Solutions to Homework 1 Practice Problems

## [DPV] Problem 2.1 – Practice Multiplication

The product is 28,830. See the final page of this file to visualize the value calculated at each level of the recursive stack.

## [DPV] Problem 2.5 – Recurrence

**Solutions:**

- **(a)** $T(n) = 2T(n/3) + 1 = O(n^{\log_3 2})$ by the Master theorem
- **(b)** $T(n) = 5T(n/4) + n = O(n^{\log_4 5})$ by the Master theorem
- **(c)** $T(n) = 7T(n/7) + n = O(n \log_7 n) = O(n \log n)$ by the Master theorem
- **(d)** $T(n) = 9T(n/3) + n^2 = O(n^2 \log_3 n) = O(n^2 \log n)$ by the Master theorem
- **(e)** $T(n) = 8T(n/2) + n^3 = O(n^3 \log_2 n) = O(n^3 \log n)$ by the Master theorem
- **(f)** $T(n) = 49T(n/25) + n^{3/2} \log n = O(n^{3/2} \log n)$
  - Hint: the contribution of level $i$ of the recursion is $(\frac{49}{125})^i O(n^{3/2} \log n)$.
- **(g)** $T(n) = T(n-1) + 2 = O(n)$
  - This can be found through substitution:

$$\begin{aligned} T(n) &= T(n-1) + 2 \\ &= (T((n-1)-1) + 2) + 2 \\ &= T(n-2) + 2 + 2 \\ &= \dots \\ &= T(n-n) + 2n \end{aligned}$$

- **(h)** $T(n) = T(n-1) + n^c = \sum_{i=1}^{n} i^c + T(0) = O(n^{c+1})$

$$\begin{aligned} T(n) &= T(n-1) + n^c \\ T(n) &= (T((n-1)-1) + (n-1)^c) + n^c \\ &= T(n-2) + (n-1)^c + n^c && \text{(we simplify)} \\ T(n) &= (T((n-2)-1) + (n-2)^c) + (n-1)^c + n^c \\ &= T((n-3) + (n-2)^c) + (n-1)^c + n^c \end{aligned}$$

- This pattern repeats n times until $T(n-n) = T(0)$; the $(n-1)^c$ resolves to a polynomial where $n^c$ is the dominating term. This repeats $n$ times, giving $O(n * n^c) = O(n^{(c+1)})$

- **(i)** $T(n) = T(n-1) + c^n = \sum_{i=1}^{n} c^i + T(0) = O(c^n)$

  - This can be solved similar to (h); This pattern repeats $n$ times until $T(n-n) = T(0)$; we have a series $c^0 + c^1 + \ldots + c^n$, the last term dominates giving $O(c^n)$

- **(j)** $T(n) = 2T(n-1) + 1 = \sum_{i=0}^{n-1} 2^i + 2^n T(0) = O(2^n)$
- **(k)** $T(n) = T(\sqrt{n}) + 1 = \sum_{i=0}^{k} 1 + T(b)$

  - where $k$ is an integer such that $n^{(1/2^k)}$ is a small constant $b$ (the size of the base case). This implies that $k = O(\log \log n)$ and $T(n) = O(\log \log n)$.

## [DPV] Problem 2.16 – Infinite Array

### (a) Algorithm:

Starting with $index = 1$ and doubling the index on each iteration, we continue until $A[index] \geq x$. This efficiently finds an upper bound for our target value using an exponential search.

Perform a black box Binary Search from $A[1]$ to $A[index]$ for the target value of $x$, and return the result of that search directly.

### (b) Justification of Correctness:

The array is sorted, which allows the use of both the exponential search and the binary search. By doubling the index at each step, we can deterministically find an upper bound that must be either infinity or a value greater than the target value we are trying to find. We know the search for this upper bound must terminate because we exceed either the target value or $n$ elements (at which point $\infty$ occurs). Once we have this upper bound, we can leverage binary search on the now-bounded portion of the array to find the target value.

### (c) Runtime Analysis:

By doubling the index value at each iteration, we are exploring all $A[2^i]$ where $0 \leq i \leq \log(2n)$. This results in $O(\log n)$ since the highest our upper bound search could have gone is $2n$, only in the case where $x > A[n]$.

The known runtime of the binary search black box is $O(\log n)$, since the bounded portion of the array is at most 2n elements.

The final runtime is thus $O(\log n) + O(\log n) = O(\log n)$

# [DPV] Problem 2.23a – Majority Element

## (a) Algorithm:

Recursively divide the input into two equal-sized subarrays, $A_1$ and $A_2$. When a base case of a subarray of size 1 is reached, report that value as the *majority element*.

At each level of recursion, combine the results from each left/right subarray, as such:

1. Check the *majority element* from each subarray, if it exists.
2. For each *majority element*, count its number of occurrences in the combined array
3. If one of the elements remains a majority for the combined array, then report it as the *majority element*, else **NO**.

When the recursion unwinds to the original array A, return the *majority element* if it exists, else **NO**.

## (b) Justification of Correctness:

By recursively dividing the input space, we can efficiently compute a count at each level of recursion to obtain a final result when the recursion unwinds. If a *majority element* exists at all, then at every level of recursion, there must be at least one subarray in which that element can be found as the *majority element*. Thus, as the recursion unwinds, that *majority element* is guaranteed to be passed to each successive level of recursion until the algorithm completes.

## (c) Runtime Analysis:

At each level of recursion, we have split the work into two halves ($b = 2$), operating on both halves ($a = 2$), and performing O(n) ($d = 1$) work by counting the number of occurrences at each level. This gives us a recurrence of $T(n) = 2T(n/2) + O(n)$, which solves to $O(n \log n)$ by the Master Theorem.

# [DPV] Problem 2.23b – Majority Element (linear)

### (a) Algorithm:

Pair up the elements of $A$ arbitrarily, to get $\frac{n}{2}$ pairs. Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them.

Each time there is an odd number of elements, take one element and check if it is the majority by comparing against all elements at the current level. If it is, return it. If it is not the majority element, discard it.

Do this until there are one or two elements left.

Check each remaining element (two max) against all elements in the original $A$ to see if either of them represent a majority. Return this element if a majority exists.

### (b) Justification of Correctness:

If there is a majority, x, then at least $\frac{n}{2} + 1$ of x exists. So after every round of elimination, x would remain the majority because it would create more pairs than all other elements combined. For each x not paired with an x, a non x also does not get paired with itself (paired with x instead). So x pairs always outnumber any other pair. If there is an odd number of elements, we can eliminate the extra element by checking to see if it is a majority element or not. Finally we check the remaining elements (two max) against the original array to confirm if there was a majority.

### (c) Runtime Analysis:

The recurrence relation is $T(n) = T(n/2) + O(n) = O(n)$

- We split the problem in half each time, and do n comparisons at each recurrence
- For a=1, b=2, d=1, this is $O(n)$ via Master Theorem

Then we do two scans to check that x is a majority, so $2 * O(n)$

Giving us a overall runtime of $3 * O(n) = O(n)$

## Binary Search Modified

*Design an $O(log(n))$ algorithm to find the smallest missing natural number in a given sorted array. The given array only has distinct natural numbers. For example, the smallest missing natural number from $A = \{3, 4, 5\}$ is 1 and from $A = \{1, 3, 4, 6\}$ is 2.*

## Solution:

### (a) Algorithm:

To solve our problem we will modify Binary Search in the following way. Once the midpoint index is found, we check if the value at the midpoint is equal to the midpoint; that is, does $A[mid] = mid$. If it does, that tells us all the expected values to the left of the midpoint are in place, so we recurse on the right half of the array. If the value found is greater then the midpoint, we know there is at least one missing value to the left, so we recurse on that half.

Our base case is when we get to a single value. If that value, $A[i]$ equals it's index $i$, the first missing value is $i + 1$. If the value does not equal the index, the missing value is $i$. We return the missing value.

### (b) Justification of Correctness:

Why does this work? Our input is natural numbers sorted in non-decreasing order, which allows us to take advantage of Binary Search. As natural numbers begin at 1, then the missing value is the lowest $i$ such that $A[i] \neq i$. When we check if $A[mid] = mid$, if they match we know all the values $1 \ldots mid$ are present. If they don't match, we know some value in the range $1 \ldots mid - 1$ is missing.

### (c) Runtime Analysis:

Our modification to Binary Search simply changes the comparison and return value. Both changes remain constant time operations, so there is no impact to the original $O(\log n)$ runtime.

---

**NOTE**: We can convince ourselves of this by considering the following: at each level of recursion we perform one constant time comparison and then solve a single recursive problem on just one-half of the current input.

---

*[DPV] 2.1 Use the divide-and-conquer integer multiplication algorithm to multiply the two binary integers 10011011 and 10111010.*

This tree displays the return values from each level of the recursion; the values are ordered A,B,C for the formula $return = A \times 2^n + (C - A - B) \times 2^{n/2} + B$