CS 170 Efficient Algorithms and Intractable Problems
Fall 2017 Prasad Raghavendra and Sanjam Garg

Homework 1

Instructions: You are welcome to form small groups (up to 4 people total) to work through the homework, but you **must** write up all solutions by yourself. List your study partners for homework on the first page, or “none” if you had no partners.

If using LaTeX (which we recommend), you may use the homework template linked on this Piazza post to get started.

Begin each problem on a new page. Clearly label where each problem and subproblem begin. The problems must be submitted in order (all of P1 must be before P2, etc). For questions asking you to give an algorithm, respond in what we will refer to as the *four-part format* for algorithms: main idea, pseudocode, proof of correctness, and running time analysis.

Read the Homework FAQ Piazza post on Piazza before doing the homework for more explanation on the four-part format and other clarifications for our homework expectations.

No late homeworks will be accepted. No exceptions. This is not out of a desire to be harsh, but rather out of fairness to all students in this large course. Out of a total of approximately 12 homework assignments, the lowest two scores will be dropped.

Special Questions:

- *Shortcut questions*: Short questions are usually easy questions that give you opportunities to practice basic materials. However, we understand that some of you are very familiar with the topics and do not want to spend too much time on easy questions. Therefore, we design shortcut questions for this purpose. A shortcut question usually has multiple parts that build upon each other and are ordered by their difficulty level. You can work on those in order or start from wherever you like. However you only need to submit the last part you are able to solve. For example, if a question has 5 parts (a, b, c, d, e), you are confident about part e, you should submit part e without any of the previous four parts. If you are confident about d but not sure about e, you should submit d for grading purposes. Please clearly indicate in your submission which part you are submitting.
- *Redemption questions*: It is important that you carefully read the posted solutions, even for problems you got right. To encourage this, you have the option of submitting a redemption file, a few paragraphs in which you explain, for each problem you choose to cover, what you did wrong and what the right idea was in your own words (not cutting and pasting from the solution!), and appending it to your homework. For example, suppose that as you review your solutions to HW1, you realize you had misunderstood question 3 and answered it incorrectly. You would write down what you just learned, and then submit it in your HW2 assignment the following week. Because these are mainly for your benefit, feel free to format them however is most useful for you.
- *Extra credit questions*: We might have some extra credit questions in the homework for people who really enjoy the materials. However, please note that you should do the extra credit problems only if you really enjoy working on these problems and want an extra challenge. It is likely not the most efficient manner in which to maximize your score.

Due Wednesday, September 6, at 4:59pm

1. (★ level) Course Syllabus

Before you answer any of the following questions, please read over the syllabus carefully. The syllabus is pinned on the Piazza site. For each statement below, write *OK* if it is allowed by the course policies and *Not OK* otherwise.

(a) You ask a friend who took CS 170 previously for her homework solutions, some of which overlap with this semester's problem sets. You look at her solutions, then later write them down in your own words.

Solution: Not OK.

(b) You had 5 midterms on the same day and are behind on your homework. You decide to ask your classmate, who's already done the homework, for help. He tells you how to do the first three problems.

Solution: Not OK.

(c) You look up a problem online to search an algorithm, write it in your words and cite the source. **Solution:** Not OK.

(d) You were looking up Dijkstra's on the internet, and run into a website with a problem very similar to one on your homework. You read it, including the solution, and then you close the website, write up your solution, and cite the website URL in your homework writeup.

Solution: OK. Given that you'd inadvertently found a resource online, clearly cite it and make sure you write your answer from scratch.

(e) You are working on the homework in one of the TA's office hours with other people. You hear that student John Doe asked the TA if his solution is correct or not and the TA is explaining it. You join their conversation to understand what John has done.

Solution: Not OK.

2. (★★ level) Asymptotic Complexity Comparisons

(a) Order the following functions so that $f_i \in O(f_j) \iff i \leq j$. Do not justify your answers.

(i) $f_1(n) = 3^n$

(ii) $f_2(n) = n^{\frac{1}{3}}$

(iii) $f_3(n) = 12$

(iv) $f_4(n) = 2^{\log_2 n}$

(v) $f_5(n) = \sqrt{n}$

(vi) $f_6(n) = 2^n$

(vii) $f_7(n) = \log_2 n$

(viii) $f_8(n) = 2^{\sqrt{n}}$

(ix) $f_9(n) = n^3$

Solution: $f_3, f_7, f_2, f_5, f_4, f_9, f_8, f_6, f_1$

(b) In each of the following, indicate whether $f = O(g)$, $f = \Omega(g)$, or both (in which case $f = \Theta(g)$). ***Briefly*** justify each of your answers.

	$f(n)$	$g(n)$
(i)	$\log_3 n$	$\log_4 n$
(ii)	$n\log(n^4)$	$n^2\log(n^3)$
(iii)	$\sqrt{n}$	$(\log n)^3$
(iv)	2^n	2^{n+1}
(v)	n	$(\log n)^{\log\log n}$
(vi)	$n+\log n$	$n+(\log n)^2$
(vii)	$\log n!$	$n\log n$

Solution:

(i) $f=\Theta(g)$; using the log change of base formula, $\frac{\log n}{\log 3}$ and $\frac{\log n}{\log 4}$ differ only by a constant factor.

(ii) $f=O(g)$; $f(n)=4n\log(n)$ and $g(n)=3n^2\log(n)$, and the polynomial in g has the higher degree.

(iii) $f=\Omega((\log n)^3)$; any polynomial dominates a product of logs.

(iv) $f=\Theta(g)$; $g(n)=2f(n)$ so they are constant factors of each other.

(v) $f=\Omega(g)$; $n=2^{\log n}$ and $(\log n)^{\log\log n}=2^{(\log\log n)^2}$, so f grows faster than g since $\log n$ grows faster than $(\log\log n)^2$.

(vi) $f=\Theta(g)$; Both f and g grow as $\Theta(n)$ because the linear term dominates the other.

(vii) $f=\Theta(g)$;
Observe that

$$n! = 1*2*3\cdots*n \le n*n*n\cdots*n \le n^n$$

and assuming n is even (without loss of generality)

$$n! = 1*2*3\cdots*n \ge n*(n-1)*(n-2)\cdots*(n-n/2) \ge \left(\frac{n}{2}\right)^{\frac{n}{2}+1}.$$

Hence $\left(\frac{n}{2}\right)^{\frac{n}{2}} \le n! \le n^n$. Then,

$$\frac{n}{2}\log\left(\frac{n}{2}\right) \le \log(n!) \le n\log n.$$

and we conclude that the functions grow at the same asymptotic rate.

(c) Let $f(\cdot)$ be a function. Consider the equality

$$\sum_{i=1}^{n} f(i) \;\in\; \Theta(f(n)),$$

give a function f_1 such that the equality holds, and a function f_2 such that the equality does not hold.

Solution: There are many possible solutions.
$f_1(i)=2^i$: $\sum_{i=1}^{n} 2^i = 2^{n+1}-2 \in \Theta(2^n)$.
$f_2(i)=i$: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2) \ne \Theta(n)$.

(d) Prove or disprove: If $f : \mathbb{N} \to \mathbb{N}$ is any positive-valued function, then either (1) there exists a constant $c > 0$ so that $f(n) \in O(n^c)$, or (2) there exists a constant $\alpha > 1$ so that $f(n) \in \Omega(\alpha^n)$.

Solution: False.

Let $f(n) = 2^{\sqrt{n}}$. $f(n) \in \Omega(n^c)$ for any constant $c > 0$ and the best case is asymptotically slower than n^c. $f(n) \in O(\alpha^n)$ for any constant $\alpha > 1$ and the worst case is asymptotically faster than α^n.

As a side note, this shows that there are algorithms whose running time grows faster than any polynomial but slower than any exponential. In other words, there exists a nether between polynomial-time and exponential-time.

3. (★★★ level) Recurrence Relations

Derive an asymptotic *tight* bound for the following $T(n)$. Cite any theorem you use.

(a) $T(n) = 2 \cdot T(\frac{n}{2}) + \sqrt{n}$.

Solution: Master theorem: $a = 2, b = 2, d = 1/2$. So that $d < \log_b a = 1$: $T(n) = \Theta(n)$

(b) $T(n) = T(n-1) + c^n$ for constants $c > 0$.

Solution: Expanding out the recurrence, we have $T(n) = \sum_{i=0}^{n} c^i$.

By the formula for the sum of a partial geometric series, for $c \neq 1$: $T(n) := \sum\limits_{i=0}^{n} c^i = \frac{1-c^{n+1}}{1-c}$. Thus,

- If $c > 1$, for sufficiently large k, $c^{k+1} > c^{k+1} - 1 > c^k$. Dividing this inequality by $c - 1$ yields: $\frac{c}{c-1} c^k > T(k) > \frac{1}{c-1} c^k$. Thus, $T(k) = \Theta(c^k)$, since $\frac{1}{c-1}$ is constant.
- If $c = 1$, then every term in the sum is 1. Thus, $T(k) = k + 1 = \Theta(k)$.
- If $c < 1$, then $\frac{1}{1-c} > \frac{1-c^{k+1}}{1-c} = T(k) > 1$. Thus, $T(k) = \Theta(1)$.

(c) $T(n) = 2T(\sqrt{n}) + 3$, and $T(2) = 3$.

Solution: The recursion tree is a full binary tree of height h, where h satisfies $n^{1/2^h} = 2$. Solving this for h, we get that $h = \Theta(\log\log n)$. The work done at every node of this recursion tree is constant, so the total work done is simply the number of nodes of the tree, which is $2^{h+1} - 1 = \Theta(\log n)$, so $T(n) = \Theta(\log n)$.

4. (★★★★ level) Recurrence Relations Part II

Solve the following recurrence relations and give a Θ bound for each of them.

(a) (i) $T(n) = 3T(n/4) + 4n^2$

(ii) $T(n) = 45T(n/3) + .1n^3$

(iii) $T(n) = 2T(\sqrt{n}) + 5$, and $T(2) = 5$. (Hint: this means the recursion tree stops when the problem size is 2)

(b) (i) Consider the recurrence relation $T(n) = 2T(n/2) + n\log n$. We can't plug it directly into the Master theorem, so solve it by adding the size of each layer.

Hint: split up the $\log(n/(2^i))$ *terms into* $\log n - \log(2^i)$*, and use the formula for arithmetic series.*

(ii) A more general version of Master theorem, like the one on Wikipedia, incorporates this result. The case of the master theorem which applies to this problem is:

If $T(n) = aT(n/b) + f(n)$ *where* $a \geq 1$, $b > 1$, *and* $f(n) = \Theta(n^c \log^k n)$ *where* $c = \log_b a$, *then* $T(n) = \Theta(n^c \log^{k+1} n)$.

Use the general Master theorem to solve the following recurrence relation:

$T(n) = 9T(n/3) + n^2 \log^3 n$.

Solution:

(a) (i) Since $\log_4 3 < 1$, by the Master Theorem, $T(n) = \Theta(n^2)$.

(ii) Since $\log_3 45 > 3$, by the Master Theorem, $T(n) = \Theta(n^{\log_3 45})$.

(iii) The recursion tree is a full binary tree of height h, where h satisfies $n^{1/2^h} = 2$. Solving this for h, we get that $h = \log\log n)$. The work done at every node of this recursion tree is constant, so the total work done is simply the number of nodes of the tree, which is $2^{h+1} - 1 = \Theta(\log n)$, so $T(n) = \Theta(\log n)$.

(b) (i) The amount of work done at the ith layer is $2^i(n/2^i \log(n/2^i))$. Since there are a total of $\log_2 n$ layers, we can find the total work done as follows:

$$\begin{aligned}
\sum_{i=1}^{\log n} 2^i(n/2^i \log(n/2^i)) &= \sum_{i=1}^{\log n} n\log(n/2^i) \\
&= n\sum_{i=1}^{\log n} \log n - \log 2^i = n\log^2 n - n\sum_{i=1}^{\log n} i\log 2 \\
&= n\log n - n\log(2) \times \frac{(\log n)(\log n + 1)}{2} \\
&= n\log^2 n\left(1 - \frac{\log 2}{2}\right) - n\log n \times \frac{\log 2}{2} \\
&= \Theta(n\log^2 n)
\end{aligned}$$

where step (4) uses the formula for arithmetic series, and step (6) discards the constant factors, and then discards the second term because its log factor, $(\log n)$, is dominated by $(\log^2 n)$ in the first term.

(ii) Since $\log_3 9 = 2$ (the exponent on n), by Case 2 of Wikipedia's general Master theorem, $T(n) = \Theta(n^2 \log^4 n)$.

5. (★★★★ level) Two Sorted Arrays

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? *(You need to give a four-part solution for this problem.)*

Solution:

Main idea The algorithm will be a divide and conquer algorithm (without much conquering) which will eliminate half the candidate elements in each step by carefully comparing the middle elements of each list. 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 $a[\lfloor k/2 \rfloor]$
```

Correctness: Since the lists are sorted, we can guarantee that any element in either list is at least as large as all the elements before it. So, in the case that $n > k$, we need not consider any elements beyond the kth element of each list, for we already know of k smaller elements. In the case of $n < k < 2n$, added additional infinite valued elements will not affect the correctness of the algorithm because there certainly exist at least k other elements that will be chosen before them. Therefore, if the algorithm is correct for the $n = k$ case, it is also correct for this case.

We will now prove the correctness of the algorithm for the $n = k$ case by strong induction on k. First consider the base case, $k = 1$. This means we want the smallest element in the two sorted lists, which is correctly given by $\min(a[1], b[1])$ (the pseudocode is 1-indexed).

For the recursive case, we now consider arbitrary k, and assume we have proved the algorithm succeeds on any input with problem size i, where $i < k$. First we compare the medians of the two lists, which are $a[\lfloor k/2 \rfloor]$ and $b[\lfloor k/2 \rfloor]$. If they are equal, then we can conclude that either median is greater than $k/2$ elements in its own list and another $k/2$ elements in the other list, meaning they are the overall median as well, so we just return one of them.

If that is not the case, then one of the medians is greater; WLOG, assume $a[\lfloor k/2 \rfloor] < b[\lfloor k/2 \rfloor]$. Then for every element in the front half of list a, we have shown that every element in the back half of list b is greater. Combining that with the back half of list a, we have at least k elements out of a total of $2k$ that are greater. This means the kth element cannot be in this section, and we can throw out these $k/2$ elements and decrement k by $k/2$. Similarly, for the back half of list b, we have shown that the front halves of both lists are smaller, meaning the element in the back half of list b are all larger than the kth element. We can remove them and not decrement k.

This leaves us with a subproblem in which we have two sorted lists of length $k/2$ and are looking for the $k/2$th element. This is exactly the same problem but with a smaller input size!. By our inductive hypothesis, we know that applying the algorithm recursively will work from here on.

Runtime:
The input size of the problem is halved at each step. The work we do involves comparing the medians of the lists, and deciding which halves to keep. Assuming we can reassign the list pointers without copying the lists, this can all be done in constant time.

Therefore, on each recursive call to the algorithm, we do $\Theta(1)$ work before making 1 recursive call of half the size. The results in a recurrence relation of

$$T(n) = T(n/2) + \Theta(1)$$

By Master Theorem, this comes out to $T(k) = \Theta(\log k)$.

6. (★★★★★ level) Merged Median

Given k sorted arrays of length l, design an efficient algorithm to finding the median element of all the $n = kl$ elements. Your algorithm should run asymptotically faster than $O(n)$. Your answer from 5 may be helpful.

(You need to give a four-part solution for this problem.)

Solution:

There were two main types of solutions. One used partitioning on a pivot (this was probably the easier approach). The other involved deleting m elements larger than the median, and deleting m elements smaller than the median on each iteration, where m is some number that has more asymptotic significance than a constant. We can call the second approach the "gradual-deletion" method. The gradual-deletion method will have no sample solution provided, but feel free to ask about it. Here is a full solution for the partitioning approach.

Main Idea. Since we are asked for a solution faster than $O(kl)$, if we simply loop through all the elements in all k arrays, we are not able to achieve the runtime requirement. Therefore we should start to think about using a divide-and-conquer method. Actually, this question is another follow-up question on the median-finding problem that we talked about in the lecture. The algorithm is very similar to it with just a few modifications. Recall that in quickselect, we use a random pivot to partition the array. If we use a random pivot, the overall asymptotics will still work out, but we'd have to make justifications on the average case. Instead, we want to find a good pivot to partition all k arrays. The one that we are using here is the median of the medians of each sorted arrays, say x. Then we use such number as the pivot to partition each array into three pieces (one with elements smaller than x, one with elements equal to x, and one with elements bigger than x). Similar to quickselect, we can recursively use one of those three pieces to find the merged median. (See the pseudocode below to see the detailed implementation) However, we have an issue here: the arrays after partitioning can have various sizes! Some lists may remain large in the recursive call but some will be smaller. Therefore the median of medians theoretically can still be a bad pivot. So, in the recursion we dont actually use the median of medians. We want to have such a pivot that is greater than or equal to at least $1/4n$ elements. (See correctness to see the detailed explanation) Therefore we order all the arrays according to their medians. Then we start from the array that has the smallest median and add up the size of each array until the sum adds up to half of the total elements. Then we pick the median of next array to be the "pseudo" median of medians. This will guarantee us a good pivot.

In the following implementation, we design an auxiliary function `MergedSelect`$(A_1,\dots,A_k,i)$ which returns the `i`-th smallest element of the merged array $A_1,\dots,A_k$. The merged median can be found by calling `MergedSelect`$(A_1,\dots,A_k,\lfloor n/2 \rfloor)$.

Pseudocode.

```
procedure MergedSelect($A_1,\dots,A_k,i$)
    if length[$A_1$] + ... + length[$A_k$] = 1 then        ▷ End condition: one element left which is the median!
        return the only element in those arrays
    $m \leftarrow []$
    for $i = 1,\dots,k$ do
        $m[i] = (A_i[l/2], A_i)$        ▷ Find the median of each array and store (median, array) as a tuple
    Sort list $m$ according to the first element (medians) in an ascending order.
    halfSizeCounter $\leftarrow$ 0, indexCounter $\leftarrow$ 1
    while halfSizeCounter $< (\text{length}[A_1] + \cdots + \text{length}[A_k])/2$ do
        halfSizeCounter $\leftarrow$ halfSizeCounter + length[$m$[indexCounter][2]]        ▷ Consistent with 1-index
        indexCounter $\leftarrow$ indexCounter + 1
    medianOfMedians ← $m$[indexCounter][1]
    **for** $i = 1, \ldots, k$ **do**
        $B_i \leftarrow A_i[A_i <$ medianOfMedians$]$    ▷ $B$ get all the elements in $A$ which are less than the pivot
        $C_i \leftarrow A_i[A_i =$ medianOfMedians$]$    ▷ $C$ get all the elements in $A$ which are equal to the pivot
        $D_i \leftarrow A_i[A_i >$ medianOfMedians$]$    ▷ $D$ get all the elements in $A$ which are greater than the pivot
    **if** $i <=$ length$[B_1] + \cdots +$ length$[B_k]$ **then**    ▷ $i$-th element must be in $B$
        **return** MergedSelect($B_1, \cdots, B_k, i$)
    **else if** $i <=$ length$[B_1] + \cdots +$ length$[B_k]$ +length$[C_1] + \cdots +$ length$[C_k]$ **then**    ▷ In C
        **return** medianOfMedians
    **else**    ▷ In D
        **return** MergedSelect($D_1, \cdots, D_k, i-$length$[B_1] - \cdots -$ length$[B_k]$ $-$length$[C_1] - \cdots -$ length$[C_k]$)
```

Correctness: The key proof of correctness is very similar to the one shown in problem 5 and in the book median-finding section. The only tricky part that we want to show here is the way that we choose the "medianOfMedians" will eliminate at least 1/4 of total elements.

Suppose $A_1^*, \ldots A_k^*$ are those sorted arrays according to their medians. The algorithm above adds up the size of each array until the sum is greater than $n/2$. Suppose we stop adding at array A_i^*. Then "medianOfMedians" guarantees that

$$\text{length}(A_1^*) + \cdots + \text{length}(A_i^*) \geq n/2$$

and

$$\text{median}(A_1^*) \leq \cdots \text{median}(A_i^*) \leq \text{medianOfMedians}.$$

In arrays $A_1^*, \ldots A_i^*$, each median is greater than or equal to $\lfloor l/2 \rfloor$ elements in each array. There are $\geq n/2$ elements in those arrays. Since each median is less than or equal to our "median of medians", the "median of medians" must be greater than at least $n/4$ elements in all k arrays.

Running time: Finding the median of each array is essentially choosing the middle element which takes $O(1)$ per array. Then, finding the median of these medians which takes $O(k \log k)$ time since we need to sort them. Then, we use this element to partition the elements in each list. By binary search we can find which are bigger and which are smaller, it requires $O(k \log l)$ time.

Since we eliminate at least $n/4$ elements from the array, we have the recurrence relation

$$T(n) \leq O(k \log l) + O(k \log k) + T(3n/4).$$

The first term is the time to find the split index for the arrays using binary search in each array. The second for finding the median of medians and the third for the recursive call. Initially there were $n = kl$ elements in all, and after $O(\log l)$ recursive calls we have $O(k)$ elements, in which case we can finish in time $O(k)$. Thus, the total cost is bounded by the time for $O(\log l)$ recursive calls. This is bounded by $O(k \log^2 l + k \log l \log k)$.

Note: We allow a randomized algorithm: randomly pick from the medians of the arrays as the pivot. This will give us an expected sub-linear runtime. However, we cannot just pick the median of medians for an expected sub-linear runtime because certain inputs will not achieve it. Whereas if we randomly pick the pivot, these inputs will still have the expected runtime we are looking for.