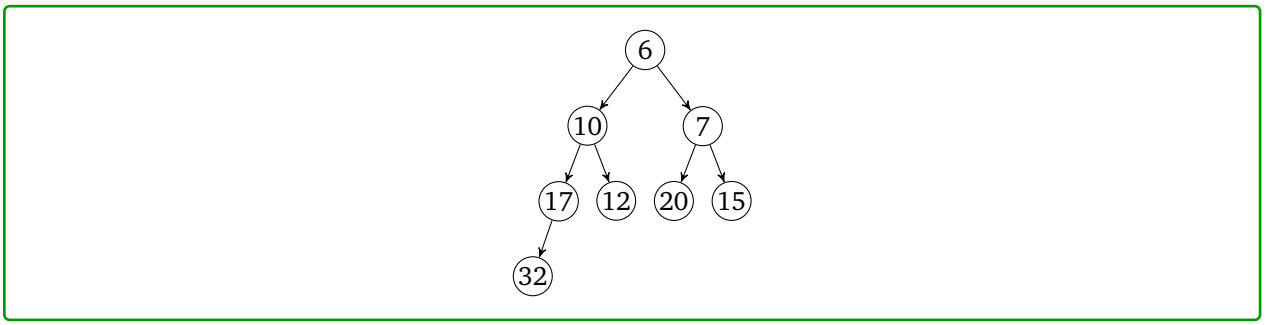
Section 06: Solutions

1. Heaps

(a) Insert the following sequence of numbers into a *min heap*:

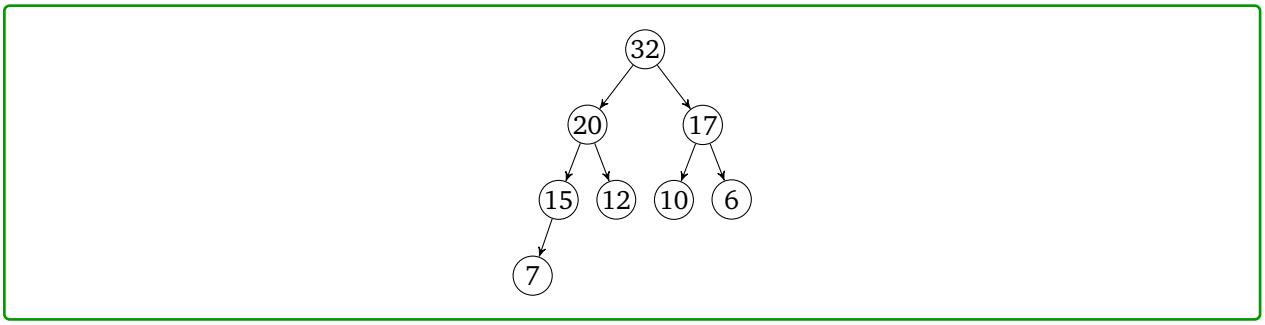
[10, 7, 15, 17, 12, 20, 6, 32]

Solution:



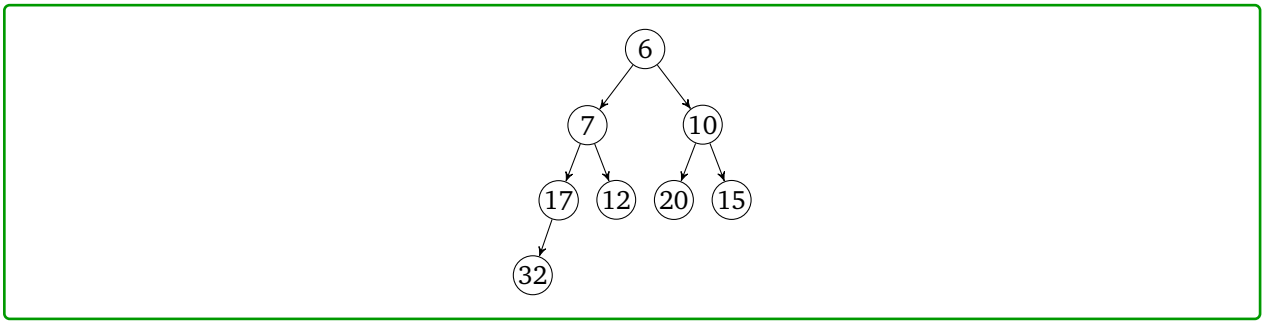
(b) Now, insert the same values into a *max heap*.

Solution:



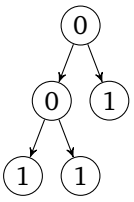
(c) Now, insert the same values into a *min heap*, but use Floyd's `buildHeap` algorithm.

Solution:



(d) Insert 1, 0, 1, 1, 0 into a *min heap*.

Solution:



2. Sorting

(a) Demonstrate how you would use quick sort to sort the following array of integers. Use the first index as the pivot; show each partition and swap.

[6, 3, 2, 5, 1, 7, 4, 0]

Solution:

[Solutions omitted]

(b) Show how you would use merge sort to sort the same array of integers.

Solution:

[Solutions omitted]

(c) Suppose we have an array where we expect the majority of elements to be sorted "almost in order". What would be a good sorting algorithm to use?

Solution:

Merge sort and quick sort are always predictable standbys, but we may be able to get better results if we try using something like insertion sort, which is $\mathcal{O}(n)$ in the best case.

Alternatively, we could try using an adaptive sort such as Timsort, which is specifically designed to handle almost-sorted inputs efficiently while still having a worst-case $\mathcal{O}(n \log(n))$ runtime.

3. Analyzing recurrences, redux

Consider the following recurrences. For each recurrence, (a) find a closed form using the tree method and (b) check your answer using the master theorem.

(a) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 8T(n/2) + 4n^2 & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Number of nodes on level i: 8^i
- Total work done per level: $8^i \cdot 4 \left(\frac{n}{2^i}\right)^2 = 8^i \cdot 4 \cdot \frac{n^2}{4^i}$
- Total number of *recursive* levels: $\log_2(n)$
- Total work done in base case: $8^{\log_2(n)}$

So we get the expression:

$$\left(\sum_{i=0}^{\log_2(n)-1} 8^i \cdot 4 \cdot \frac{n^2}{4^i} \right) + 8^{\log_2(n)}$$

We can simplify by pulling the $4n^2$ out of the summation:

$$4n^2 \left(\sum_{i=0}^{\log_2(n)-1} \frac{8^i}{4^i} \right) + 8^{\log_2(n)}$$

This further simplifies to:

$$4n^2 \left(\sum_{i=0}^{\log_2(n)-1} 2^i \right) + 8^{\log_2(n)}$$

After applying the finite geometric series identity, we get:

$$4n^2 \cdot \frac{2^{\log_2(n)} - 1}{2 - 1} + 8^{\log_2(n)}$$

This is a closed form so we could stop, but if we want a tidy solution, we can continue simplifying:

$$\begin{aligned} T(n) &= 4n^2 \cdot \frac{2^{\log_2(n)} - 1}{2 - 1} + 8^{\log_2 2(n)} \\ &= 4n^2 \cdot \left(2^{\log_2(n)} - 1\right) + 8^{\log_2(n)} \\ &= 4n^2 \cdot \left(n^{\log_2(2)} - 1\right) + n^{\log_2(8)} \\ &= 4n^2 \cdot (n - 1) + n^3 \\ &= 5n^3 - 4n^2 \end{aligned}$$

We can apply the master thereom here. Note that $log_b(a) = \log_2(8) = 3 > 2 = c$, which means that $T(n) \in \Theta\left(n^{\log_b(a)}\right)$ which is $T(n) \in \Theta\left(n^{\log_2 8}\right)$ which in turn simplifies to $T(n) \in \Theta\left(n^3\right)$.

This agrees with our simplified form.

(b) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 7T(n/3) + 18n^2 & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Number of nodes on level i: 7^i
- Total work done per level: $7^i \cdot 18 \left(\frac{n}{3^i}\right)^2 = 7^i \cdot 18 \cdot \frac{n^2}{9^i}$
- Total number of *recursive* levels: $\log_3(n)$
- Total work done in base case: $7^{\log_3(n)}$

So we get the expression:

$$\left(\sum_{i=0}^{\log_3(n)-1} 7^i \cdot 18 \cdot \frac{n^2}{9^i} \right) + 7^{\log_3(n)}$$

We can simplify by pulling the $18n^2$ out of the summation:

$$18n^2 \left(\sum_{i=0}^{\log_3(n)-1} \frac{7^i}{9^i} \right) + 7^{\log_3(n)}$$

This is equivalent to:

$$18n^2 \left(\sum_{i=0}^{\log_3(n)-1} \left(\frac{7}{9}\right)^i \right) + 7^{\log_3(n)}$$

After applying the finite geometric series identity, we get:

$$18n^2 \cdot \frac{\left(\frac{7}{9}\right)^{\log_3(n)} - 1}{\frac{7}{9} - 1} + 7^{\log_3(n)}$$

This is a closed form so we could stop, but if we want a tidy solution, we can continue simplifying:

$$\begin{aligned} T(n) &= 18n^2 \cdot \frac{\left(\frac{7}{9}\right)^{\log_3(n)} - 1}{\frac{7}{9} - 1} + 7^{\log_3(n)} \\ &= 18n^2 \cdot \frac{\left(\frac{7}{9}\right)^{\log_3(n)} - 1}{-\frac{2}{9}} + 7^{\log_3(n)} \\ &= -81n^2 \cdot \left(\left(\frac{7}{9}\right)^{\log_3(n)} - 1 \right) + 7^{\log_3(n)} \\ &= -81n^2 \cdot \left(n^{\log_3(7/9)} - 1 \right) + n^{\log_3(7)} \\ &= -81n^2 \cdot \left(n^{\log_3(7)-2} - 1 \right) + n^{\log_3(7)} \\ &= -81n^2 n^{\log_3(7)-2} + 81n^2 + n^{\log_3(7)} \\ &= -80n^{\log_3(7)} + 81n^2 \end{aligned}$$

We can apply the master thereom here. Note that $\log_b(a) = \log_3(7) < 2 = c$, which means that $T(n) \in \Theta\left(n^c\right)$ which is $T(n) \in \Theta\left(n^2\right)$

This agrees with our simplified form.

(c) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n/2) + 3 & \text{otherwise} \end{cases}$

Solution:

Given this recurrence, we know...

- Number of nodes on level i: $1^i = 1$
- Total work done per level: $1 \cdot 3 = 3$
- Total number of *recursive* levels: $\log_2(n)$
- Total work done in base case: $1^{\log_3(n)} = 1$

So we get the expression:

$$\left(\sum_{i=0}^{\log_2(n)-1} 3 \right) + 1$$

Using the summation of a constant identity, we get:

$$3\log_2(n) + 1$$

We can apply the master thereom here. Note that $\log_b(a) = \log_2(1) = 0 = c$, which means that $T(n) \in \Theta\left(n^c \log(n)\right)$ which is $T(n) \in \Theta\left(n^0 \log(n)\right)$ which further simplifies to $T(n) \in \Theta\left(\log(n)\right)$.

This agrees with our simplified form.

4. Divide and conquer

(a) Suppose we have an array of sorted integers that has been *circularly shifted* k positions to the right. For example, $[35, 42, 5, 10, 20, 30]$ is a sorted array that has been circularly shifted $k = 2$ positions, while $[27, 29, 35, 42, 5, 9]$ is a sorted array that has been shifted $k = 4$ positions.

Now, suppose you are given a sorted array that has been shifted an unknown number of times – we do not know what k is.

Describe how you would implement an algorithm to find k in $\mathcal{O}\left(\log(n)\right)$ time.

Solution:

First, notice that the smallest element in our array will tell us precisely how many times the array has been shifted, since the smallest element should go in index 0. This is equivalent to looking for an element in our array which is smaller than its neighbor to the left. Additionally, notice that given two elements on the circular sorted array, the smallest element in the array (if it isn't one of the selected elements) cannot be to the right of the smaller element and left of the larger. Therefore, we know it must exist on the other part of the array.

Algorithm:

Begin by choosing an arbitrary element of the list - for the purpose of this example, we'll pick the first element. Then, if we did not find the smallest element (by checking the element to the left of it), we can check the $\frac{n}{2}$th element in the list. If that element is greater than the first element, then we know that the smallest element must be in the second-half of the array. If that element is less than the first element, then we know that the smallest element in the array must be in the first-half of the array. In any case, we know that the half-array must still be a circular sorted array, since we only remove elements, which cannot break the sortedness of an array.

Therefore, we can recurse on this half array, guaranteeing that the smallest element overall is on the half array. When the array size is 1, we certainly must have found the smallest element.

(b) Suppose we have some Java method `double foo(int n)`. This function is *monotonically decreasing* – this means that as we keep plugging in larger and larger values of `n`, the `foo(...)` method will keep returning smaller and smaller numbers.

More specifically, for any integer `i`, it is always true that `foo(i) > foo(i + 1)`.

We want to find the smallest value of `n` that when plugged in will make `foo(...)` return a negative number.

Describe how you would implement a $\mathcal{O}(\log(n))$ algorithm to do this (where n is the final answer).

Solution:

Algorithm:

Check `foo(`2^k`)`, incrementing k until the output of the function is negative.

This gives us bounds on n, specifically that $2^{c-1} < n \leq 2^c$ for whichever c we end up on (which is also approximately $\log(n)$). Finally, we can perform a binary search over the elements between 2^{c-1} and 2^c to find the exact value of n.

Ultimately, we first do c checks to find the c, then do $c-1$ more checks binary search over the $2^c - 2^{c-1} = 2^{c-1}$ remaining elements on the range. This results in $\mathcal{O}(c)$ checks overall, which is on order of $\mathcal{O}(\log(n))$ as desired.

(c) Describe how you would modify merge sort so that it can sort a singly linked list in $\mathcal{O}(n \log(n))$ time. Your algorithm should modify the linked list in place, without needed extra data structures.

Solution:

We first split the array by obtaining a pointer to the middle of the linked list, then splitting it in two.

We can find the middle in one of two ways. One way is to have the linked list always maintain its size and loop size/2 times. The other way is to have two pointers. Both pointers begin at the start, but one moves two spaces per iteration and the other moves only one. Once the "move-two-spaces" pointer reaches the end of the linked list, we know the first pointer must be in the middle.

Either way, this takes $\mathcal{O}(n)$ time since we need to examine $\frac{n}{2}$ nodes. Once we have the middle pointer, split (which takes $\mathcal{O}(1)$ time) and recurse.

We can then merge using nearly the identical algorithm we had for arrays in $\mathcal{O}(n)$ time.

We end up with the exact same recurrence as the original version of merge sort, which means the runtime must be $\mathcal{O}(n \log(n))$.

(d) Describe how you would modify your answer from the previous question to randomly shuffle a linked list in $\mathcal{O}(n \log(n))$ time. As before, your algorithm should modify the linked list in place, again without needing any extra data structures.

Solution:

Modify the merge step so that instead of moving the smallest item from the two linked lists to the combined one, pick them at random.

So, instead of combining two sorted lists into a bigger sorted list, our `merge(a, b)` algorithm will instead take two (randomized) lists and produce another randomized list.

5. Divide and conquer: challenge questions

(a) Design an algorithm that accepts an unsorted array of integers and finds the subarray with the maximum possible sum.

For example, consider the array `[2, -4, 1, 9, -6, 7, -3]`. The maximum subarray would be `[1, 9, -6, 7, -3]`, which sums to 11.

A naive solution that considers every possible subarray would take $\mathcal{O}\left(n^2\right)$ time. Design a more efficient algorithm that uses divide and conquer and runs in $\mathcal{O}\left(n \log(n)\right)$ time.

Challenge: Can you design an algorithm that runs in $\mathcal{O}\left(n\right)$ time?

Solution:

For an explanation of how to solve this in $\mathcal{O}\left(n \log(n)\right)$ time, see this webpage: `https://www.geeksforgeeks.org/divide-and-conquer-maximum-sum-subarray/`

The basic intuition is that we split the array in half, recursive on both the left and the right, and find the largest subarray that crosses over the middle. We then return the max of those three numbers.

The trick ends up being that if we insist that a subarray must cross over the middle, we can actually find the largest possible subarray sum in $\mathcal{O}\left(n\right)$ time. So we end up doing $2T(n) + n$ work per each recursive call.

For an explanation of how to solve this in $\mathcal{O}\left(n\right)$ time, see this webpage: `https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/`.

The basic intuition is start with a `sum` variable set to 0. We then loop over the array from the start and add each element we encounter to `sum`.

If at any point, `sum` becomes negative, we know that particular sequence of values is irrecoverable. We're better off "abandoning" that particular subarray and starting over at that particular index.

(b) Given an array containing elements of type `E` design an algorithm that finds the **majority element** – that is, an element that appears more then $n/2$ times. If no majority element exists, return `null`.

Your algorithm should run in $\mathcal{O}\left(n \log(n)\right)$ time (and use only $\mathcal{O}\left(1\right)$ extra memory).

Note: the items in the array do **NOT** implement `compareTo`. This means you cannot sort the array!

Challenge: can you find the majority in $\mathcal{O}\left(n\right)$ time and $\mathcal{O}\left(1\right)$ extra memory?

Solution:

The $\mathcal{O}\left(n \log(n)\right)$ solution works by first splitting the array into two halves. We recurse on both halves and receive back the majority elements for the two halves (if they exists).

Once we finish recursing, there are four different scenarios:

(a) The two subarrays have the same majority element.

This means, by definition, that element must also be the majority of the full array.

Why is this? Suppose that there are n elements in the overall array. If A is the majority of the left half, then that means that by definition, there must be $> \frac{n}{4}$ occurrences of A on the left. Similarly, if A is the majority on the right, there must be $> \frac{n}{4}$ occurrences there.

Therefore, there must be $> \frac{n}{2}$ occurrences of A overall. So we can just return A without needing to check anything else.

(b) The two subarrays have different majority elements.

In that case, we need to figure out which one is the true majority. We take the majority element from the left and loop over the entire array to figure out how many times it appears. We do the same thing with the majority element from the right. This will take $\mathcal{O}(2n) = \mathcal{O}(n)$ time.

If either of them appear more then $\frac{n}{2}$ time, return that element as the majority. If neither of them appear enough times, return null (or whatever else we're using to indicate that there's no majority).

(c) Only one subarray has a majority; the other doesn't.

We do the same sort of looping thing as before, again in $\mathcal{O}(n)$ time.

(d) Neither subarrays have a majority.

We can automatically give up here, for basically the same reason why we could automatically return in case 1.

We end up doing $2T(n) + n$ work in the worst case in the recursive case, which results in $\mathcal{O}(n \log(n))$.

The $\mathcal{O}(n)$ solution is called "Boyer-Moore majority vote algorithm." The Wikipedia page has a good overview of how the algorithm works: `https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_majority_vote_algorithm`

(c) Suppose we have two polynomials represented as two int arrays, where the i-th item represents the i-th coefficient. So, the array `[5, 10, 0, 2, -3]` would represent the polynomial $5 + 10x + 2x^3 - 3x^4$.

Design an algorithm that accepts two of these arrays and returns a new one representing the product of the two. You may assume both input arrays both have length n. A naive implementation using nested loops will have $\mathcal{O}(n^2)$ work; your algorithm must be asymptotically better.

Hint: Note that a polynomial A can be written as $A_0 + A_1 x^{n/2}$, where A_0 is the first $n/2$ terms and A_1 is the latter $n/2$ terms. This means that $A \cdot B = (A_0 + A_1 x^{n/2}) \cdot (B_0 + B_1 x^{n/2})$. With some algebra, we can simplify to obtain:

$$A \cdot B = A_0 B_0 + ((A_0 + A_1)(B_0 + B_1) - A_0 B_0 - A_1 B_1) x^{n/2} + A_1 B_1 x^{n/2}$$

This means that computing the product of A and B requires you to multiply polynomials exactly three times. (Note: not 5 times – why?). You should exploit this property when implementing your algorithm.

Solution:

This algorithm is known as "Karatsuba multiplication" and is used to efficiently multiply large integers and polynomials. You can find more detailed info about this algorithm (and faster variations!) online.

The core idea is that to compute our final equation, we actually only need to multiply together three distinct polynomials:

- $A_0 B_0$
- $A_1 B_1$
- $(A_0 + A_1)(B_0 + B_1)$

This means our runtime will end up forming the following recurrence:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 3T(n/2) + n & \text{otherwise} \end{cases}$$

That is, within a single recursive call, we multiply together three polynomials that are all half the size of the input polynomial. We do an additional $\mathcal{O}(n)$ work per each recursive call to add and subtract everything together.

The final algorithm looks roughly like so:

```
int[] multiplyPoly(int[] A, int[] B):
    if A.length == 0 or B.length == 0:
        return empty array
    else if A.length == 1 and B.length == 1:
        return [A[0] * B[0]]
    else:
        int n = A.length;
        int mid = n / 2;

        int[] A0, A1 = A.sublist(0, mid), A.sublist(mid, n)
        int[] B0, B1 = B.sublist(0, mid), B.sublist(mid, n)

        int[] X = multiplyPoly(A0, B0)
        int[] Y = multiplyPoly(A1, B1)
        int[] Z = multiplyPoly(add(A0, A1), add(B0, B1))

        int[] out = new int[2 * n]
        add X to out starting at 0
        add Z - X - Y to out starting at mid
        add Y to out starting at n

        return out
```

(d) Suppose you are trying to write a video game containing thousands of different moving elements and want to check if two elements have collided or overlapped.

A naive way of implementing this would be to use two nested loops and check every pair of elements. This often ends up being too inefficient for most video games, even for only a few thousand elements (especially games that require a high degree of responsiveness).

Describe how you would design a data structure to store these points in a way that lets you more efficiently check whether two elements are colliding.

For the sake of simplicity, you may assume that each element is a circle and has a relatively small radius. You may also assume that the elements are moving on a 2d plane – you don't need to worry about collisions in 3d.

As a hint: think about recursively subdividing the 2d plane.

Solution:

We can solve this by creating a kind of a data structure known as a "quadtree" – in this case, probably using a variation known as a "region quadtree".

A "quadtree", as its name suggests, is a kind of tree. Each branch node (in a region quadtree) represents one rectangular region within the 2d plane. A branch node also has at most four children that contain all elements located within the upper-left, upper-right, lower-left, and lower-right corners of that rectangle respectively.

Each child can either be another branch node (which contains four more children), or a leaf node (representing a single gameplay element).

Now, suppose we want to insert a new element into an existing quadtree. To do so, we run the following algorithm:

(a) We assume the root node is a branch node. We take the coordinate of the point we want to add and determine which of the four quadrants it belongs in.

(b) This branch node could either be null, point to a leaf node, or point to another branch node.

    (a) If it's null, add a new leaf node for that coord there.

(b) If it's a leaf node, replace the leaf node with a branch node containing both the point that was originally there, as well as the new one.

(c) If it's a branch node, recurse and repeat this entire procedure.

This ends up forming a tree, where the regions that contain a lot of points end up being deeply nested, and the regions with few points end up being shallow.

This data structure also lets us search for all points within a certain bounding relatively efficiently: since each branch node stores information about what region in the 2d plane its supposed to contain, we can recursively search and find all leaf nodes that fall within those coordinates without having to search through every single existing point.

Now, to check if a given point is colliding with any other one, we no longer need to compare it against every single other point. Instead, we just find the leaf node corresponding to our point, move up a level or two, and look at all of the children to get the list of all points that happen to be close by.

The core idea is we were able to speed up traversal by recursively dividing up our points into different regions based on their x-y coordinates in the planes.

For more details, see this article: https://gamedevelopment.tutsplus.com/tutorials/quick-tip-use-quadtrees-to-detect-likely-collisions-in-2d-space–gamedev-374

You may also want to try googling "quadtree visualization".