Artur Idrissov
Due: 11:59pm Thursday, Mar 21

Homework 7

Spring 2023, Bhakta
CS 315, Algorithms

This assignment covers recurrence relations and divide and conquer algorithms. (2.1-2.4 in your book).

1. (20 points) Derive asymptotic bounds for the following recurrences, and give some justification for your answer. For all of these, you may take $T(1) = 1$. For full credit, you must give the best possible bound.

   (a) (4 points) $T(n) = 7T(n/3) + n^2$.

   Solution:
   a = 7, b = 3, d = 2
   $7 < 3^2$
   $7 < 9$
   Final: $O(n^2)$

   (b) (4 points) $T(n) = 27T(n/3) + n^3$.

   Solution:
   a = 27, b = 3, d = 3
   $27 = 3^3$
   $27 = 27$
   Final: $O(n^3 logn)$

   (c) (4 points) $T(n) = 8T(n/4) + n$.

   Solution:
   a = 8, b = 4, d = 1
   $8 > 4^1$
   $8 > 4$
   $O(n^{(log_4(8))})$
   Final: $O(n^{3/2})$

   (d) (4 points) $T(n) = T(n-1) + n^2$. Hint: The master theorem doesn't work here. Use the tree method and math.

   Solution:
   from the tree (shown below), we get:
   $n^2 + (n-1)^2 + (n-2)^2 + ... + 2^2 + 1^1$, for convenience, this can be flipped to get:
   $1^2 + 2^2 + ...(n-1)^2 + n^2$
   For asymptotic bounds, we know that $1^2 + 2^2 + ...(n-1)^2 + n^2 \leq n^2 + n^2 + ... + n^2$.
   Because there are n number of n^2 terms, the asymptotic bound is $O(n^3)$

d).png

Figure 1: Recursion tree for problem 1d

(e) (4 points) $T(n) = 2T(n-1) + 1$. Hint: The master theorem doesn't work here. Use the tree method and math.

Solution:
from the tree (shown below), we get:
$1 + 2^2 + 2^3 + ... + 2^k = 2^{k+1} - 1$ (we get this by applying the geometric progression formula for the sum of the first nth terms)
Let $n - k = 1$, therefore, $n = k + 1$
So, our asymptotic bound is $O(2^n - 1) = O(2^n)$

2. (EXAMPLE) You are given two *unsorted* arrays of distinct integers A and B, where A has length n and B has length $n-1$. B is the same array as A, except that one element, somewhere in the array, has been deleted. The other elements of B are otherwise in exactly the same order as in A. Give an $O(\log n)$ algorithm to find the index of that missing element. Also give pseudocode.

Solution: High level idea: Check the middle of the array. If the arrays match at this point, then the missing element must occur after this point. If they mismatch, then the missing element has occurred already (or is this element). All that remains is to get your indices right.

Pseudocode: In the below, I pass arrays A and B into the function, and indices i, j mean that I am searching through $[i, \cdots, j]$ (including both i and j) in array A.

```
    def missingElement(A,B,i,j)):
```

e).png

Figure 2: Recursion tree for problem 1e

```
    if(i == j):
      return i         # or return A[i], that was ambiguous.
    mid = (i+j)//2
    if A[mid] == B[mid]:
      return binSearch(A,t,mid+1,j)
    else:
      return binSearch(A,t,i,mid)
```

3. (20 points) You want to generate random samples from a set of n objects. However, each item isn't chosen with equal probability. Each index x will be chosen proportionately to its weight $w[x]$, which is given in the array w.

   For example, if $w = [1, 1, 2, 1, 3]$, then we expect elements $0, 1, 3$ to be chosen with the same probability, element 2 to be twice as likely as 0, and element 4 is 3 times as likely as 0.

   If you work out the math, each element should be chosen with probability $p = w[x]/sum(w)$. The probability that element 4 is chosen is $w[4]/sum(w) = 3/8$.

   We will write a program that does $O(n)$ preprocessing, and then takes $O(\log n)$ time to generate each sample. This is more efficient than the straightforward algorithm when generating many samples.

   Random number generation basics: Most modern programming languages offer at least two basic random generation operations.

   - Generate a random integer in the range $[i, j]$ with each element produced with probability $\frac{1}{j-i+1}$.

- Generate a random *real* in the range $[0, 1)$ according to the uniform distribution (all elements equally likely). You can use this to make a weighted choice with probability p by generating a random $[0, 1)$ and checking if the number is $<= p$.

(a) (5 points) In $O(n)$ time total, find and store for each i, the sum of all integers from $w[0]$ to $w[i]$.

For example, if $w = [1, 1, 2, 1, 3]$, then the output array should be $sumw = [1, 2, 4, 5, 8]$.

Write psuedocode/code that can compute this running sum array. After computing this array, explain how to use this array to compute the sum of the elements from $w[i]$ to $w[j]$ in $O(1)$ additional time for any $i < j$.

Solution:

```
The code to compute the running sum array
-----------------------------------------
def runningSumArray(w):
    length = len(w)
    sumw = [0] * length
    sumw[0] = w[0]

    for i in range(1,length):
        sumw[i] = sumw[i-1] + w[i]

    return sumw
-----------------------------------------
```

In order to compute the sum of elements from $w[i]$ to $w[j]$ in $O(1)$ additional time for any $i < j$, we take the computed running sum array sumw, and indices i and j as input.

Then if i is 0, it means we need to compute the sum from the beginning of the list to index j, which is simply sumw[j].

Otherwise, it computes the sum between indices i and j as sumw[j] - sumw[i - 1], leveraging the precomputed running sum values.

This approach allows to efficiently compute the sum of elements from w[i] to w[j] in constant time complexity O(1), using the precomputed running sum array.

(b) (15 points) *After preprocessing*, design an $O(\log n)$ divide and conquer algorithm that can generate a random sample from the indices of w with each index proportional to weight $w[x]$.

There are two or three different ways to do this, but your idea should behave quite a bit like binary search.

Solution:
>Preprocessing - Computing the Running Sum Array:
We preprocess the weights array w to compute the running sum array sumw. This array contains cumulative sums of the weights such that sumw[i] represents the sum of weights from index 0 to index i in w.
We get this from part (a) of this problem.

>Random Number Generation:
When we generate a random number RandNum between 0 and sumw[-1], we are essentially selecting a point uniformly at random within the entire range of cumulative sums. Since the length of each subinterval in the range [0, sumw[-1]-1] is proportional to the weight of the corresponding index in w, the probability of RandNum falling within a specific subinterval is proportional to the weight of that index in w.

>Binary Search-Like Process:
We use a binary search-like process to find the index x such that sumw[x] <= RandNum < sumw[x+1]. This step is important because it ensures that the random number RandNum falls within the range of cumulative sums associated with a specific index in w.
By checking if RandNum falls within the range [sumw[mid], sumw[mid+1]), where mid is the middle index during each iteration, we effectively narrow down the search space to the subinterval that contains RandNum.
This process is logarithmic in time complexity O(log n) because, at each step, we halve the search space by comparing RandNum with the middle element of the current range

>Proportional Sampling:
Once we find the index x, we return it as the randomly sampled index from w. This index is selected proportionally to its weight w[x], as indices with higher weights have larger cumulative sums and thus a higher probability of being selected.

Here is the code:

```
import random
def generate_random_index(w,sumw):
    RandNum = random.randint(0,sumw[-1]-1)
    left = 0
    right = len(sumw)-1 #from index 0 to index 4 (5-1=4)

    while left < right:
        mid = left + (right - left) // 2
        if RandNum >= sumw[mid]:
            left = mid + 1
        else:
            right = mid
    return left
```

4. (20 points) A common operation in arithmetic is exponentiation, which is NOT a constant time operation like 64-bit addition and multiplication.

   Given 64 bit inputs a, k, n, the goal is to compute $a^k \mod n$. (a^k for 64 bit integers a, k is far too large to fit in 64 bits)

   You can multiply any two 64 bit integers in constant time and do 64 bit modular arithmetic in constant time as well.

   You may assume that a and n are both less than 2^{32}.

   Also, note that for any integers x,y, $xy\%n = (x\%n)(y\%n)\%n$.

   (a) (5 points) Describe a naive $O(k)$ algorithm that can compute $a^k \mod n$

   Solution:
   Pseudocode:

```
function computeExponentMod(a, k, n):
    result = 1
    for i from 1 to k:
        result = (result * a) % n
    return result
```

   This algorithm essentially computes a^k in $O(k)$ time by iteratively multiplying a with itself k times and taking the modulo n after each multiplication to prevent overflow. This approach is not efficient for large values of k, as it requires $O(k)$ multiplications and modulo operations.

   (b) (15 points) Give a more efficient $O(\log k)$ algorithm using divide and conquer that can do this. IMPLEMENT THIS as a function in code, in which ever language you prefer. Test your implementation on the following:

   input: $a = 1234567890, k = 1234567890, n = 987654321$

   answer: 385198425

   You can't use any library functions for this : only +,-,*,/,%. Submit your code as hw7p4.extension, using whatever extension is appropriate for the language you are writing in.

   Hint: Think about what recurrence relation you will need to achieve an $O(\log k)$ run time. Make sure that your solution fits that recurrence relation.

   Hint: You may want to consider two cases, when k is even and when k is odd.

   Solution:

```
def exponentiation(a, k, n):
    if k == 0:
        return 1
    elif k % 2 == 0:
        half_power = exponentiation(a, k // 2, n)
        return (half_power * half_power) % n
    else:
        half_power = exponentiation(a, (k - 1) // 2, n)
        return (a * half_power * half_power) % n

Runtime: O(log k)
```

5. (40 points) You're given an array of n numbers. A *hill* in this array is an element $A[i]$ that is at least as large as its neighbors on either side. In other words, $A[i] \geq A[i-1]$ and $A[i] \geq A[i+1]$. (for the boundaries, the first position $i = 0$ is a hill if $a[0] \geq a[1]$, resp. the last position $i = n-1$ is a hill if $a[n-1] \geq a[n-2]$.) Your goal is to return *some* hill in the array - not every hill of the array.

   (a) Prove that every array must have at least one hill, somewhere in the array.

   Solution:
   Suppose we have an array with numbers ranging from 0 to i arranged with some order (order doesn't matter). Because we have the biggest number i, regardless of where it is positioned in the array it will always be greater than its neighboring numbers, and therefore, i will be a hill. Otherwise, i is not the biggest number.

   (b) Give a naive brute force algorithm that can find a hill in $O(n)$ time. Explain your algorithm in words and analyze its running time.

   Solution:

```
def hill(A):
  # If array has only one element
  if len(A) == 1:
      return A[0]

  # Checking if the first element is a hill
  if A[0] >= A[1]:
      return A[0]

  # pointers to track previous and next item in array
  prv = 0
  nxt = 2
  for curr in range(1, len(A)-1):
      # finding the hill
      if A[prv] <= A[curr] and A[nxt] <= A[curr]:
          return A[curr]
      else:
          prv += 1
          nxt += 1
```

```
        # If there is no hill found at the end of the for loop
        # then the last element must be a hill
        return A[len(A)-1]
```

Base Case:
If the array has only one element, it's the hill. So, the function returns that single element.

The algorithm then initializes two pointers, *prv* (previous) and *nxt* (next), to track elements adjacent to the current element.
It then iterates through the array from the second element to the second last element. At each iteration:

- It checks if the current element is greater than or equal to both its previous and next elements. If so, it's considered a hill, and that element is returned.
- If not, the pointers are incremented to move to the next set of adjacent elements.

(c) Give a divide and conquer algorithm, with proof of correctness and analysis of running time, that will return *some* hill in the array in $\log(n)$ time. Note: it doesn't need to find *every* hill!

Solution:

(d) Similarly, a $k - hill$ is an element that is at least as large as its k neighbors on each side. By this definition, a *hill* is just a $1 - hill$. Modify the above divide and conquer algorithm so that it will find a $k - hill$. Analyze its running time, in terms of n and k.

Solution:

```
def kHillRecursion(A, left, right, k):
  if k == 1:
      hillRecursion(A, left, right)

  if left == right:
      return A[left]

  mid = (left + right) // 2

  # checking if middle element is a hill
  if hillChecking(A, mid, k):
      return A[mid]

  # UPDATED to look at mid - k
  if A[mid-k] > A[mid+k]:
      return kHillRecursion(A, left, mid-1, k)
  else:
```

```
            return kHillRecursion(A, mid+1, right, k)

    # Support function
    def hillChecking(A, x, k):
        for i in range(1, k+1):
            if (A[max(0, x-i)] > A[x] and x-i >= 0
                or A[min(len(A)-1, x+i)] > A[x] and x+i < len(A)):
                return False
        return True
```

Runtime:
$O(k \log(n))$ - Because each recursion step the array is cut in half, and each time the k amount of work is being done.

6. (10 points) (Bonus) Repeat the hill problem, but for a $2-d$ $n \times n$ grid. Here, a hill is a 2-d index (x, y) that needs to be at least as large as its four neighbors above, below, left, and right of that entry, if they exist. The naive approach would take $O(n^2)$ time.

   Goal: Find an O(n) algorithm, and justify its correctness.

   Solution: