CS:3330 Homework 9, Spring 2017

Due at the start of class on Thursday, April 27th

1. In Problem 11, Lecture 1 from Jeff Erickson's notes, there is pseudocode for a recursive function called StoogeSort.
   (a) Write down the recurrence relation for the running time of StoogeSort. You are welcome to ignore floors and ceilings, but make sure you write the base case also, expicitly.
   (b) Solve the recurrence using the Master Theorem and find the running time of Stooge-Sort.
2. In Problem 12, Lecture 1 from Jeff Erickson's notes, there is pseudocode for a recursive function called Unusual.
   (a) Write down the recurrence relation for the running time of Unusual. You are welcome to ignore floors and ceilings, but make sure you write the base case also, expicitly.
   (b) Solve the recurrence using the Master Theorem and find the running time of Stooge-Sort.
3. Solve the following recurrences by using the *unroll, guess, confirm* method. You can skip the inductive proof needed for the cofirm step.
   (a) $T(n) = T(n-2) + 2^n$ for $n \geq 2$, $T(1) = 1$, $T(0) = 0$.
   (b) $T(n) = (T(n-2))^2$ for $n \geq 1$, $T(0) = 2$.
   (c) $T(n) = T(n/2) + n$ for $n \geq 2$, $T(1) = 1$.
4. Consider the following recursive function that takes as arguments an array `L` and two non-negative integers `first` and `last`, that serve as indices into `L`. Therefore, if `L` has length n, then `first` and `last` are guaranteed to be in the range 0 through $n-1$.

```
function strangeSum(L, first, last)
    if (last < first) then
        return 0
    if (last = first) then
        return L[first]
    if (last = first + 1) then
        return L[first] + L[first+1]
    else
        m ← last - first + 1
        leftSum ← strangeSum(L, first, first + m/2 - 1)
        midSum ← strangeSum(L, first + m/4, first + 3 * m/4 - 1)
        rightSum ← strangeSum(L, first + m/2, last)
        return leftSum + midSum + rightSum
```

   (a) What is the value returned by the function call `strangeSum(L, 0, 3)` where `L` is the array `[1, 4, 2, 3]`.
   (b) Write a recurrence relation describing the running time the function call `strangeSum(L, 0, n-1)` on an array `L` of length `n`.
   (c) Solve the recurrence in (b) to obtain the running time of the function call `strangeSum(L, 0, n-1)`, in terms of n, the length of the given array `L`.

5. Suppose that we want to compute the value of the expression

$$a^d \bmod n$$

for positive integers a, d, and n, where a and d are guaranteed to belong to $\{1, 2, \ldots, n-1\}$. Since a and d are both guaranteed to be less than n, we know that the input size is $\Theta(\log n)$. Therefore, an efficient (i.e., polynomial time) algorithm for the problem is one that runs in $O(\log^c n)$ for some constant c. In this problem you are required to design *and implement* an $O(\log^3 n)$-time algorithm. In practical terms, I want your function to be able to run very fast (essentially instantaneously) on pretty large numbers, e.g., numbers with roughly 100 digits. You can use your favorite high level language (e.g., `Java`, `C`, `C++`, `Python`, `Scala`, etc.). Specifically, I want you to implement a function – let us call it `bigPowerMod` – that takes as arguments a, d, and n and returns $a^d \bmod n$.

   There are two efficiency-related issues to pay attention to.

   (i) It is possible to compute a^d by performing $d-1$ multiplications. But, this is too many because $d-1$ can be as large as $n-2$ and therefore performing so many multiplications would take $\Theta(n)$ time. For another way of seeing how inefficient this would be suppose that n is a 100-digit number, then your program would be performing, roughly 10^{100} multiplications, which would literally take forever!

   (ii) While your final answer is an integer in the range $[0, n-1]$ (because of the $\bmod n$), you have to watch out for the size of intermediate answers. When we mutiply a number with b bits with another number with b' bits, the answer can have answer many as $b+b'$ bits. This means that if a has $\log_2 n$ bits, then a^2 can have $2\log_2 n$ bits, a^3 can have $3\log_2 n$ bits, and so on. Thus a^d can have $d \log n$ bits, which is $\Theta(n \log n)$ in the worst case. Again, if n is a 100-digit number, this amounts to more that 10^{100} bits (which is more than the number of atoms in the universe!). Of course, carrying such large intermediate answers around also means that each multiplication will take forever.

   You can use *divide-and-conquer* to solve the first problem. Here is a hint: to compute a^d, you can (recursively) compute $a^{d/2}$ and after that it takes one or two multiplications, depending on whether d is even or odd, to compute a^d. This approach allows you to compute $a^d \bmod n$ using $O(\log d)$ multiplications. To solve the second problem, you should note that you can perform $\bmod n$ as soon as you get intermediate answers rather than wait to compute mod n at the very end. This is because of the following property of $\bmod$: $(a \cdot b \cdot c) \bmod n = (((a \cdot b) \bmod n) \cdot c) \bmod n$.

   (a) Print and submit your code. No matter what programming language you use, I expect that your function will be no more than 10 lines long. Make sure your code is well documented.

   (b) Let n be the following 100-digit number.

   ```
   29085119528125578724347048203972299284505302539901
   58990550731991011846571635621025786879881561814989
   ```

   Use the function `bigPowerMod` to compute $(n-100)^{n-1} \bmod n$. Report the answer you get.

   (c) Provide an argument for why the running time of your algorithm/code is $O(\log^3 n)$.

6. Like `mergeSort`, `quickSort` is a sorting algorithm based on the divide-and-conquer paradigm. As we have seen, the worst case running time of `mergeSort` is $\Theta(n \log n)$, as was shown by solving the `mergeSort` recurrence relation. The situation with the running time of `quickSort` is a bit murkier, and more interesting. Here is Python code for `quickSort` that I wrote a few years ago:

```
def partition(L, first, last):
    # Pick L[first] as the "pivot" around which we partition the list
    p = first

    for current in range(p+1, last+1):
        if L[current] < L[p]:
            swap(L, current, p+1)
            swap(L, p, p+1)
            p = p + 1

    return p

def generalQuickSort(L, first, last):
    # Base case: if first == last, there is nothing to do

    # Recursive case: 2 or more elements in the slice L[first..last]
    if first < last:
        # Divide step
        p = partition(L, first, last)

        # Conquer step
        generalQuickSort(L, first, p-1)
        generalQuickSort(L, p+1, last)

        # Combine step: there is nothing left to do!
```

Suppose that L is an already-sorted list (in increasing order) of length n. This problem asks you to figure out the worst case running time of the function call `generalQuickSort(L, 0,` $n-1$`)`. Start by writing a recurrence relation and then solve it.
Hint: The complication with `quickSort` is that the choice of the "pivot" in the `partition` function affects the sizes of the two subproblems that are solved by the recursive calls. This in turn affects the overall running time quite significantly.

7. I want you to now remember the following partitioning problem we considered a while ago. You are given a list L of length n and asked to partition the elements of L into two sublists L_1 and L_2 such that (i) $n/3 \leq |L_1|, |L_2| \leq 2n/3$ and (ii) all elements in L_1 are less than or equal to all elements in L_2.

   We designed a simple, randomized (Las Vegas) algorithm for this problem that ran in $O(n)$ expected time. Let us replace the call to `partition` in the code for `quickSort` given above by a call this Las Vegas algorithm. After we obtain a partition (L_1, L_2) by calling this Las Vegas algorithm, we can simply call `quickSort` on L_1 and then on L_2. Now we have a randomized (Las Vegas) version of `quickSort`. I would like to analyze the expected running time of this algorithm.

   (a) Write down a recurrence relation for the expected running time of this randomized version of `quickSort`.

   (b) Solve this recurrence to obtain an upper bound on the expected running time of this randomized version of `quickSort`.