running time as a function of n and k?

The first round does $k/2$ merges costing $2n$ comparisons each, for a total of kn comparisons. The second round does $k/4$ merges costing $4n$ comparisons, for a total, again, of kn. There are $\log k$ rounds, for a total complexity of $O(kn \log k)$.

31 Stable sorting algorithms leave equal-key items in the same relative order as the original permutation. Explain what must be done to ensure that mergesort is a stable sorting algorithm.

A secondary key must be added, equal to each element's original position in the array. Comparisons take this key into account to break ties, resulting in a stable sort.

32 Wiggle sort: Given an unsorted array A, reorder it such that $A[0] < A[1] > A[2] < A[3]....$ For example, one possible answer for input $[3, 1, 4, 2, 6, 5]$ is $[1, 3, 2, 5, 4, 6]$. Can you do it in $O(n)$ time using only $O(1)$ space?

Wiggle sort was assigned earlier as a leetcode exercise, which included the possibility of duplicate elements. My approach was to traverse the array repeatedly if necessary, swapping neighboring elements if an element at an even index was larger than its neighbor at an odd index. The difficulty was when neighboring elements were equal, which I handled by swapping one with a more distant element. I don't know how to prove that the solution I gave wouldn't become $O(n^2)$ or worse with lots of duplicate elements, but in practice it was very fast. Most other solutions worked by finding the median element first and sorting so that each even indexed element was less than the median and each odd indexed element was greater than the median, but I preferred a more organic, wiggly approach, which had the unfortunate quality of having a time complexity that was hard to pin down.

33 Show that n positive integers in the range 1 to k can be sorted in $O(n \log k)$ time. The interesting case is when $k \ll n$.

Insert each element into a self-balancing binary search tree, indexed by the value of the integers, where duplicates are handled by incrementing a count stored by each node. After all elements have been inserted in $O(n \log k)$ time, do an in-order traversal of the tree, inserting each item into the sorted list a number of times equal to its count. Traversing the tree takes time proportionate to the number of unique elements in n and inserting each takes $O(n)$ time, for a total time complexity equal to $O(n \log k)$.

34 Consider a sequence S of n integers with many duplications, such that the number of distinct integers in S is $O(\log n)$. Give an $O(n \log \log n)$ worst-case time algorithm to sort such sequences.

Insert each element into a self-balancing binary search tree that tracks duplicate counts at each of the nodes. There will be a total of $\log n$ elements in the tree, so inserting all of them will cost $O(n \log \log n)$ time. Afterwards, do an in-order traversal of the tree and insert each element into the sorted list a number of times equal to the node's duplicate count, which can be done in $O(n)$ time. The total time complexity is $O(n \log \log n)$.

35 Let $A[1..n]$ be an array such that the first $n - \sqrt{n}$ elements are already sorted. Give an algorithm that sorts A in substantially better than $n \log n$ steps.

First, use any reasonable sorting algorithm to sort the last $\sqrt{n}$ elements in $\sqrt{n} \log \sqrt{n}$ time. Then, use mergesort to merge the first $n - \sqrt{n}$ elements with the last $\sqrt{n}$ elements in $O(n - \sqrt{n} + \sqrt{n}) = O(n)$ time. The total time complexity is then $O(n + \sqrt{n} \log \sqrt{n}) = O(n)$.

36 Assume that the array $A[1..n]$ only has numbers from $\{1, ..., n^2\}$ but that at most $\log \log n$ of those numbers ever appear. Devise an algorithm that sorts A in substantially less than $O(n \log n)$.

This array is guaranteed to have many, many duplicates, so use a binary search tree. Make sure it is self-balancing to keep insertion times low, and that each node tracks how many duplicates have been inserted. Inserting n elements where only $\log \log n$ of them are unique into such a tree costs $O(n \log \log \log n)$ time, and traversing the tree and inserting each into the array in sorted order will cost $O(n)$ time, for a total of $O(n \log \log \log n)$, which is substantially less than $O(n \log n)$.

37 Consider the problem of sorting a sequence of n 0's and 1's using comparisons. For each comparison of two values x and y, the algorithm learns which of $x < y$, $x = y$, or $x > y$ holds.

37.1 Give an algorithm to sort in $n - 1$ comparisons in the worst case. Show that your algorithm is optimal.

As a first step, do $n/2$ comparisons, comparing each element to one of its neighbors. If one is greater than the other, put the small element in the first open space in the return array and put the larger one in the last open space. If they are equal, ignore them for now, but track which element it was compared to. Then, repeat the procedure for the groups of two, comparing one element of one pair to an element from a different pair. If one is greater than the other, put the greater element and its pair in the last two open spaces in the return array and put the smaller element and its pair in the first two. If they are equal, set all four aside until the next round. Continue until it is determined where each element goes, either because it was compared to something not equal to itself or because all elements are equal, in which case order doesn't matter. (If there is an odd number of groups remaining, compare an element from that group to a known element from the return array.)

In the best case, there will be $\lfloor n/2 \rfloor$ comparisons when each element is unequal to the first element it is compared to. This is as good as it gets, because all but one element must be compared at least once. (The value of element that ends up in the very middle of the return array doesn't need to be determined.) In the worst case, there will be $n - 1$ comparisons, because all comparisons (except maybe the last one) were between equal elements or groups of elements. The solution is also optimal in this case, because it is equivalent to a tournament tree. Visually, each element is a leaf node in a binary tree and the node above each pair of leaves represents the comparison. If the comparison is anything but equal, those leaves' mini-tree ends there. Otherwise, the parent node will be paired with a different parent node and compared, creating a new parent node representing twice as many elements as each of its children. In any binary tree with n leaf nodes, there are at most $n-1$ internal nodes, in this case representing comparisons.

37.2 Give an algorithm to sort in $2n/3$ comparisons in the average case (assuming each of the n inputs is 0 or 1 with equal probability). Show that your algorithm is optimal.

This is the same algorithm as before. In the first round, there are $n/2$ comparisons. Half return equal, half don't. Out of eight elements, four will be in pairs that return unequal and four will need further examination. Of those four, there will be one comparison in the second round, for a total of $n/8$ comparisons in the second round. Half of those will return unequal and be eliminated. In the third round, there will be one comparison per 32 original elements. That pattern continues until only one group remains, and that can be compared to an already decided element from the return array. $n/2+n/8+n/32+... = 2n/3$

38 Let P be a simple, but not necessarily convex, n-sided polygon and q an arbitrary point not necessarily in P. Design an efficient algorithm to find a line segment originating from q that intersects the maximum number of edges of P. In other words, if standing at point q, in what direction should you aim a gun so the bullet will go through the largest number of walls? A bullet through a vertex of P gets credit for only one wall. An $O(n \log n)$ algorithm is possible.

Each of the n edges has two endpoints, each of which forms a vertex with another edge. Calculate the angle from q to each of those endpoints and record the range for each edge, choosing an arbitrary angle to represent $0°$. The goal is to find which angle occurs the most ranges, which can be accomplished by marking each angle as opening or closing a range, then sorting them. In general, the smaller angle in a range should be listed as the opener and the larger angle as the closer. The exception is ranges that include $0°$, for example $(350°, 10°)$. Now, mark each angle as an opener or a closer and sort them. Traverse the list of angles, incrementing a counter at each opener and decrementing it at each closer. Keep track of the location of the max so far, and be sure to properly handle segments such as $(350°, 10°)$. When the max has been located, choose an arbitrary angle in that range, excluding the endpoints. A potential wrinkle is a segment whose opening and closing angles are exactly the same, from q's perspective. That would be the exception to the rule of not counting endpoints, because the ‘endpoint’ would actually be the entire edge. Either handle those manually or fudge the endpoints slightly.

39 In one of my research papers, I discovered a comparison-based sorting algorithm that runs in $O(n \log \sqrt{n})$ time. Given the existence of an $\Omega(n \log(n))$ lower bound for sorting, how can this be possible?

$O(n \log \sqrt{n})$ only differs from $O(n \log(n))$ by a constant factor.
$n \log(\sqrt{n}) = n \log(n^{1/2}) = \frac{1}{2} n \log(n)$
For $c > 2$, $\frac{1}{2} cn \log(n) > n \log(n)$, so $n \log(\sqrt{n}) = \Omega(n \log(n))$.

40 Mr. B. C. Dull claims to have developed a new data structure for priority queues that supports the operations $Insert$, $Maximum$, and $Extract - Max$, all in $O(1)$ worst-case time. Prove that he is mistaken.

This data structure would only function properly if it was guaranteed that each item inserted would be the new maximum. Otherwise, a newly inserted element would need to be sorted into place either immediately after insertion or lazily when $Maximum$ or $Extract - Max$ was called. The data structure could get away with constant time if there were only ever two elements, but if n items were inserted and then $Extract - Max$ was called n times, the items would need to emerge in sorted order. There is a well established lower bound on sorting, and $O(n \log n) \gg O(1)$.

41 A company database consists of 10,000 sorted names, 40% of whom are known as good customers and who together account for 60% of the accesses to the database. There are two data structure options to consider for representing the database:

• Put all the names in a single array and use binary search
• Put the good customers in one array and the rest of them in a second array. Only if we do not find the query name on a binary search of the first array do we do a binary search of the second array.

Demonstrate which option gives better expected performance. Does this change if linear search on an unsorted array is used instead of binary search for both options?

For a single array, every query costs something like $\log_2(n) = log_2(10000) = 13.29$ operations. For two arrays, every query costs $\log_2(.4n)$ and 40% of queries cost an additional $\log_2(.6n)$. On average, each query will then cost:
$\log_2(.4n) + .4(\log_2(.6n))$
$\log_2(4000) + .4(\log_2(6000))$
$11.96 + 5.02 = 16.98$
Clearly, a binary search is efficient enough that separating out the most frequently accessed customer data is counterproductive. On the other hand, the expected number of operations for finding one customer out of 10,000 in an unsorted array is 5,000, and splitting the arrays reduces that value to $4000/2 + .4(6000/2) = 3,200$, a significant improvement.

42 A Ramanujan number can be written two different ways as the sum of two cubes, meaning there exist distinct positive integers a, b, c, and d such that $a^3 + b^3 = c^3 + d^3$. For example, 1729 is a Ramanujan number because $1729 = 1^3 + 12^3 = 9^3 + 10^3$.

42.1 Give an efficient algorithm to test whether a given single integer n is a Ramanujan number, with an analysis of the algorithm's complexity.

Start with $i = 1$. Subtract i^3 from n and take the cube root of the difference. If the result is an integer, save that integer as *prev*, set a flag, and keep going. Increment i and take the cube root of $n - i^3$. (Skip $i = prev$.) If the result is an integer and the flag is set, n is a Ramanujan number. Otherwise, save the result in *prev*, set the flag, and continue. Repeat until $n - i^3 > i^3$ or it is determined that n is a Ramanujan number. The set of calculations will be performed for $i \in \{1, 2, ..., \left\lceil \sqrt[3]{n/2} \right\rceil\}$, so the algorithm is $O(\sqrt[3]{n})$, assuming the cube root function is constant time.

42.2 Now give an efficient algorithm to generate all the Ramanujan numbers between 1 and n, with an analysis of its complexity.

Start with $i = 1$. While $i^3 < n/2$, let $j = i$ and calculate $i^3 + j^3$. Store the result in a hash table and increment j. Repeat until $i^3 + j^3 > n$, then increment i and repeat. If a duplicate is inserted into the hash table, that number is a Ramanujan number. This time, the set of calculations is performed $(\sqrt[3]{n})^2$ times, so the algorithm is $O(n^{2/3})$.

43 Consider an $n \times n$ array A containing integer elements. Assume that the elements in each row of A are in strictly increasing order and the elements of each column of A are in strictly decreasing order. Describe an efficient algorithm that counts the number of occurrences of the element 0 in A. Analyze its running time.

Start in the $n/2$nd row and perform a binary search, first checking the element at $A[n/2]$. If $A[n/2] > 0$, check $A[n/4]$. If it is less than zero, check $A[3n/4]$. Repeat until a zero is found or determined not to exist. At each step, part of A is eliminated. If $A[i, j] > 0$, then every $A[i', j']$ such that $i' > i$ and $j' > j$ must contain a positive integer. Conversely, if $A[i, j] < 0$, then every $A[i', j']$ such that $i' < i$ and $j' < j$ will contain a negative integer. After the middle row has been evaluated, recursively check the upper left and bottom right 'quadrants,' or the sections that weren't eliminated. In other words, whether or not a zero was found, the boundary between positive and negative integers in the $n/2$nd row was determined. All elements in the rectangle formed by the last negative integer and the element in the bottom left corner of A have been eliminated, and so have those in the rectangle formed by the first positive integer in the $n/2$nd row and the top right corner of A.

The first row costs $O(\log n)$ to examine in a binary search. The second two rows examined cost $O(\log n/2)$ each. Then, four rows of (on average) $n/4$ elements will be searched, and so on. As far as I can figure, the complexity is represented by $\sum_{i=0}^{\log_2 n} 2^i \log_2 \frac{n}{2^i}$, which simplifies to $\log_2 n \sum_{i=0}^{\log_2 n} 2^i - \sum_{i=0}^{\log_2 n} i2^i$. Applying the formulas for the sums of arithmetic series and arithmetico–geometric series, we get a bunch of junk that still works out to $O(n \log n)$. As frustrating as it is to do all that work to optimize the search and have the complexity be no better than the naive approach, the lesson is that when the complexity for a certain element is only $\log n$, cutting n in half, even repeatedly, doesn't change much.

44 Implement versions of several different sorting algorithms, such as selection sort, insertion sort, heapsort, mergesort, and quicksort. Conduct experiments to assess the relative performance

Skipped

45 Implement an external sort, which uses intermediate files to sort files bigger than main memory.

Skipped (but use mergesort)

46 Design and implement a parallel sorting algorithm that distributes data across several processors.

Skipped

47 If you are given a million integers to sort, what algorithm would you use to sort them? How much time and memory would that consume?

If the range of the integers was relatively small, I would use a counting sort. Scan the list for the minimum and maximum, and instantiate an array with the same range. Sorting the integers would just be a matter of incrementing the count (initialized to zero) at the corresponding array element, calculated by subtracting the value of the minimum element from each element inserted, so that the minimum element mapped to zero and the maximum element mapped to the last index in the array. Once all integers are inserted, iterate over the array and insert each element in sorted order into the original array a number of times equal to the count at that element's index. Time complexity would be $O(n)$, memory use would be $O(N)$, where N is the difference between the largest and smallest element.

For an input set with a larger range, go for radix sort instead, which sorts input into buckets one digit at a time, then reconstructs them in order at the end, also in $O(n)$ time but in $O(n)$ space.

48 Describe advantages and disadvantages of the most popular sorting algorithms.

Quicksort - Generally the fastest comparison based sorting algorithm, except that it risks $O(n^2)$ time if choosing pivots from the beginning of the input and given sorted input or when choosing pivots more randomly and just astonishingly unlucky. Not stable.

Mergesort - Guaranteed $O(n \log n)$ time and faster than heapsort. Requires additional memory, but is good for large inputs where the sort must be done externally. Stable.

Heapsort - Also $O(n \log n)$, but in-place. Not stable.

Insertion sort - The most efficient of the major $O(n^2)$ algorithms, and a good choice for smaller data sets. Does well on almost-sorted input. In-place, stable.

Selection sort - Simple to implement, but that's about it. In-place, unstable.

Bubble sort - Don't do it.

Counting sort - The big advantage is that it is $O(n)$, but it has a large memory footprint except for input with a tiny range. Works best when there are many repetitions in the input data and they keys must be relatively small integers, or only over a small range of integers. Often used as a subroutine for radix sort.

Radix sort - Also linear time, but only appropriate for data like strings and numbers that can be broken down into individually ordered pieces. Uses $O(n)$ extra memory.

49 Implement an algorithm that takes an input array and returns only the unique elements in it.

This implementation uses the strategy outlined two problems ago with integer input, so it has the side effect of sorting the numbers as well. An approach that generalizes to non-integer input would use any sorting algorithm, then either delete or refrain from copying to a return array any element that is equal to its predecessor.

```
int *unique_ints(int ints[], int num_elements) {
        int max = INT_MIN, min = INT_MAX, i=0, j=0;

        /* Calculate range of integers */
        for (i = 0; i < num_elements; i++) {
                if (ints[i] < min) {
                        min=ints[i];
                }
                if (ints[i] > max) {
                        max=ints[i];
                }
        }
        int range = max-min;

        /* Save one copy of each integer */
        int counts[range];
        int num_uniques = 0;
        for (i = 0; i < num_elements; i++) {
                int offset = ints[i] - min;
                if (counts[offset] == 0) {
                        num_uniques++;
                        counts[offset] = ints[i];
                }
        }

        /* Copy each unique value to the return array */
        int *uniques = malloc(sizeof(int) * num_uniques);
        for (i = 0; i < range; i++) {
                if (counts[i] != 0) {
                        uniques[j] = counts[i];
                        j++;
                }
        }
        return uniques;
}
```

50 You have a computer with only 4 GB of main memory. How do you use it to sort a large file of 500 GB that is on disk?

Use quicksort to sort in place four gigabytes of the file at a time and save each of the 125 sorted chunks to separate files. From there, use mergesort. Only two gigabytes can be sorted at a time now, since mergesort needs $O(n)$ memory as a workspace. Draw one gigabyte from each of the first two files, replenishing each file's allotted gigabyte when the previous one has been fully merged, and save the resulting eight gigabyte file. Once all of the four gigabyte files have been merged (find a generous pair to take in the lonely 125th file), repeat the process for the eight gigabyte files, then the 16 GB files, and so on until all have been merged into one sorted 500 GB file. Make sure to delete all the intermediate files after they have been merged.

51 Design a stack that supports push, pop, and retrieving the minimum element in constant time.

This is a standard stack with a pointer to the top of the stack that also has a pointer to the minimum element. Additionally, each node has space for an extra pointer, which is null for most nodes. When an element is pushed onto the stack, its value is compared to that of the current minimum element. If its value is smaller, it uses its extra pointer to point to the previous minimum element and the stack's minimum element pointer is updated to point to the new element instead. When a minimum element is popped, the new minimum will be the element pointed to by the popped element's extra pointer. Update the stack's minimum element pointer accordingly.

52 Given a search string of three words, find the smallest snippet of the document that contains all three of the search words–that is, the snippet with the smallest number of words in it. You are given the index positions where these words occur in the document, such as $word1 : (1, 4, 5)$, $word2 : (3, 9, 10)$, and $word3(2, 6, 15)$. Each of the lists are in sorted order, as above.

First, use mergesort to combine the three lists, keeping each element tagged as 1, 2, or 3 for its search word. Then, initialize three ints to -1, one for each of the three search words, one to INT_MAX, which will track the smallest range so far, and two more, *start* and *end*, for the locations of the words that define that range. (The location of the middle word isn't important.) Pop elements from the merged list one at a time, moving them to the holder variables corresponding to their search word. Keep popping and storing elements in their holders (discarding previously stored elements) until all three have non-negative values. At that point, determine the smallest and largest of the three values, store them in *start* and *end* respectively, and store their difference in *min_distance*. As each new element is popped, calculate the new range and store the distance and the current elements if the new distance beats the old distance. After all elements have been popped, the values stored in *start* and *end* represent the location of the snippet we're looking for.

53 You are given twelve coins. One of them is heavier or lighter than the rest. Identify this coin in just three weighings with a balance scale.

Twelve coins, three weighings remaining: Put four coins on one side and four on the other. If the weights are the same, the coin is one of the four not on the scale. Jump to 'Four coins.' If they are unequal, the coin could be any of the eight.

Eight coins, two weighings remaining: Label the coins in the heavy group ABCD and the coins in the lighter group EFGH. We'll also be using one of the coins from the third group, call it J, that we know to be safe. On one side of the scale, put ABE. On the other, put CFJ. Leave DGH off to the side. If the two sides are equal, either D is heavy or G or H is light. If ABE is heavy, either A or B is heavy, or F is light. If ABE is light, then either E is light or C is heavy. In any case, jump to 'Three coins.' (We'll need three coins for the next weighing, so if ABE was light, send J along with E and C.)

Four coins, two weighings remaining: Call these coins IJKL. We know nothing about whether they are heavy or light. Weigh I against J. If they are equal, either K or L is the unsafe coin. Send them (along with J) to 'Three coins.' If I and J aren't equal, send them instead (along with K).

Three coins, last weighing: At this point, we either have three suspects with some information about which might be heavy and which might be light, or two suspects and a control coin. In the case of two suspects and a control, weigh one of the suspects against the control. If they are equal, the other suspect is the unsafe coin. If they aren't, it's the suspect on the scale. In the other case, we either have two coins that might be heavy and one that might be light, or two that might be light and one that might be heavy. In the first case, weigh the two maybe-heavies against each other. If they are equal, the maybe-light is unsafe. If they aren't, whichever one is heavier is the problem coin. In the second case, weigh the two maybe-lights against each other. This time, if one of them is lighter than the other, the lighter one is the 'winner.' Of course, if they are equal, the coin not on the scale wins.