# CS3510-A: Design and Analysis of Algorithms, Spring 2021

Homework-4

February 9, 2021

**DUE DATE: Tuesday, February 16, 1l:59pm**

**Note-1:** Your homework solutions should be electronically formatted as a single PDF document that you will upload on Gradescope. If you have to include some handwritten parts, please make sure that they are very clearly written and that you include them as high resolution images.
**Note-2:** Please think twice before you copy a solution from another student or resource (book, web site, etc). It is not worth the risk and embarrassment.
**Note-3:** You need to **explain/justify** your answers. Do not expect full credit if you just state the correct answer.
**Note-4: You will get 2 extra points if you submit electronically typed solutions instead of hand-written.**

## Problem-1 (36 points)

The Master Theorem of Recurrence we covered in class is not applicable in all recursions. For instance, it does not apply on the following recursions.

Solve each of the following recursions. You do not need to give inductive proofs. We are only looking for the asymptotic run-time complexity of each recursion using the Big-Theta notation.

a) $T(n) = 49T(\frac{n}{25}) + n^{(3/2)} * \log n$

b) $T(n) = T(n-1) + c^n$, where $c > 1$ is a constant

c) $T(n) = 2\,T(n-1) + 1$

### Solution

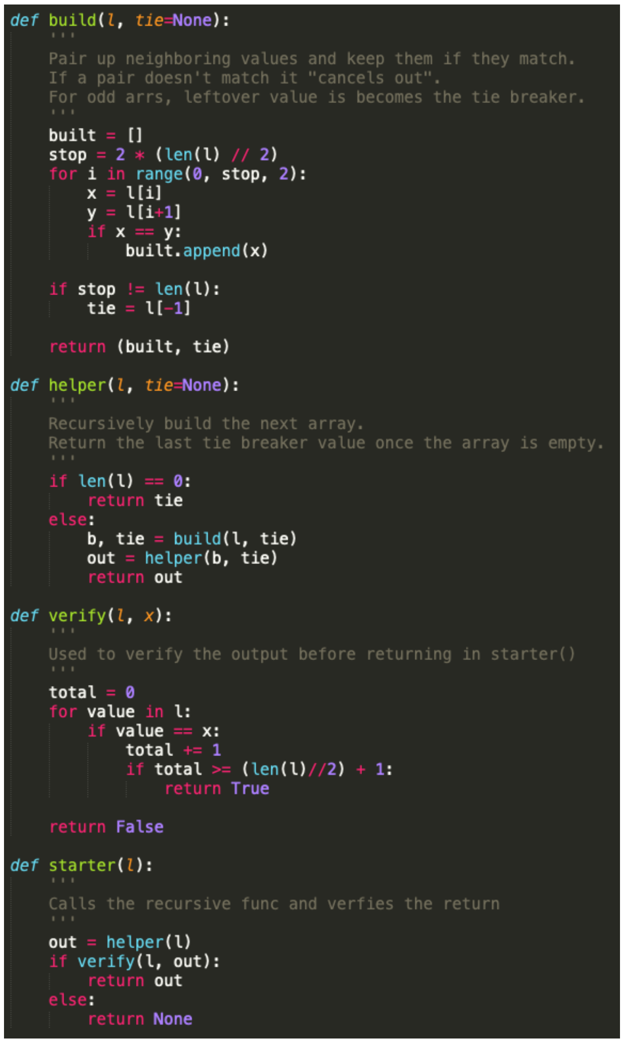
a) $\theta(2^n)$
b) $\theta(c^n)$
c) $\theta(2^n)$

## Problem-2 (34 points)

In Lesson 8.2, Constantine described at a high-level a divide-and-conquer linear-time algorithm to compute the majority element of an array.

In this exercise, you are asked to write detailed pseudocode for that algorithm. Your algorithm should cover the case that either the original array $A$ or the reduced array $A'$ has an odd number of elements. It should also cover any other corner cases you can think of.

### Solution



```python
def build(l, tie=None):
    '''
    Pair up neighboring values and keep them if they match.
    If a pair doesn't match it "cancels out".
    For odd arrs, leftover value is becomes the tie breaker.
    '''
    built = []
    stop = 2 * (len(l) // 2)
    for i in range(0, stop, 2):
        x = l[i]
        y = l[i+1]
        if x == y:
            built.append(x)

    if stop != len(l):
        tie = l[-1]

    return (built, tie)

def helper(l, tie=None):
    '''
    Recursively build the next array.
    Return the last tie breaker value once the array is empty.
    '''
    if len(l) == 0:
        return tie
    else:
        b, tie = build(l, tie)
        out = helper(b, tie)
        return out

def verify(l, x):
    '''
    Used to verify the output before returning in starter()
    '''
    total = 0
    for value in l:
        if value == x:
            total += 1
            if total >= (len(l)//2) + 1:
                return True

    return False

def starter(l):
    '''
    Calls the recursive func and verfies the return
    '''
    out = helper(l)
    if verify(l, out):
        return out
    else:
        return None
```

### Explanation

The algorithm makes pairs through the list. Any pairs that match get passed down to the array for the next recursion. Non-matching pairs ”cancel” eachother’s votes out and are removed. For odd-length lists the leftover entry becomes a tie-breaker value. Once the function receives an empty array it will return the tie-breaker value, after verifying the value by counting the number of occurrences in the original array.

## Runtime

General runtime for this algo:
$T(n) = T(n/2) + O(n)$

For the master theorem, the values are:
$\alpha = 1$: because each call of the function results in one recursive call.
$\beta = 2$: because the size of the array for each next call will be at most of size $n/2$
$\gamma = 1$: because the work of verifying the solution is $O(n)$, meaning for $O(n^\gamma)$, $\gamma = 1$

In this case: $\gamma > \log_\beta \alpha$ $(1 > \log_2 1)$
Therefore this algorithm has: $O(n^\gamma)$ or $O(n)$.

## Problem-3 (30 points)

Design an algorithm to multiply any $n$-bit integer with any $m$-bit integer where $n \geq m$ in $O(n\, m^{\log_2(3)-1})$ time.

### Solution

**Algorithm 1** multiplySameSize$(x, y)$

```
Require: two $m$-bit sized integers
  $m \leftarrow$ bit size of $x$ (or $y$)
  $x_l, x_r \leftarrow$ bits of $x$ split at $\lfloor m/2 \rfloor$
  $y_l, y_r \leftarrow$ bits of $y$ split at $\lfloor m/2 \rfloor$
  $a \leftarrow$ multiplySameSize$(x_l, y_l)$
  $b \leftarrow$ multiplySameSize$(x_r, y_r)$
  $c \leftarrow$ multiplySameSize$(x_l + x_r, y_l + y_r)$
  return $(a \cdot 2^x) + ((c - a - b) \cdot 2^{x/2}) + b$
```

**Algorithm 2** multiply$(x, y)$

```
Require: int $x$ with size $n$; int $y$ with size $m$ (where $n \geq m$)
  $iters \leftarrow \lceil n/m \rceil$
  zero-pad $x$ to be size: $m * iters$
  $total \leftarrow 0$
  for $j \in 0..iters$ do
    $x_j \leftarrow$ the $j$-th chunk of $m$ bits in $x$
    $c \leftarrow$ multiplySameSize$(x_j, y)$
    $c \leftarrow c \cdot 2^{m \cdot j}$ (leftshift section to match original position in x)
    $total$ += $c$
  end for
  return $total$
```

### Explanation

A multiplication can be done using the multiplication of: the first halves of each number, second halves of each number, and sum of the halves for each number. The result can be found with the components and some addition/subtraction, as well as multiplication with powers of two (which can be easily done with left shifts). Once the function recursively halves the inputs down to 1-bit numbers, then multiplication can be done with a binary AND.
If the two numbers are different sizes then the second function multiply() can be used to split up the larger integer into chunks of the size of the smaller integer. If the larger integer doesn't cleanly divide into chunks with sizes of the smaller integer, then the larger integer can be zero-padded to make it large enough do divide cleanly.

## Runtime

General process for this multiplySameSize() involves:
$T(n) = 3T(n/2) + O(1)$

Master theorem application for multiplySameSize():
$\alpha = 3$: because each call of the multiply function will result in 3 recursive sub-calls.
$\beta = 2$: $n$ will be reduced to $n/2$ for subcalls of multiplySameSize().
$\gamma = 0$: because the work to combine sub calls will be a constant runtime process (the additions/multiplications for the second return line). $O(n^\gamma) = O(1)$ requires $\gamma = 0$.

In this case: $\gamma < \log_\beta \alpha$ $(0 < \log_2 3)$, therefore multiplySameSize() has $O(m^{\log_2 3})$, where $m$ is the bit-size of $x$ and $y$.
During the algorithm, multiply() will call multipySameSize() at least $\lceil n/m \rceil$ times. Therefore multiply() has $O((n/m) \cdot m^{\log_2 3})$ or $O(nm^{\log_2 3 - 1})$.