CS 170 Efficient Algorithms and Intractable Problems
Spring 2016 Alessandro Chiesa and Umesh Vazirani

HW Solution 2

Due February 1, 11:59pm

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

(b) $(\cos 0 + i\sin 0), (\cos\frac{2\pi}{3} + i\sin\frac{2\pi}{3}), (\cos\frac{4\pi}{3} + i\sin\frac{4\pi}{3})$

(c) 0

(ii) $\sqrt{-1} = \pm i$;
$\sqrt{i} = \pm\frac{\sqrt{2}}{2}(1+i)$, $\sqrt{i} = \pm\frac{\sqrt{2}}{2}(1-i)$.

Alternatively, $-1 = \cos\pi + i\sin\pi = \cos 3\pi + i\sin 3\pi$.
So, $\sqrt{\cos\pi + i\sin\pi} = \{(\cos\frac{\pi}{2} + i\sin\frac{\pi}{2}), (\cos\frac{3\pi}{2} + i\sin\frac{3\pi}{2})\}$.
Therefore:
$\sqrt{(\cos\frac{\pi}{2} + i\sin\frac{\pi}{2})} = \sqrt{(\cos\frac{5\pi}{2} + i\sin\frac{5\pi}{2})} = \{(\cos\frac{\pi}{4} + i\sin\frac{\pi}{4}), (\cos\frac{5\pi}{4} + i\sin\frac{5\pi}{4})\}$, and
$\sqrt{(\cos\frac{3\pi}{2} + i\sin\frac{3\pi}{2})} = \sqrt{(\cos\frac{7\pi}{2} + i\sin\frac{7\pi}{2})} = \{(\cos\frac{3\pi}{4} + i\sin\frac{3\pi}{4}), (\cos\frac{7\pi}{4} + i\sin\frac{7\pi}{4})\}$.

(iii) $C(3) = A(3)B(3) = (7)(-2) = -14$.

2. (15 pts.) Two sorted arrays

(a) **(15 pts.)** You are given two sorted arrays, each of size n. Give as efficient an algorithm as possible to find the k-th smallest element in the union of the two arrays. 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 in the comparisons model.

Solution:

(a) One way to approach this problem was to notice that, since the answer must be among the k smallest elements of the two arrays, we can restrict attention to the first k elements of each of the two arrays, and then the answer is the median of the two restricted arrays. Then we can just use the divide and conquer algorithm from lecture finds the median of these two sorted arrays in $O(\log k)$ steps.

If you solved the problem in this way, by explaining how to solve it with our median-finder from lecture (ie, *reducing* it to median-finding problem), and justified the runtime, that solution gets full credit (indeed, it's cleaner than redoing the problem from scratch).

Note that this is an example of a reduction, a very important concept that we will encounter repeatedly in this course. We showed how to use an existing program to solve a seemingly new problem.

Alternatively, you could have written out a full solution for this algorithm, which follows:

Main idea We generalize from the algorithm presented in lecture for the median of the union of two arrays. The algorithm will be a divide and conquer algorithm (without much conquering) which will eliminate half the candidate elements in each step. Each step takes constant time, so we can achieve a $\Theta(\log k)$ runtime.

Pseudocode

procedure TWOARRAYSELECTION($a[1..n]$, $b[1..n]$, element rank k)
  Constrain a and b to have length k:
    If $n > k$, chop off the last $(n-k)$ elements of each array
    If $n < k$, add $k-n$ infinite-valued elements to the end of each array.
    (If $2n < k$ so that there is no k-th smallest element, throw an exception.)
  while $a[\lfloor k/2 \rfloor] \neq b[\lceil k/2 \rceil]$ and $k > 1$ **do**
    if $a[\lfloor k/2 \rfloor] > b[\lceil k/2 \rceil]$ **then**
      Set $a := a[1, \cdots, \lfloor k/2 \rfloor]$; $b := b[\lceil k/2 \rceil + 1, \cdots, k]$; k := $\lfloor k/2 \rfloor$
    else
      Set $a := a[\lfloor k/2 \rfloor + 1, \cdots, k]$; $b := b[1, \cdots, \lceil k/2 \rceil]$; $k := \lceil k/2 \rceil$
  if $k = 1$ **then**
    return min($a[1]$, $b[1]$)
  else
    return $a[\lfloor k/2 \rfloor]$

Proof of correctness

Our algorithm starts off by comparing elements $a[\lfloor k/2 \rfloor]$ and $b[\lceil k/2 \rceil]$. Suppose $a[\lfloor k/2 \rfloor] > b[\lceil k/2 \rceil]$. Then, in the union of a and b there can be at most $k-2$ elements smaller than $b[\lceil k/2 \rceil]$, i.e. $a[1, \cdots, \lfloor k/2 \rfloor - 1]$ and $b[1, \cdots, \lceil k/2 \rceil - 1]$, and we must necessarily have $s_k > b[\lceil k/2 \rceil]$. Similarly, all elements $a[1, \cdots, \lfloor k/2 \rfloor]$ and $b[1, \cdots, \lceil k/2 \rceil]$ will be smaller than $a[\lfloor k/2 \rfloor + 1]$; but these are k elements, so we must have $s_k < a[\lfloor k/2 \rfloor + 1]$.

This shows that s_k must be contained in the union of the subarrays $a[1, \cdots, \lfloor k/2 \rfloor]$ and $b[\lceil k/2 \rceil + 1, \cdots, k]$. In particular, because we discarded $\lceil k/2 \rceil$ elements smaller than s_k, s_k will be the $\lfloor k/2 \rfloor$th

smallest element in this union.

We can then find s_k by recursing on this smaller problem. The case for $a[\lfloor k/2 \rfloor] < b[\lceil k/2 \rceil]$ is symmetric.

If we reach $k = 1$ before $a[\lfloor k/2 \rfloor] = b[\lceil k/2 \rceil]$, we can cut the recursion a little short and just return the minimum element between a and b. You can make the algorithm work without this check, but it might get clunkier to think about the base cases.

Alternatively, if we reach a point where $a[\lfloor k/2 \rfloor] = b[\lceil k/2 \rceil]$, then there are exactly k greater elements, so we have $s_k = a[\lfloor k/2 \rfloor] = b[\lceil k/2 \rceil]$.

Running time analysis

At every step we halve the number of elements we consider, so the algorithm will terminate in $\log(2k)$ recursive calls. Assuming the comparison takes constant time, the algorithm runs in time $\Theta(\log k)$. Note that this does not depend on n. (If n is too small, we throw an exception, but the algorithm doesn't take longer as n grows.)

(b) We prove a lower bound in the comparison model as follows: each of the k smallest elements in the first array can be the answer, depending on the k smallest values in the second array. The pairs of arrays for each of these k possibilities must end up at different leaves in the comparison tree. Therefore there are at least k different leaves in this binary tree, and so it has depth at least $\log k$, meaning that the algorithm must make at least $\log k$ comparisons.

3. (20 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:

(a) **Main idea** Divide and conquer. At each step, test if the middle house has a view. If not, at least one of its neighbors must be taller. Recurse on a side with a taller neighbor.

(b) **Psuedocode**

procedure PEAK($a[1..n]$)
  if $n \leq 3$ **then** ▷ base case
    Return index of max element.

  $m \leftarrow \lceil n/2 \rceil$ ▷ divide and conquer
  if $a[m] > a[m+1]$ AND $a[m] > a[m-1]$ **then**
    Return m.
  else if $a[m] < a[m-1]$ **then**
    Return PEAK($a[1..m-1]$).
  else
    Return $m +$ PEAK($a[m+1..n]$).

(c) **Proof of correctness** When $a[m]$ has a view, the algorithm outputs it. If $a[m]$ does not, suppose we are in the case that the algorithm chooses to search $A[1..m-1]$ (symmetric argument for the other case). Now, imagine if we start at $a[m-1]$, and continue searching to the left until we either reach $a[1]$ or a house that has a smaller neighbor to its left. Either way, this house must have a view. Thus, there must be a house with a view in this half of the street. Furthermore, exactly one end house of this half of the street is not an end house of the original street, that is $a[m-1]$, but we know $a[m-1] > a[m]$, so if $A[m-1]$ has a view with respect to this half, it has a view in the original street.

(d) **Running time analysis** The running time is $\Theta(\log n)$, since at each level we solve **one** problem of size $\frac{n}{2}$, plus constant work (comparing the middle elements to neighbors). $T(n) = T(\frac{n}{2}) + \Theta(1)$, and so $T(n) = \Theta(\log n)$ by the Master Theorem.

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) (i) **Main idea** Divide and conquer. At each step, partition into two arrays of equal size. Notice that if A has a majority element v, v must also be a majority element of A_1 or A_2 or both.

(ii) **Psuedocode**

```
procedure MAJORITY(A[1..n])
    if n = 1 then                                          ▷ base case
        Return A[1].

    m ← ⌈n/2⌉                                              ▷ divide and conquer
    v1 ← MAJORITY(A[1..m])
    v2 ← MAJORITY(A[m+1..n])

    c1, c2 ← 0                                             ▷ are v1, v2 majority of entire array?
    for i ∈ 1..n do
        if A[i] == v1 then c1++.
        if A[i] == v2 then c2++.
    if c1 > n/2 then
        Return v1.
    else if c2 > n/2 then
        Return v2.
    else
        Return ⊥                                           ▷ did not find a majority element
```

(iii) **Proof of correctness** By induction:

Induction hypothesis The algorithm is correct for smaller inputs.

Base Case If $n = 1$, the array contains exactly one element, and we always return it.

Induction step Suppose v appears more than $n/2$ times in the array. Then after we partition into sub-arrays of sizes $n1, n2$, either the first array contains more than $n1/2$ copies, or the second

array contains more than $n2/2$ copies. Either way, v is a majority element of at least one of the sub-arrays, and -by our induction hypothesis- will be returned by the recursive call to the algorithm.

(iv) **Running time analysis** Two calls to problems of size $n/2$, and then linear time to test $v1, v2$: $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

b) (i) **Main idea** Removing two different items does not change the majority element (but may create spurious solutions!). We recursively pair up the elements and remove all pairs with different elements. For every pair with identical elements, it suffices to keep just one (since we do so for all pairs, we again don't change the majority). If n is odd, brute-force test in linear time whether the unpaired element is a majority element.

(ii) **Psuedocode**

```
procedure MAJORITY-PAIRS(A[1..n])
    if n = 1 then                                          ▷ base case
        Return A[1].

    if n is odd then
        c ← 0                                              ▷ is A[n] a majority element?
        for i ∈ 1..n − 1 do
            if A[i] == A[n] then c++.
        if c ≥ (n − 1)/2 then
            Return A[n].
        else
            Return MAJORITY-PAIRS(A[1..n − 1])             ▷ A[n] is not a majority element

    else                                                   ▷ even n
        j ← 1
        for i ∈ 1..n/2 do
            if A[2i] == A[2i + 1] then
                B[j] ← A[2i]
                j++
        v ← MAJORITY-PAIRS(B[1.. j − 1])
        c ← 0                                              ▷ is v really a majority element?
        for i ∈ 1..n do
            if A[i] == v then c++.
        if c > n/2 then
            Return v.
        else
            Return ⊥                                       ▷ there is no majority element
```

(iii) **Proof of correctness** By induction:

Induction hypothesis The algorithm is correct for smaller inputs.

Base Case If $n = 1$, the array contains exactly one element, and we always return it.

Induction step If n is odd, the brute-force test for $A[n]$ is straightforward.

Assume wlog that n is even. We must show that if A has a majority element, then it is also a majority element in B; then we'd be done by the induction hypothesis. Consider a modified algorithm which only removes all the pairs with different elements; let's call the resulting array B'. Suppose v appears in A more often than all other elements combined. Whenever

we remove a pair that contains v and another element, this property continues to hold. When we remove a pair that contains two other elements, v definitely remains a majority element. Either way, after each pair removal v remains a majority element. Therefore v (by another induction) is also a majority element in B'. Now from B' to B we do not change the proportions of elements, so the majority elements remains.

(iv) **Running time analysis** One call to a problem of size $\leq n/2$, as well as additional linear time to test each candidate: $T(n) = T(\frac{n}{2}) + O(n) = O(n)$.

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)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^2 = \begin{bmatrix} a^2+bc & b(a+d) \\ c(a+d) & bc+d^2 \end{bmatrix}$$

Hence the 5 multiplications $a^2, d^2, bc, b(a+d)$ and $c(a+d)$ suffice to compute the square.

b) Matrices don't commute! That is, in general $BC \neq CB$, so if our $n/2 \times n/2$ submatrices are A, B, C, D, we actually have:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^2 = \begin{bmatrix} A^2+BC & AB+BD \\ CA+DC & CB+D^2 \end{bmatrix} \neq \begin{bmatrix} A^2+BC & B(A+D) \\ C(A+D) & BC+D^2 \end{bmatrix}.$$

(Also, even if we were lucky and our submatrices did commute, we would get 5 subproblems that are *not of the same type as the original problem*: We started with a squaring problem for a matrix of size $n \times n$ and three of the 5 subproblems now involve *multiplying* $n/2 \times n/2$ matrices. Hence the recurrence $T(n) = 5T(n/2) + O(n^2)$ does not make sense.)

c) Given two $n \times n$ matrices X and Y, create the $2n \times 2n$ matrix A:

$$A = \begin{bmatrix} 0 & X \\ Y & 0 \end{bmatrix}$$

It now suffices to compute A^2, as its upper left block will contain XY:

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

Hence, the product XY can be calculated in time $O(S(2n))$. If $S(n) = O(n^c)$, this is also $O(n^c)$.

Note: This is an example of a reduction, and is an important concept that we will see over and over again in this course. We are saying that matrix squaring is no easier than matrix multiplication — because we can trick any program for matrix squaring to actually solve the more general problem of matrix multiplication.

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 v is a column vector of length $n = 2^k$, then the matrix-vector product $H_k v$ 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:

For any column vector u of length n, let $u^{(1)}$ denote the column vector of length $n/2$ consisting of the first $n/2$ coordinates of u. Similarly, define $u^{(2)}$ to be the vector of the remaining coordinates. Note then that $(H_k v)^{(1)} = H_{k-1} v^{(1)} + H_{k-1} v^{(2)}$ and $(H_k v)^{(2)} = H_{k-1} v^{(1)} - H_{k-1} v^{(2)}$. This shows that we can find $H_k v$ by recursively computing $x = H_{k-1} v^{(1)}$ and $y = H_{k-1} v^{(2)}$, and setting $(H_k v)^{(1)} = x + y$ and $(H_k v)^{(2)} = x - y$. The running time of this algorithm is given by the recursion $T(n) = 2T(\frac{n}{2}) + O(n)$, where the linear term is the time taken to perform the two sums. This has solution $T(n) = O(n \log n)$ by the Master theorem.

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

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?

Solution:

We will design a recursive algorithm that finds the kth element of the input array. For simplicity, let c be odd. Split the array arbitrarily into lists of c elements. Find the medians of the lists. Recursively compute the median of this list of medians. Use that element as a pivot. We will get a linear time algorithm if $c \geq 5$.

Pseudocode:

procedure ARRAYSELECT(Array A, element rank k)
  if $|A| < c^2$ **then** Output the kth element of A using brute force
  Let M be a c by $\lceil |A|/c \rceil$ matrix, where A is grouped into rows with c elements from A arbitrarily.
  Let N be the matrix M with each row sorted.
  Let B be the array of medians of rows of M.
  Let a = ArraySelect(B, $\lfloor |B|/2 \rfloor$).
  Let i be the number of elements in A that are less than or equal to a.
  if $k \leq i$ **then** Output ArraySelect(all elements of A less than a, k)
  elseOutput ArraySelect(all elements of A greater than a, $k - i$)

Runtime analysis:

Let $T(n)$ be the runtime. There are two calls to ArraySelect and a linear amount of additional work. The first call to ArraySelect is on an array with size at most n/c. The other call to ArraySelect involves an array with size at most $3n/4$. This occurs for the following reason. Imagine breaking up N into quadrants using a as follows. Sort the rows of N by their medians to obtain a new matrix P. a is the center element of the

matrix P. One quadrant only contains elements greater than a and another only contains elements less than a. The other quadrants cannot be reasoned about in general. Nonetheless, this shows that the array in the second call to ArraySelect will contain no elements from some quadrant.

This means that

$$T(n) \leq T(3n/4) + T(n/c) + O(n)$$

Setting $c \geq 5$ ensures that $T(n) \leq O(n)$ by the Master Theorem. Making $|A| > c^2$ ensures that the algorithm will always recur on strictly smaller arrays.

Proof of correctness:

We can prove this inductively as follows. The base case follows from the first if statement. To prove the inductive step, we just need to notice that if $k > i$, the kth element of A is the $k-i$th element of the recurred array.