CS 170 Efficient Algorithms and Intractable Problems
February 2, 2016 Liam Leahy

HW 2

Due February 1, 11:59pm

Partners: Stephen Qu and Sahil Doshi

1. (15 pts.) Complex numbers review *Briefly justify your answers to parts (ii) and (iii).*

(i) Write each of the following numbers in the form $\rho(\cos\theta + i\sin\theta)$ (for real ρ and θ):

(a) $-\sqrt{3}+i$

(b) The three 3-rd roots of unity

(c) The sum of your answers to the previous item

(ii) Let $sqrt(x)$ represent one of the complex square roots of x, so that $sqrt(x) = \pm\sqrt{x}$. What are the possible values of $sqrt(sqrt(-1))$?

You can use any notation for complex numbers, e.g., rectangular, polar, or complex exponential notation.

(iii) Let $A(x) = ax^2+bx+c$ and $B(x) = dx^2+ex+f$. Define $C(x) = A(x)B(x)$. If $A(3) = 7$ and $B(3) = -2$, do you have enough information to determine $C(3)$? If so, what is the value of $C(3)$? If not, explain why not.

Solution:

(i) (a) $2\left(\cos\frac{5\pi}{6} + i\sin\frac{5\pi}{6}\right) = -\sqrt{3}+\frac{i}{2}$.

(b) $1, \cos\frac{2\pi}{3} + i\sin\frac{2\pi}{3} = -\frac{1}{2}+i\frac{\sqrt{3}}{2}, -\frac{1}{2}-i\frac{\sqrt{3}}{2}$.

(c) 0.

(ii) Solving through polar form, you can find that $\sqrt{i}$ is the same as $e^{i\pi/2}$. Thus, the two solutions are $\pm e^{i\pi/2}$. The same goes for $\pm\sqrt{-i}$ but you will get $\pm e^{5i\pi/4}$.

(iii) Yes, it is just $7 \cdot -2 = -14$.

2. (15 pts.) Two sorted lists

(a) **(15 pts.)** You are given two sorted lists, each of size n. Give as efficient an algorithm as possible to find the k-th smallest element in the union of the two lists. What is the running time of your algorithm as a function of k and n?

(b) **(5 pts. of optional extra credit)** Prove your algorithm is optimal.

Solution:

(a) **Main Idea:** To find the kth element of the union of two lists a and b (call the union c). We use a modified binary search to find the index m such that the kth value of c is either the mth value of a or the $(k-m)$th value of b.

Pseudocode:

Algorithm 1 Find kth element of the union of a and b, both arrays of size n.

▷ Note that detail has been omitted concerning remainders and edge cases that are not important for the main idea of the algorithm functioning.

```
1:  function FIND_KTH(a[1,...,n], b[1,...,n])
2:      if SIZEOF(a) == 0 then
3:          return b[k−n]
4:      else if SIZEOF(b) == 0 then
5:          return a[k−n]
6:      else if SIZEOF(a) ≥ k then          ▷ We only need the first k elements of each list.
7:          a ← the first k elements in a
8:      else if SIZEOF(b) ≥ k then
9:          b ← the first k elements in b
10:     a_i ← the (k/2)th element in a
11:     b_j ← the (k/2)th element in b
12:     if a_i < b_j then                   ▷ The mth element of a is beyond i.
13:         a ← the rest of a after a_i
14:         b ← the first k/2 elements of b
15:     else if a_i > b_j then              ▷ The (k−m)th element of b is beyond j.
16:         a ← the first k/2 elements of a
17:         b ← the rest of b after b_j
18:     else if a_i == b_j then
19:         return a_i                      ▷ They're the same value so we choose one arbitrarily to return.
20:     FIND_KTH(a, b)
```

Proof: Let a and b be the two lists of size n and c be the union of a and b. First of all, we only need the first k elements of each set as there is no way that the kth element in c comes after the kth element in either set. Therefore, we start with a problem size of $2k$.

We will obtain the values $a_i = a\left[\frac{k}{2}\right]$ and $b_j = b\left[\frac{k}{2}\right]$ and compare them. We are looking for some m such that the mth element of a is the kth element of c or the $(k-m)$th element of b is the kth element of c. There are definitely $k-1$ values, among a and b, that come before the kth element in c. So, once we add $\frac{k}{2}$ elements from each set, we know that we have added the kth element into c as there are at least k elements in c.

Additionally, we can say that the kth element in c necessarily falls between a_i and b_j, inclusive. This can be shown by working through the construction of c and observing that once *both* a_i and b_j are added, there are invariably k elements in c. This observation lets us narrow down our search to $\frac{k}{2}$

elements in each iteration. We do this by comparing the value of a_i with the value of b_j. Once we know how they compare, we can determine which one would be added into c first during its construction and deduce which direction we should be searching. Take, for example, the case in which a_i is less than b_j. We know, therefore, that the kth element in c comes after this element in a, as k elements have not been added into c at this point. We can eliminate all of the $\frac{k}{2}$ elements in a that come up to this element, reducing the problem size to $\frac{1}{2}$ times the original size. This will continue until there are no elements in a, in which case we just take the $(k-n)$th element of b. Or we reach a base case in which we make our decision based on their relative values. □

Running Time: $\Theta(\log k)$.

Justification: Every iteration, we cut our problem size in half as we eliminate $p \cdot 1/2$ (where p is our current problem size) elements that *all come before* the kth element. We can start by saying that the kth element must be in the first k elements from a and b. Therefore, we simply start with $2k$ elements from both a and b. Following this, our algorithm eliminates $\frac{k}{2}$ elements from each set on the first step, leaving us with half the starting size each iterations. In doing so, we only perform constant time operations (comparison, division, and array slicing) and therefore, in the worst case, we will do this in $\Theta(\log k)$ time as our problem size approaches 1.

(b) There are k values and we are using comparison to find the kth value so we have to, at the least, look at $\log k$ values in order to be able to consider all indices. We know that there are at most k positions that the kth element could be located in. Any algorithm that compares two elements and eliminates some fraction that is not one half of them *could* be slower if the larger side was chosen as the possible side. Since we are using half every time there is no case that it could be worse than $\log k$.

3. (15 pts.) Peak element

Prof. B. Inary just moved to Berkeley and would like to buy a house with a view on Euclid Ave. Needless to say, there are n houses on Euclid Ave, they are arranged in a single row, and have distinct heights. A house "has a view" if it is taller than its neighbors. For example, if the houses heights were $[2,7,1,8]$, then 7 and 8 would "have a view".

Devise an efficient algorithm to help Prof. Inary find a house with a view on Euclid Ave. (If there are multiple such houses, you may return any of them.)

Solution:

Main Idea: Find any one of the peaks in the n houses. We perform a modified binary search using the comparison of the middle house to its immediate neighbors, always choosing the side with a larger house (breaking ties arbitrarily) until we find a peak.

Pseudocode:

Algorithm 2 Find a house with a view on Euclid Ave.

```
1: function FIND_PEAK(S[1,...,n])
2:     s ← the value of S at the middle of S
3:     if the values surrounding s are both less than the value of s then      ▷ s is a peak.
4:         return s
5:     else
6:         o ← the neighbor that is larger than s                              ▷ Ties broken arbitrarily.
7:     return FIND_PEAK(the subset of S on the side of o)
```

Proof: Consider s from the above algorithm. We have to show both that there is a peak to the side that contains the larger neighbor and that we eventually reduce the problem size to 1 (we have found a peak).

Proof: Proof by induction on m, where m is the number of elements to the side of s, in the direction of o (call this subset of s in the direction of o O), that there is a peak to this side of s.

Base Case: There is only one element in O, it has no neighbor on the other side and therefore it must be a peak.

Induction Hypothesis: It is true that there is a peak with m elements in O.

Induction Step: Add one house u anywhere in O. We know that there was a peak p in O before u was added. Therefore, there is still a peak in O if u was not added directly to the side of p. Consider the case that u was added to the side of p. If u is smaller, p is still a peak. If u is larger than p, then it is necessarily larger than p's original neighbor too, and therefore it is a peak in the new set. □

Since there is always a peak in O and we invariably reduce the size of S by a factor of 2 every time, we will eventually find a peak. □

Running Time: $\Theta(\log n)$.

Justification: We start with a problem size of n. Each iteration, regardless of input, we will always eliminate $\frac{n}{2}$ elements from further consideration. Therefore, in the worst case, we will work until the problem size is 1, which will require $\log n$ steps.

4. (20 pts.) Majority Elements

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. (Hint: Split the array A into two arrays A_1 and A_2 of half the size. Does knowing the majority elements of A_1 and A_2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.)

(b) Can you give a linear-time algorithm? (Hint: Here's another divide-and-conquer approach:

- Pair up the elements of A arbitrarily, to get about $n/2$ 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
- If $|A|$ is odd, there will be one unpaired element. What should you do with this element?

Show that after this procedure there are at most $n/2$ elements left, and that they have a majority element if A does.)

Solution:

(a) **Main Idea:** If A has a majority element, then A_1 and or A_2 will also have that element. The majority element of A must be one of: the majority element of A_1, the majority element of A_2, or none. We can linearly check if any one of these is the majority element by just counting the number of times that each occurs in A in linear time.

Pseudocode:

Algorithm 3 Find the majority element of A, if it exists.

```
1:  function FIND_MAJ(A[1, ..., n])
2:      if SIZEOF(A) is equal to 1 then
3:          return the one element in A
4:      else
5:          A_1 ← the first half of A
6:          A_2 ← the second half of A
7:          a ← SET(FIND_MAJ(A_1), FIND_MAJ(A_2))
8:      for all elements e in a do
9:          if e is a majority element of A then
10:             return e
11:     return None
```

Proof: Let k be any element in A that is not the majority element in either A_1 or A_2. The count of this element is therefore less than $\frac{|A|}{2} + 1$ as its count must be less than $\frac{|A|}{4} + 1$ in A_1 and the same for A_2. This means that k is not the majority element in A. Therefore, if A has a majority element it cannot be any k, which implies that it must be one of the majority elements of A_1 or A_2. □

Running Time: $\Theta(n \log n)$.

Justification: We can model the running time as a function $T(n) = 2T(n/2) + \Theta(n)$ as we make two recursive calls on each subset of half the size and we perform, at most, linear time checking to check for majority elements. We know that this means that $T(n) \in \Theta(n \log n)$ from the Master Theorem.

(b) **Main Idea:** We pair up all elements and are left with either one (in the odd number case) or zero. If there is one left, we check if it is the majority and if not, get rid of it. We continually apply this procedure until we are left with one element which is the majority element assuming there is one, or none.

Pseudocode:

Algorithm 4 Find the majority element of A, if it exists.

```
1:  function FIND_MAJ(A[1, ..., n])
2:      B ← A                                          ▷ Save the original A.
3:      if SIZEOF(A) is odd then
4:          a ← one arbitrarily-chosen element popped from A     ▷ The size of A is now even.
5:          if a is the majority element in B then
6:              return a
7:      while SIZEOF(A) > 1 do
8:          pair all elements in A and keep one of each element that is paired with a copy of itself
9:          A ← the remaining elements
10:     if there is one remaining element in A then
11:         a ← the element in A
12:         if a is the majority element in B then
13:             return a
14:     return None
```

Proof: If there is not a majority element in A the final check will simply fail.

First, if $|A|$ is odd, we have to remove one of the elements from A. If this is not the majority element, removing it will not change the results as we are only making it easier for the algorithm to find the majority element.

If there is a majority element in A, let m be this element, let M be the subset of A that is all of m in A, and let N be the subset of all elements in A that are not the majority element. For each pairing, there are only three possible scenarios:

1. An element from N is paired with another element from N. We do not know what will happen to these elements as they could match or not. Regardless, however, now $|M| - |N|$ is two greater than it was before. This implies that *necessarily*, one pair of m will remain in the end such that a corresponding majority element for this pairing will be promoted as well.
2. An element from M will be paired with an element from N. This will result in an elimination as they are not equal and no elements will be promoted. The relative sizes of M and N do not change (m is still a majority element in the reduced set).
3. Two elements from M are paired. This results in a promotion of one element from M.

Following this, we now know that it is impossible for a non-majority element to be promoted without there being a corresponding majority element being promoted. Also, since there is a majority element, $|M| > \frac{|A|}{2}$. This means that there must be at least one majority element promotion. Therefore, we can define Q to be the set of all elements promoted (where $|Q| \geq 1$). Since there is at least one m being promoted, we can simply remove 2 elements from M (and put one m in Q) and focus on the remaining pairs. This means that the size of M is now bounded below by the size of N as M is the majority element and if two are removed then there could be the same number of non-majority and majority elements at a minimum. There are $\frac{|A|}{2} - 1$ pairs and for each one, if there is a promotion and this promotion is a non-majority element, there is also a promotion of a majority element as well. This implies that there is at least one majority element for every non-majority element promoted in Q and

one from the inevitable promotion mentioned above. Therefore, the majority element of A is still the majority element of Q. $\square$

Running Time: $\Theta(n)$.

Justification: We can show that the time complexity can be modeled as $T(n) = T\left(\frac{n}{2}\right) + \Theta(n)$, as at each step, we divide the problem size by 2 and do linear work (pairing and eliminating). We do not recurse on two problems of half the size, so we do not need to multiply $T\left(\frac{n}{2}\right)$ by anything. This is $\Theta(n)$ by the Master Theorem.

5. (20 pts.) Squaring vs multiplying: matrices

The square of a matrix A is its product with itself, AA.

(a) Show that five multiplications are sufficient to compute the square of a 2×2 matrix.

(b) What is wrong with the following algorithm for computing the square of an $n \times n$ matrix?

"Use a divide-and-conquer approach as in Strassen's algorithm, except that instead of getting 7 subproblems of size $n/2$, we now get 5 subproblems of size $n/2$ thanks to part (a). Using the same analysis as in Strassen's algorithm, we can conclude that the algorithm runs in $\Theta(n^{\log_2 5})$ time."

(c) In fact, squaring matrices is no easier than multiplying them. Show that if $n \times n$ matrices can be squared in $\Theta(n^c)$ time, then any $n \times n$ matrices can be multiplied in $\Theta(n^c)$ time.

Solution:

(a) $A^2 = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a^2+bc & ab+bd \\ ac+cd & bc+d^2 \end{bmatrix} = \begin{bmatrix} a^2+bc & b(a+d) \\ c(a+d) & bc+d^2 \end{bmatrix}$. There are 5 multiplications: $a^2, bc, b(a+d), c(a+d), d^2$.

(b) Matrix multiplication does not commute! Therefore, this simplification, that works in the case of polynomials, does not work here and cannot be used.

(c) We can compute

$$A^2 = \begin{bmatrix} XY & 0 \\ 0 & YX \end{bmatrix}.$$

As this implies that the value XY can be done in $O((2n)^c)$ time, which is still in $O(n^c)$.

6. (15 pts.) The Hadamard matrices

The Hadamard matrices $H_0, H_1, H_2, \ldots$ are defined as follows:

- H_0 is the 1×1 matrix $[1]$
- For $k > 0, H_k$ is the $2^k \times 2^k$ matrix

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]$$

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

Solution:

We write

$$H_k \cdot \mathbf{v} = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} \cdot \begin{bmatrix} v_u \\ v_l \end{bmatrix} = \begin{bmatrix} H_{k-1} \cdot v_u + H_{k-1} \cdot v_l \\ H_{k-1} \cdot v_u + -H_{k-1} \cdot v_l \end{bmatrix}$$

where v_u and v_l stand for the upper and lower halves of the vector $\mathbf{v}$, respectively. Additionally, we note that there are only two necessary matrix multiplications (where each matrix is an $\frac{n}{2}$-sized matrix) and we need $O(n)$ additions per level. We can model this as $T(n) = 2T(\frac{n}{2}) + O(n)$. Thus the final running time would be $O(n \log n)$ by the Master Theorem.

7. (5 pts.) Optional bonus problem: More medians

(This is an *optional* challenge problem. It is not the most effective way to raise your grade in the course. Only solve it if you want an extra challenge.)

We saw in class a randomized algorithm for computing the median, where the expected running time was $O(n)$. Design a algorithm for computing the median where the *worst-case* running time is $O(n)$.

Hint: It's all in finding a good pivot. If you divide the array into small groups of c elements, can you use that to help you a good pivot?