CS 170 Spring 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 Tuesday, January 31, at 11:59am

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) Shortcut Question: Asymptotic Complexity Comparisons

Please read the instruction for shortcut questions carefully. You only need to submit the part that you are able to solve. Please clearly indicate which part you submit.

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

(a) $f_1(n) = 3^n$
(b) $f_2(n) = n^{\frac{1}{3}}$
(c) $f_3(n) = 12$
(d) $f_4(n) = 2^{\log_2 n}$
(e) $f_5(n) = \sqrt{n}$
(f) $f_6(n) = 2^n$
(g) $f_7(n) = \log_2 n$
(h) $f_8(n) = 2^{\sqrt{n}}$
(i) $f_9(n) = n^3$

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

(b) Prove that for any $\varepsilon > 0$ we have $\log x \in O(x^\varepsilon)$.

Solution: The limit of both functions goes to infinity and both are monotonically increasing. $\frac{d^2x}{dx^2}\log(x) < 0$ but $\frac{d^2x}{dx^2}x^\varepsilon > 0$. Therefore there must exist a point after which $x^\varepsilon > \log x$ for some x large enough. More formally, using l'Hopital's rule:

$$\begin{aligned}\lim_{x\to\infty}\frac{\log x}{x^\varepsilon} &= \lim_{x\to\infty}\frac{\frac{d}{dx}\log x}{\frac{d}{dx}x^\varepsilon}\\ &= \lim_{x\to\infty}\frac{\frac{1}{x}}{\varepsilon x^{\varepsilon-1}}\\ &= \lim_{x\to\infty}\frac{1}{\varepsilon x^\varepsilon} = 0\end{aligned}$$

And so therefore $\log x \in O(x^\varepsilon)$.

(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.
$f_1(i) = 2^i$: $\sum_{i=1}^n 2^i = 2^{n+1} - 2 \in \Theta(2^n)$.
$f_2(i) = i$: $\sum_{i=1}^n i = \frac{n(n+1)}{2} \in \Theta(n^2) \neq \Theta(n)$.

(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) Bound the running time

For each of the following functions, give a tight $\Theta(\cdot)$ bound on the running time. Justify your answer. For this problem you can assume all arithmetic operations run in $O(1)$ time.

Example:

```
function f1(n):
    if n < 3:
         return n
    else:
         return f1(n-1) + f1(n-2)
```

Solution: This algorithm runs in exactly $\Theta(fib(n))$ time, or $\Theta((\frac{1}{2}(1+\sqrt{5}))^n)$. There are exactly $fib(n)$ calls used to compute the nth Fibonacci number.

(a) (Hint: how is this different than the example?)

```
function f2(n):
    if n < 3:
         return n
    else:
         return f2(n/2) + f2(n/4) // division truncates
```

Solution: Instead of subtracting one and subtracting two, this can be seen as bit-shifting by one and bit-shifting by two. Therefore we compute the mth Fibonacci number where m is the bit-length of n. $\Theta(fib(\log n))$

(b)
```
function f3(n):
    if n == 1: return 0
    if n is even:
        return f3(n/2)
    else:
        return f3(n+1)
```

Solution: This algorithm will run at most $2\log n$ iterations when $n = 2^k + 1$ for some k and at least $\log n$ iterations when $n = 2^k$. So $f_5(n) \in \Theta(\log n)$

(c) (Extra Credit Question: *For Eternal Fame Or Eternal Misery*)

```
function f4(n):
    if n == 1: return 0
    if n is even:
        return f4(n/2)+1
    else:
        return f4(3n+1)+1
```

Solution: This is an unsolved problem in computer science. Despite only differing from (b) in one location, it is even unknown if the program always terminates for all values of n.

5. (★★★★ level) γ-approximate Median

In the lecture, we saw an algorithm to compute Median that runs in *expected linear time*, i.e., an algorithm whose average run-time is linear, but in rare cases could take much longer. In this problem, we will design a different algorithm to compute the median that runs in linear time, always!

Consider an array $A[1 \dots n]$ of distinct integers. The *median* of A is the $\lceil n/2 \rceil$-th largest integer in the array. A *γ-approximate median* of A is one of the integers that is not one of the $\lfloor \gamma n - 1 \rfloor$ smallest integers, and not one of the $\lfloor \gamma n \rfloor$ largest integers. For example, the median is a 1/2-approximate median. You may assume that $0 < \gamma \leq \frac{1}{2}$.

(a) You should see that finding GAM is an easier problem than finding the exact median. Now assume that you have a linear time algorithm for GAM. Use this GAM algorithm as a subroutine to design a linear time algorithm for finding the median. Please note that γ is a fixed constant.

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

Solution:

Main Idea. In the lecture, we have seen an algorithm to computer the median of an array. However, we also see that if we choose the pivot badly, the algorithm will end up getting $O(n^2)$ runtime performance. Here the key idea is still very similar to median finding algorithm that we presented in lecture. However, instead of randomly using a pivot, we use the pivot directly from the GAM procedure. The pivot that GAM returns is a better pivot as it guarantees that it is not those very small nor very large numbers. Therefore it won't end up with the worst case scenarios as randomly picking up a pivot. In the following implementation, we design an auxiliary function `Select(A, i)` which returns the `i`-th smallest element of `A`. The median can be found by calling `Select(A, floor(n/2))`.

Pseudocode.

```
procedure SELECT($A, i$)
    if length[$A$] = 1 then
        return $A[1]$
    $x \leftarrow GAM(A)$
    $B \leftarrow A[A < x]$                          ▷ $B$ get all the elements in $A$ which are less than $x$
    $C \leftarrow A[A = x]$                          ▷ $C$ get all the elements in $A$ which are equal to $x$
    $D \leftarrow A[A > x]$                          ▷ $D$ get all the elements in $A$ which are greater than $x$
    if $i <=$ length[$B$] then                       ▷ $i$-th element must be in $B$
        return Select($B$, $i$)
    else if $i <=$ length[$B$]+length[$C$] then      ▷ $i$-th element must be in C
        return $x$
    else                                             ▷ $i$-th element must be in D
        return Select($D$, $i-$length[$B$]$-$length[$C$])
```

Correctness. This algorithm is correct for the same reason the select algorithm in the text (Section 2.4 Median finding) is correct; no modification of the proof is required.

Running time. Let n = `length[A]`. In the worse-case scenario, the pivot we used from GAM will eliminate $\lfloor \gamma n \rfloor$ elements from the array. Therefore every time when we call SELECT, the size of the input array will be at most $\lceil (1-\gamma)n \rceil$. Since we are given the linear time GAM procedure, the runtime to ”combine subproblems together” is $\Theta(n)$. So the running time satisfies

$$T(n) \leq T((1-\gamma)n) + \Theta(n)$$

which implies $T(n) \in \Theta(n)$ by the master theorem.

(b) Consider the following high-level algorithm:

1. Partitioning the elements into $\lceil n/5 \rceil$ groups of 5 elements each where the last group may have less than 5.
2. Find the exact median of each group for all groups in $O(n)$ time.
3. Find the exact median, x, of the $\lceil n/5 \rceil$ group medians. If you need to choose between two medians, always prefer medians that derived from a full group of 5, over ”pseudo-medians” that derived from a group with less than 5.

Give an algorithm for step 2 (You only need to briefly describe your idea and why it is $O(n)$). Prove that there exists a choice of γ and a constant $n_0 > 0$, such that x is a γ-approximate median when $n \geq n_0$. (Hint: you can consider the choice that $\gamma = \frac{3}{10}$)

Solution:

1. Algorithm for step 2.

The key idea here is that sorting an array with constant amount of elements takes $O(1)$. Using insertion sort to find the median of ≤ 5 elements takes $O(1)$ time. This can be done to all $\lceil n/5 \rceil$ groups in $O(n)$ time.

2. Proof the existence of γ.

Since x is the exact median of the group medians, x is not smaller than $\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil$ of the group medians. In turn, each group median is not smaller than 2 other elements, except possibly the last group. In order to give a tight lower bound for x, we also need to discount these two possible exceptions and also x itself. So x is not smaller than $3\lceil \frac{1}{2} \lceil \frac{n}{5} \rceil \rceil - 2 - 1 \geq \frac{3}{10}n - 3$ other elements. For the same reason, x is not larger than $\frac{3}{10}n - 3$ other elements.

Now we can show how to find such γ. Suppose say γ is 1/4. Then in order to make the procedure above return the 1/4-approximate median, we must make sure that

$$\frac{3}{10}n - 3 \geq 1/4n.$$

By solving this, you will end up with a requirement that $n \geq 600$. Therefore we showed that when γ is 1/4, n need to be at least 600 (n_0) to make x the γ-approximate median.

We can even generalize our finding above by letting $\gamma = \frac{3}{10} - \varepsilon$. Then again in order to ensure γ-approximate properties,

$$\frac{3}{10}n - 3 \geq \frac{3}{10}n - \varepsilon n.$$

Simplifying the inequality will give us $n_0 = \frac{3}{\varepsilon}$. Then finally we reach to the conclusion that for any ε, as long as $n \geq n_0 = \frac{3}{\varepsilon}$, x is a γ-approximate median where $\gamma = \frac{3}{10} - \varepsilon$. (You can pick any ε, (You only need to find one such gamma. This part is more general than what was asked.)

Aside: It turns out that using the algorithm of (b) as GAM in part (a) will give an algorithm that finds the median in worst-case linear time (this is tricky to analyze since both algorithms call each other).

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 $O(n)$.

(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, in problem 5, we use a guaranteed good pivot (γ-approximate median) to partition the array. Again here we also 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 ← indexCounter + 1
    medianOfMedians ← $m$[indexCounter][1]
    for $i = 1, \dots, k$ 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 $A_1^*, \dots A_k^*$ 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 $A_1^*, \dots A_i^*$, 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 $O(k \log^2 l + k \log l \log k)$.