CS 170 Fall 2017 — Efficient Algorithms and Intractable Problems — 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

(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

$$O(12) \in O(\log_2(n)) \in O(n^{\frac{1}{3}}) \in O(n^{\frac{1}{2}}) \in O(2^{\log_2(n)}) \in O(n^3) \in O(2^{\sqrt{n}}) \in O(2^n) \in O(3^n)$$

$$3,7,2,5,4,9,8,6,1$$

(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.

$$\frac{\ln(n)}{\ln(3)} \in \theta(\frac{\ln(n)}{\ln(4)}) \Rightarrow f \in \theta(g)$$

ii.

$$\lim_{n\to\infty} \frac{4\log(n)}{3n\log(n)} = 0$$

$$f \in O(g)$$

iii.

$$\lim_{n\to\infty} \frac{\sqrt{n}}{(\log(3n))^3}$$

Take the derivative wrt n of numerator and denominator five times, and L'Hopital gives:

$$\lim_{n\to\infty} \frac{\sqrt{n}}{48} = \infty \Rightarrow f \in \Omega(g)$$

iv.

$$\forall n \in \mathbb{R}, a*2^n \leq 2^{n+1} \leq b*2^n, a = \frac{1}{4}, b = 4$$

$$f \in \theta(g)$$

v.

$$n \in \Omega((\log(n))^{\log(\log(n))})$$

Because the log is not specified, I rewrite this as

$$n \in \Omega((\ln(n))^{\ln(\ln(n))})$$

Then, make the substitution $s = \ln(n)$

$$(s^{log_s(e)})^s \in \Omega(s^{\ln(s)})$$

Now we can just compare the exponents

$$s\frac{\ln(e)}{\ln(s)} \in \Omega(\ln(s))$$

This is true because

$$\lim_{s\to\infty} \frac{s}{(\ln(s))^2} = \infty$$
$$f \in \Omega(g)$$

(Alternate Solution)
For large n.

$$2\log(\log(\log(n))) \le \log(\log(n))$$
$$\Rightarrow (\log(\log(n)))^2 \le \log(n)$$
$$\Rightarrow \log((\log(n))^{\log(\log(n))}) \le \log(n)$$
$$\Rightarrow (\log(n))^{\log(\log(n))} \le n$$
$$\Rightarrow g \le f$$
$$f \in \Omega(g)$$

vi. I omit the calculus.

$$\lim_{n\to\infty} \frac{f}{g} = 1 \Rightarrow f \in \theta(g)$$

vii.

$$(\frac{n}{2})^{\frac{n}{2}} \le n!$$
$$\frac{n}{2}(\log(n) - \log(2)) \le \log(n!)$$
$$n\log(n) \in O(\log(n!))$$
$$f \in \Omega(g)$$
$$\log(n!) = \log(n) + \log(n-1) + \log(n-2) + \ldots$$
$$n\log(n) = \log(n) + \log(n) + \ldots$$
$$n\log(n) \in \Omega(\log(n!))$$
$$f \in O(g)$$
$$f \in \Omega(g) \wedge f \in O(g) \Rightarrow f \in \theta(g)$$

(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:

$$\sum_{i=1}^{n} 2^n = 2^{n+1} \in \theta(2^n)$$
$$\sum i = 1^n n = \frac{n(n+1)}{2} \in \theta(n^2)$$
$$f_1(n) = 2^n, f_2(n) = n$$

(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: Because the function maps natural numbers to natural numbers, there are only a number of possible options for the time complexity of f.

$$f(n) \in \theta(1), \theta(n), \theta(n^c), \theta(c^n), \theta(n!), \theta(n^n)$$

Then, all of these possibilities satisfy the claim.

$f(n) = 1 \in O(n), f(n) = n \in O(n^2), f(n) = n^k \in O(n^k), f(n) = k^n \in \Omega(k^n), f(n) = n! \in \Omega(k^n), f(n) = n^n \in \Omega(k^n)$

Proved. $\square$

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: Using the Master Theorem, we have:

$$a = 2, b = 2, d = \frac{1}{2}$$

$$T(n) \in \theta(n^{log_b(a)}) = \theta(n)$$

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

Solution: Each call of the function on n recursively calls itself on $n-1$, and then does c^n work. The call $T(1)$ returns 1, and so all the work can be seen as the sum of c^n from 2 to n.

$$\sum_{i=2}^{n} c^i = \frac{a_1(1-r^n)}{1-r} = \frac{c^2(1-c^{n-1})}{1-c} \in \theta(c^n)$$

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

Solution:

$$T(n) = 2T(\sqrt{n}) + 3$$

$$T(n) = 2(2T(\sqrt{\sqrt{n}}) + 3) + 3$$

Because we are taking the square root of n each time, and we stop when $n = 2$, we can think of the recursion as halving the exponent each level deeper. Thus, because $n = 2^{log_2(n)}$, the number of halvings before the exponent becomes 1 is $log(log(n))$. Thus:

$$T(n) = 2^{log(logn)}T(2) + 3 * \sum_{i=0}^{log(logn)} \in \theta(log(n))$$

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

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

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

Solution: Using the Master theorem, where

$$a = 3, b = 4, d = 2, d > \log_b(a)$$

$$T(n) \in \theta(n^2)$$

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

Solution: Using the Master theorem, where

$$a = 45, b = 3, d = 3, d < \log_b(a)$$

$$T(n) \in \theta(n^{\log_3(45)})$$

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

Solution: This is similar to problem 4.a.3, so we can go about it the same way.

$$T(n) = 2T(\sqrt{n}) + 5$$

$$T(n) = 2((2T(\sqrt{\sqrt{n}} + 5))) + 5$$

Thus, we know from before that there will be a depth of $\log(\log(n))$ in the recursive tree, and so we can express $T(n)$ as

$$T(n) = 2^{\log(\log(n)}T(2) + 5 * \sum_{i=0}^{\log(logn))} 2^i \in \theta(\log(n))$$

(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.*

Solution: From the recursion tree, we have the summation:

$$\sum_{i=0}^{log_2(n)} 2^i \frac{n}{2^i} \log(\frac{n}{2^i})$$

$$= n * \sum_{i=0}^{log_2(n)} (\log(n) - i\log(2))$$

$$= n * \sum_{i=0}^{log_2(n)} \log(n) - n\log(2) \sum_{i=0}^{\log_2(n)} i$$

$$= n(\log(n))^2 - (n\log(2))\frac{\log_2(n)(\log_2(n)+1)}{2}$$

$$T(n) \in \theta(n(\log(n))^2)$$

(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: Using the general Master theorem:

$$a = 9, b = 3, c = 2, k = 3$$

$$T(n) = \theta(n^2(\log(n))^4)$$

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: In order to take advantage of both arrays being sorted, notice that when looking at the medians of the two arrays, as well as the index desired, then at any step we can eliminate half of one array. Say we have our 2n elements, as well as two arrays A and B. Then there are a number of cases.

1. $k > n$, and median of A less than median of B. We can eliminate the lower half of A. And shift our k back by half of A in order to compensate. This is because if our index was in A, then it would have to be in the upper half. Suppose it was in the lower half. Then that means that there are $> \frac{n}{2}$ elements after it in the upper half of A. Furthermore, because the median of B is larger than that of A, then that means that there would also be $> \frac{n}{2}$ elements after it in the upper half of B. But this is impossible! It cannot be that there are both $> \frac{n}{2}$ elements greater than the kth element in both A and B, because that would mean $k > n$ is false. Thus, it must be valid to remove the lower half of A, as it is impossible for the kth element to be there.
2. $k > n$, and median of A greater than median of B. With similar reasoning, eliminate the lower half of A and shift k back by half of A to compensate.
3. $k < n$, and median of A less median of B. With similar reasoning, eliminate the upper half of B. No need to shift k.
4. $k < n$, and median of A greater than median of B. With similar reasoning, eliminate the upper half of A. No need to shift k.

Pseudocode:
```
kthElement(a, b, k)
  mid1 = middle element of first array
  mid2 = middle element of the second array
  if a has length 0
    return kth element of b
  elif b has length 0
    return kth element of a
    if k is greater than sum of mid1 and mid 2
      if a[mid1] greater than b[mid2]
        return kthElement(a, second half of b, k - mid2 - 1)
      else
        return kthElement(second half of a, b, k - mid1 - 1)
    else
      if a[mid1] greater than b[mid2]
        return kthElement(first half of a, b, k)
      else
        return kthElement(a, first half of b, k)
```
Proof of Correctness
Proof by induction on $t = 2n$, the combined number of elements in the two arrays:
Base case: $n = 1, t = 2$. There are a number of options.

1. Two arrays of length 1: Then k can either be 1 or 2. If $k = 1$, then eliminate the second array from our

search, and if $k = 2$, then eliminate the first array from our search. Either one of those options would lead us to our base case in the pseudocode, and return the element in the remaining array.

2. One array of length 2: The other of length 0. This automatically goes to the base case, the case where there is only one non-zero array left. Then, simply get the desired element by indexing into the array.

Inductive Hypothesis: If we have total number of elements t' where our algorithm is known to work, then our algorithm must also work for total number of elements $2t'$.
Inductive Step: We do not assume that our proposed $2t'$ elements are split half and half, because this may not necessarily be the case somewhere along the execution of our algorithm. All we assume is that the $2t'$ elements are somehow split between two arrays. Using our trick of eliminating half of one of the two arrays, we can keep doing this until we reach t' or less elements, where in which the problem is solved because of the inductive hypothesis. However, this may not be the case, as one of the arrays may be exhausted before the total number of elements between both arrays goes below t'. But one of the arrays are exhausted, our problem is also solved because the remaining array is in sorted order and can easily be indexed into for the desired element.

(Alternative explanation)
This is mostly explained in the main idea portion. The critical part is that at each stage we can halve one of the arrays based on the two medians and the value of k, as it can be proven that it is impossible for the kth element to be in that half. Shifting k when removing from the ”front” (array A) guarantees that the index is maintained. Then, the recursive call takes care of the rest. When one list is depleted, then we can easily find the kth element in the remaining sorted list.
Running Time$O(\log(n))$
JustificationThere are two lists of size n. In the worst case, we must half each list $\log(n)$ times until one reaches size 0. This will take $2\log(n)$ steps.

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:
Main Idea
The main idea behind this algorithm is similar to that of problem 5. At any point in algorithm, half of either the largest array or smallest array can be removed.
Pseudocode
```
mergedMedian(listOfArrays, medianIndex)
  If listOfArrays contains only one nonempty array
    Get medianIndex of nonempty array
  Find the k medians of the k arrays.
  Find the two arrays with largest and smallest medians.
  Discard lower half of smallest median array.
  Discard upper half of largest median array.
  Modify median index by subtracting by number of elements discarded in lower half.
  Call mergedMedian(modifiedListOfArrays, modifiedMedianIndex)
```
Proof of Correctness
The reason that the halving of the two arrays is valid is because at any iteration, we know that the median cannot be in the lower half of the smallest median array, and the upper half of the largest median array. This

is because if the lower half of the smallest median array contained the median of the union, then there would be a contradiction, as there are more than $n/2$ elements larger than the median. Symmetrically, the median of the union cannot be within the upper half of the largest median array. This is because if that were the case then there would be a contradiction, as there are more than $n/2$ elements less than the median. Thus, removing the two halves is a valid move.