Fall 2019 B503 Homework 3 Solutions

Due date: TBD Lecturer: Qin Zhang

Problem 1 (10 points). Solve the following recursions using "Unrolling" and "Guess+Verify".

(a) $T(n) = 2T(\frac{n}{2}) + c$, with boundary condition $T(n) = 1$ for all $n \le 1$.

(b) $T(n) = 2T(\frac{n}{2}) + c\sqrt{n}$, with boundary condition $T(n) = 1$ for all $n \le 1$.

(c) $T(n) = 2T(\frac{n}{2}) + cn^2$, with boundary condition $T(n) = 1$ for all $n \le 1$.

Solution:

(a) $O(n)$

Unrolling:

$$\begin{aligned} T(n) &= c + 2T(n/2) = c + 2c + 4T(n/4) = c + 2c + 4c + 8T(n/8) = \dots \\ &= \sum_{i=0}^{\log n} 2^i c = O(n). \end{aligned}$$

Guess+Verify: Guess $T(n) \le (c+1)n - c$ works for all $n \le k$, then

$$\begin{aligned} T(k+1) = 2T\left(\frac{k+1}{2}\right) + c &\le 2\left((c+1)\frac{k+1}{2} - c\right) + c \\ &= (c+1)(k+1) - c. \end{aligned}$$

The inequality is also true for $n = k + 1$.

(b) $O(n)$

Unrolling:

$$\begin{aligned} T(n) &= c\sqrt{n} + 2T(n/2) = c\sqrt{n} + 2c\sqrt{n/2} + 4T(n/4) = c\sqrt{n} + 2c\sqrt{n/2} + 4c\sqrt{n/4} + 8T(n/8) \\ &= \dots = \sum_{i=0}^{\log n} \sqrt{2^i} c\sqrt{n} = \frac{\sqrt{2^{\log n}} - 1}{\sqrt{2} - 1} c\sqrt{n} = O(n). \end{aligned}$$

Guess+Verify: Guess $T(n) \le (c+1)n - c\sqrt{n}$ works for all $n \le k$, then

$$\begin{aligned} T(k+1) = 2T\left(\frac{k+1}{2}\right) + c\sqrt{k+1} &\le 2\left((c+1)\frac{k+1}{2} - c\sqrt{k+1}\right) + c\sqrt{k+1} \\ &= (c+1)(k+1) - c\sqrt{k+1}. \end{aligned}$$

The inequality is also true for $n = k + 1$.

(c) $O(n^2)$

Unrolling:

$T(n) = cn^2 + 2T(n/2) = cn^2 + 2c(n/2)^2 + 4T(n/4) = cn^2 + 2c(n/2)^2 + 4c(n/4)^2 + 8T(n/8)$

$$= \cdots = \sum_{i=0}^{\log n} \frac{1}{2^i} cn^2 = O(n^2).$$

Guess+Verify: Guess $T(n) \leq 2cn^2$ works for all $n \leq k$, then

$$T(k+1) = 2T\left(\frac{k+1}{2}\right) + c(k+1)^2 \leq 2\left(2c\left(\frac{k+1}{2}\right)^2\right) + c(k+1)^2$$
$$= 2c(k+1)^2.$$

The inequality is also true for $n = k+1$.

Problem 2 (10 points). An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the jth bit of $A[i]$". Using only this operation to access A, give an algorithm that determines the missing integer by looking at only $O(N)$ bits. (Note that there are $O(NlogN)$ bits total in A, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Solution: If integers from 0 to N are represented in binary format, we could easily compute how many 0s and $1s$ should be in each bit.

If we pick a random bit and count how many 0s and $1s$ are actually presented in array A, we could easily tell whether we are missing a 0 or 1. Then we could narrow our search space by half.

So our algorithm works in $\log N$ rounds. In each round, we pick a new bit and do count-and-narrow process. We will reconstruct the missing element by theses missing 0s and 1s.

The number of bits we look at is $N + N/2 + N/4 + \ldots \leq 2N = O(N)$.

Problem 3 (10 points). Suppose now that you're given an $n \times n$ grid graph G. (An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chess board. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, ℓ) are joined by an edge if and only if $|i - k| + |j - \ell| = 1$.)

Each node v is labeled by a real number x_v; you may assume that all these labels are distinct. A node v of G is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

You are given such a graph G, but the labeling is only specified in the following *implicit* way: for each node v, you can determine the value x_v by *probing* the node v.

Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 nodes.)

Solution: We will apply divide and conquer technique to narrow the search space by 1/4 each round.

We first look at center row $n/2$ and center column $n/2$ to find the minimum value x_v. If x_v is a local minimum, we could return. Otherwise, there must exist a value x_u that is smaller than x_v and is not on center row or column. Then we search the quarter where x_u locates. Repeat the process until we find a local minimum.

It is easy to prove that there must be a local minimum in the quarter where x_u locates.

The running time is $T(n) = T(n/2) + O(n) = O(n)$ or $T(n^2) = T(n^2/4) + O(n) = O(n)$.

Problem 4 (10 points). An array $A[1, \ldots, n]$ is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element.

The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] > A[j]$?". (Think of the array elements as GIF files, say.) However you can answer questions of the form: "is $A[i] = A[j]$?" in constant time.

(a) Show how to solve this problem in $O(n \log n)$ time.

(b) Can you give a linear-time algorithm?

Solution:

(a) We could the split array A into two arrays A_1 and A_2 of half the size and find the majority element in each of them. The two subproblem will return at most two values and we could do a linear scan to see whether the return values contain a true majority in original array A. The running time is $T(n) = 2T(n/2) + n = O(n \log n)$. We only need to prove that the true majority value has to be in one of the returned values.

This is easy to prove. If the majority element has size $m > \frac{n}{2}$, there will be at least $m/2 > n/4$ elements in array A_1 or A_2, making it a majority element in sub array.

(b) Another way we would do is to pair up the elements of A arbitrarily, to get $n/2$ pairs. Then we look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them. We repeat the procedure until we have a constant number of elements left and we will return the majority element in it in constant time. A $O(n)$ linear scan will tell us if the return element is a majority element. The running time is dominated by the comparisons procedure $T(n) = T(n/2) + n = O(n)$. We only need to prove that the true majority element stays majority in the new array.

If the majority element has size $m > n/2$ and after comparison, there are l of them left ($l > 0$). Then we know that $2l$ majority values are paired up and $(m - 2l)$ of them are paired with other values. The size of the new array A_{sub} is at most

$$\frac{n - 2l - 2(m - 2l)}{2} + l = (n/2 - m + l) + l$$

where $\frac{n-2l-2(m-2l)}{2} = (n/2 - m + l)$ is the maximum number of non-majority values in A_{sub}, which is still smaller than l. So the majority element stays majority in the new sub array.

Problem 5 (10 points). The *Hadamard matrices* $H_0, H_1, H_2, \ldots$ are defined as follows:

1. H_0 is the 1×1 matrix $[1]$

2. For $k > 0$, H_k is the $2^k \times 2^k$ matrix

$$H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$$

Show that if v is a column vector of length $n = 2^k$, then the matrix-vector product $H_k v$ can be calculated using $O(nlogn)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take constant time.

Solution: Let's rewrite v as $\begin{bmatrix} x \\ y \end{bmatrix}$ such that $x = v[1, 2, \ldots, n/2]$ and $y = v[n/2 + 1, n/2 + 2, \ldots, n]$. Then $H_k v$ can be transformed into the following format:

$$H_k v = \begin{bmatrix} H_{k-1}x + H_{k-1}y \\ H_{k-1}x - H_{k-1}y \end{bmatrix}$$

So we could solve a size n problem by solving two $n/2$ problems and merge then in $O(n)$ operations, giving us the following recursion:

$$T(n) = 2T(n/2) + n = O(n \log n)$$

Problem 6 (10 points). You've been working with some physicists who need to study, as part of their experimental design, the interactions among large numbers of very small charged particles. Basically, their setup works as follows. They have an inert lattice structure, and they use this for placing charged particles at regular spacing along a straight line. Thus we can model their structure as consisting of the points $\{1, 2, 3, \ldots, n\}$ on the real line; and at each of these points j, they have a particle with charge q_j. (Each charge can be either positive or negative.)

They want to study the total force on each particle, by measuring it and then comparing it to a computational prediction. This computational part is where they need your help.

The total net force on particle j, by Coulomb's Law, is equal to

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}.$$

They've written the following simple program to compute F_j for all j:

```
for $j = 1, 2, \ldots, n$ do
    Initialize $F_j$ to 0
    for $j = 1, 2, \ldots, n$ do
        if $i < j$ then
            Add $\frac{Cq_iq_j}{(j-i)^2}$ to $F_j$
        else if $i > j$ then
            Add $-\frac{Cq_iq_j}{(j-i)^2}$ to $F_j$
        end if
    end for
    Output $F_j$
end for
```

It's not hard to analyze the running time of this program: each invocation of the inner loop, over i, takes $O(n)$ time, and this inner loop is invoked $O(n)$ times total, so the overall running time is $O(n^2)$.

The trouble is, for the large values of n they're working with, the program takes several minutes to run. On the other hand, their experimental setup is optimized so that they can throw down n particles, perform the measurements, and be ready to handle n more particles within a few seconds. So they'd really like it if there were a way to compute all the forces F_j much more quickly, so as to keep up with the rate of the experiment.

Help them out by designing an algorithm that computes all the forces F_j in $O(n \log n)$ time.

Solution: This can be accomplished using a convolution. Define one vector to be

$$a = (q_1, q_2, \ldots, q_n)$$

Define the other vector to be

$$b = (-n^{-2}, -(n-1)^{-2}, \ldots, -1/4, -1, 0, 1, 1/4, \ldots, n^{-2})$$

Let $c = a * b$, which is the convolution of a and b. Now, for each j, we can find that the $(j+n)^{\text{th}}$ coordinate of c is equal to

$$\sum_{i<j} \frac{q_i}{(j-i)^2} + \sum_{i>j} \frac{-q_i}{(j-i)^2}.$$

From this term, we simply multiply by Cq_j to get the desired net force F_j.

The convolution can be computed in $O(n \log n)$ time, and reconstructing the terms F_j takes an additional $O(n)$ time. Hence, this algorithm computes all the forces F_j in $O(n \log n)$ time.

Problem 7 (10 points). The problem of greatest common divisor (gcd) of two positive integers is defined as following: the largest integer which divides them both. Euclid's algorithm is a famous one to solve this problem, but we will look at an alternative algorithm based on divide-and-conquer.

(a) Prove that the following rule is true.

$$gcd(a,b) = \begin{cases} 2gcd(a/2,b/2) & \text{if } a,\ b \text{ are even} \\ gcd(a,b/2) & \text{if } a \text{ is odd, } b \text{ is even} \\ gcd((a-b)/2,b) & \text{if } a,\ b \text{ are odd and } a \geq b \end{cases} \tag{1}$$

(b) Give an efficient divide-and-conquer algorithm for greatest common divisor. Analyze the running time of your algorithm if a and b are n-bit integers? (In particular, since n might be large you cannot assume that basic arithmetic operations like addition take constant time.)

Solution:

(a) If a and b are even numbers, 2 is surely a common divisor. Therefore the greatest common divisor will be 2 times the gcd of numbers $a/2$ and $b/2$. If a is odd and b is even, we know for sure that b is divisible by 2 while a is not. Therefore $gcd(a,b)$ remains same as the gcd of a and $b/2$. The third property follows from the fact that $gcd(a,b) = gcd(a-b,b)$ and $(a-b)$ is even when $a \geq b$. We can apply the second property to get the desired result.

(b) We could have the following divide and conquer algorithm $gcd(a,b)$ to solve gcd problem:

```
if $a < b$ then
    Return $gcd(b,a)$
end if
if $a = 0$ or $a = 1$ or $b = 0$ or $b = 1$ or $a = b$ then
    Return accordingly
end if
if $a$ and $b$ are even then
    Return $2gcd(a/2,b/2)$
end if
if $a$ is even then
    Return $gcd(a/2,b)$
end if
if $b$ is even then
    Return $gcd(a,b/2)$
end if
Return $gcd((a-b)/2,b)$
```

Assume that a and b are n-bit numbers, then the problem size is $2n$ bits. Out of the three if conditions in part (a), the problem size is at least reduce to $2n-1$ as divided by 2 is essentially reducing a n bit number to $n-1$ bit. To subtract two n bit numbers, it requires $O(n)$ operation. So we have the recursion of the following:

$$T(2n) = T(2n-1) + n = O(n^2)$$