CS 170 Homework 1

Due **09/07/2020, at 10:00 pm (grace period until 10:30pm)**

1 Three Part Solution

For each of the algorithm-design questions, please provide a three part solution which includes:

1. The main idea **or** pseudocode underlying your algorithm
2. A proof of correctness
3. A runtime analysis

Further explanation of this format is on the website (`https://cs170.org/resources/homework-guidelines/`).

2 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

3 Course Policies

(a) What dates and times are the exams for CS170 this semester?

Will we be offering alternate exams?

Solution:

(a) **Midterm 1**: Thursday 10/1, 6-8 pm PST
(b) **Midterm 2**: Thursday 10/29, 6-8 pm PST
(c) **Final**: Wednesday 12/16, 6-9 pm PST

There will be no alternate exams. Students will email cs170@berkeley.edu if they are located in a timezone where it isn't feasible to take the exams at the scheduled times.

(b) Homework is due at 10:00pm, with a late deadline at 10:30pm. At what time do we recommend you have your homework finished?

Solution: 10:00pm

(c) We provide 2 homework drops for cases of emergency or technical issues that may arise due to homework submission. If you miss the Gradescope late deadline (even by a few minutes) and need to submit the homework, what should you do?

Solution: The 2 homework drops are provided in case you have last minute issues and miss the Gradescope deadline. Homework extensions are not granted because solutions need to be released soon after the deadline, and so you do nothing.

(d) What is the primary source of communication for CS170 to reach students? We will email out all important deadlines through this medium, and you are responsible for checking your emails and reading each announcement fully.

Solution: The primary source of communication is Piazza.

(e) Please read all of the following:

(i) **Syllabus and Policies:** `https://cs170.org/syllabus/`

(ii) **Homework Guidelines:** `https://cs170.org/resources/homework-guidelines/`

(iii) **Regrade Etiquette:** `https://cs170.org/resources/regrade-etiquette/`

(iv) **Piazza Etiquette:** `https://cs170.org/resources/piazza-etiquette/`

Once you have read them, copy and sign the following sentence on your homework submission.

”I have read and understood the course syllabus and policies.”

Solution: I have read and understood the course syllabus and policies. -Alan Turing

4 Understanding Academic Dishonesty

Before you answer any of the following questions, make sure you have read over the syllabus and course policies (`https://cs170.org/syllabus/`) carefully. 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 their homework solutions, some of which overlap with this semester's problem sets. You look at their 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. They tell you how to do the first three problems.

Solution: Not OK.

(c) You look up a homework problem online and find the exact solution. You then write it in your words and cite the source.

Solution: Not OK. As a general rule of thumb, you should never be in possession of any exact homework solutions other than your own.

(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.

5 Asymptotic Complexity Comparisons

(a) Order the following functions so that for all i, j, if f_i comes before f_j in the order then $f_i = O(f_j)$. Do not justify your answers.

- $f_1(n) = 3^n$
- $f_2(n) = n^{\frac{1}{3}}$
- $f_3(n) = 12$
- $f_4(n) = 2^{\log_2 n}$
- $f_5(n) = \sqrt{n}$
- $f_6(n) = 2^n$
- $f_7(n) = \log_2 n$
- $f_8(n) = 2^{\sqrt{n}}$
- $f_9(n) = n^3$

As an answer you may just write the functions as a list, e.g. $f_8, f_9, f_1, \ldots$

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. Recall that in terms of asymptotic growth rate, logarithmic < polynomial < exponential.

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

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) $f = \Omega(g)$; any polynomial dominates a product of logs.

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

6 Computing Factorials

Consider the problem of computing $N! = 1 \times 2 \times \cdots \times N$.

(a) N is $\log N$ bits long (this is how many bits are needed to store a number the size of N). Find an $f(N)$ so that $N!$ is $\Theta(f(N)))$ bits long. Simplify your answer as much as possible, and give an argument for why it is true.

Note: You may not use Stirling's formula.

Solution: When we multiply an m bit number by an n bit number, we get an $(m+n)$ bit number. When computing factorials, we multiply N numbers that are at most $\log N$ bits long, so the final number has at most $N \log N$ bits.

But if you consider the numbers from $\frac{N}{2}$ to N, we multiply at least $\frac{N}{2}$ numbers that are at least $\log N - 1$ bits long, so the resulting number has at least $\frac{N(\log N - 1)}{2}$ bits.

(b) Give a simple (naive) algorithm to compute $N!$. Use $\Theta(mn)$ as the runtime for multiplying an m-bit number and an n-bit number.

For this problem, you don't need to write a proof of correctness (that is, just state your algorithm and analyze its runtime).

Solution: We can compute $N!$ naively as follows:

```
factorial ($N$)
    $f = 1$
    for $i = 2$ to $N$
        $f = f \cdot i$
```

Running time : $\Theta(N^2 \cdot \log^2 N)$.

We have N iterations, each one multiplying an $N \cdot \log N$-bit number (at most) by an $\log N$-bit number. Using the naive multiplication algorithm, each multiplication takes time $O(N \cdot \log^2 N)$. Hence, the running time is $O(N^2 \log^2 N)$.

A lower bound of $\Omega(N^2 \cdot \log^2 N)$ follows because the product of the first $N/2$ numbers will be at least $\Omega(N \log N)$ bits long by the previous part, and the last $N/2$ multiplications take $\Omega(N \log^2 N)$ time each.

7 Decimal to Binary

Given the n-digit decimal representation of a number, converting it into binary in the natural way takes $O(n^2)$ steps. Give a divide and conquer algorithm to do the conversion and show that it does not take much more time than Karatsuba's algorithm for integer multiplication.

Just state the main idea behind your algorithm and its runtime analysis; no proof of correctness is needed as long as your main idea is clear.

Solution: Similar to Karatsuba's algorithm, we can write x as $10^{n/2} \cdot a + b$ for two $n/2$-digit numbers a, b. The algorithm is to recursively compute the binary representations of $10^{n/2}, a$, and b. We can then compute the binary representation of x using one multiplication and one addition.

The multiplication takes $O(n^{\log_2(3)})$ time, the addition takes $O(n)$ time. So the recurrence we get is $T(n) = 3T(n/2) + O(n^{\log_2(3)})$. This has solution $O(n^{\log_2(3)} \log n)$.

(There is an $O(n^{\log_2(3)})$-time algorithm: We can compute the binary representation of 10^n in time $O(n^{\log_2(3)})$ by doing the multiplications $10 * 10, 10^2 * 10^2, 10^4 * 10^4 \ldots$ - note that the multiplication of $10^{n/2} * 10^{n/2}$ dominates the total runtime of these multiplications. This gives the recurrence $T(n) = 2T(n/2) + O(n^{\log_2(3)})$ instead, with solution $T(n) = O(n^{\log_2(3)})$.)

8 Maximum Subarray Sum

Given an array A of n integers, the maximum subarray sum is the largest sum of any contiguous subarray of A (including the empty subarray). In other words, the maximum subarray sum is:

$$\max_{i \leq j} \sum_{k=i}^{j} A[k]$$

For example, the maximum subarray sum of $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ is 6, the sum of the contiguous subarray $[4, -1, 2, 1]$.

Design an $O(n \log n)$-time algorithm that finds the maximum subarray sum.

Give a 3-part solution.

Hint: Split the array into two equally-sized pieces. What are the possibilities for the subarray, and how does this apply if we want to use divide and conquer?

Solution:

(i) **Main idea:** At first glance it seems like we have to find two indices i an j such that the subarray $A[i:j]$ has sum as large as possible. There are n^2 possibilities. However, there is a divide and conquer solution: split A into two equal pieces , L and R, defined as $A[1:n/2]$ and $A[n/2+1:n]$. There are two options for the subarray with the largest sum:

(a) It is contained entirely in L or R: Solve by recursion. Let s_1 be the result of recursively calling the algorithm on L, and s_2 be the result of recursively calling the algorithm on R.

(b) It "crosses the boundary", i.e. starts in L and ends in R: if the maximum subarray runs from $A[i:j]$, then $A[i, n/2]$ must be the maximum subarray ending at $A[n/2]$, and $A[n/2+1, j]$ must be the maximum subarray starting at $A[n/2+1]$. It might seem like we must now solve two largest subarray sum problems: find i such that the subarray sum $A[i, n/2]$ is as large as possible. And same for $A[n/2+1, j]$. Note that each of these require finding just one index i or j for which there are only $n/2$ possibilities each. So each can be computed in $O(n)$ steps. e.g. to find i, compute each successive subarray sum $A[k, n/2]$ for $k = 1$ to $n/2$ and take the maximum. Finally, let s_3 be the sum of the elements in $A[i, n/2]$ plus the sum of the elements in $A[n/2+1, j]$.

If n = 1, return $\max\{A[1], 0\}$. Else return $\max\{s_1, s_2, s_3\}$.

(ii) **Proof of correctness:** We prove by induction. When the array is length 1, our algorithm is clearly correct.

When the array is length at least 2, there are two options for the subarray with the largest sum:

(a) It is contained entirely in L or R.

(b) It "crosses the boundary", starts before element $n/2$ and ends after element $n/2$.

In the first case, either s_1 or s_2 is outputted by the algorithm, which we inductively assume is correct.

To handle the second case, the maximum subarray sum crossing the boundary must consist of the subarray with the largest sum ending in element $n/2$, and the subarray with the largest sum beginning with element $n/2+1$. So s_3 will be outputted by the algorithm in this case, which is correct.

(iii) **Runtime analysis:** Our method for computing s_3 takes $O(n)$ time, since each of the sums of $A[i : n/2]$ and $A[n/2+1 : i]$ is computed in $O(1)$ time given a previous sum. So we get the recurrence:

$$T(n) = 2T(n/2) + \mathcal{O}(n)$$

Using the Master theorem, this results in a runtime of $\mathcal{O}(n \log n)$.

9 Monotone matrices

A m-by-n matrix A is *monotone* if $n \geq m$, each row of A has no duplicate entries, and it has the following property: if the minimum of row i is located at column j_i, then $j_1 < j_2 < j_3 \ldots j_m$. For example, the following matrix is monotone (the minimum of each row is bolded):

$$\begin{bmatrix} \mathbf{1} & 3 & 4 & 6 & 5 & 2 \\ 7 & 3 & \mathbf{2} & 5 & 6 & 4 \\ 7 & 9 & 6 & 3 & 10 & \mathbf{0} \end{bmatrix}$$

Give an efficient (i.e., better than $O(mn)$-time) algorithm that finds the minimum in each row of an m-by-n monotone matrix A.

Give a 3-part solution. You do not need to write a formal recurrence relation in your runtime analysis; an informal summary of the runtime analysis such as "proof by picture" is fine.

Solution:

(i) **Main idea** If A has one row, we just scan that row and output its minimum.

Otherwise, we find the smallest entry of the $m/2$-th row of A by just scanning the row. If this entry is located at column j, then since A is a monotone matrix, the minimum for all rows above the $m/2$-th row must be located to the left of the j-th column. i.e. in the submatrix formed by rows 1 to $m/2-1$ and columns 1 to $j-1$ of A, which we will denote by $A[1 : m/2-1, 1 : j-1]$. Similarly, the minimum for all rows below the $m/2$-th row must be located to the right of the j-th column. So we can recursively call the algorithm on the submatrices $A[1 : m/2-1, 1 : j-1]$ and $A[m/2+1 : m, j+1 : n]$ to find and output the minima for rows 1 to $m/2-1$ and rows $m/2+1$ to m.

(ii) **Proof of correctness** We will prove correctness by (total) induction on m.

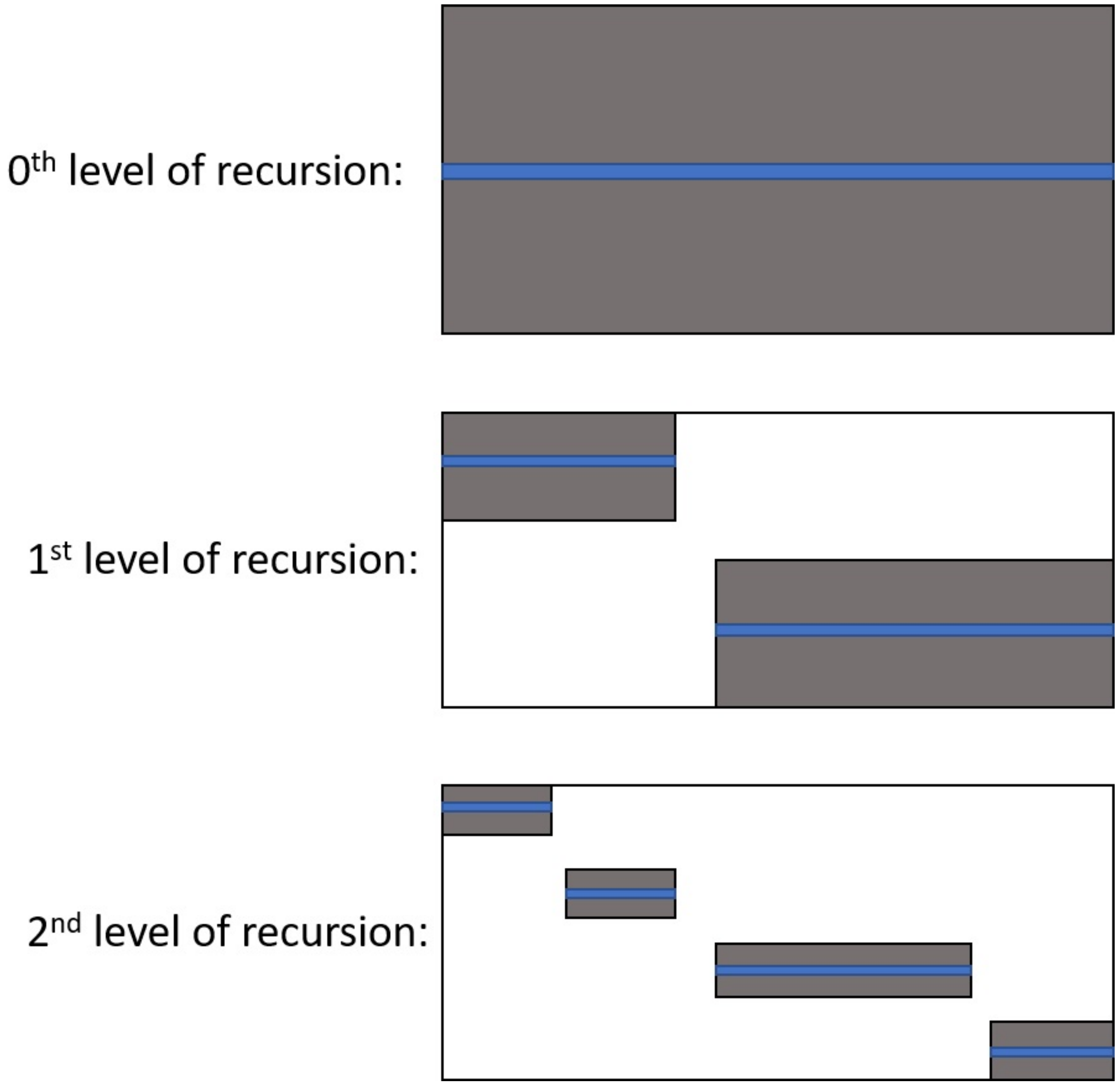
As a base case, $m = 1$, and the algorithm explicitly finds and outputs the minimum of the single row.

If A has more than one row, we of course find and output the correct row minimum for row $m/2$. As argued above, the minima of rows 1 to $m/2-1$ of A are the same as the minima of the submatrix $A[1 : m/2-1, 1 : j-1]$, and the minima of rows $m/2+1$ to m are the same as the minima of the submatrix $A[m/2+1 : m, j+1 : n]$. By the induction hypothesis, the algorithm correctly outputs the minima of these matrices, and together with the $m/2$ row above, they find and output the minima of all the rows of A.

(iii) **Running time analysis** There are two ways to analyze the run time. One involves doing an explicit accounting of the total number of steps at each level of the recursion, as we did before we relied on the master theorem, or as we did in the proof of the master theorem. Since m is halved at each step of recursion, there are $\log m$ levels of recursion. At each level of recursion, the number of columns of the matrix get split into two – those associated with the left matrix and those associated with the right matrix. Moreover, the number of steps required to perform the split is just n, since it involves scanning all the entries of a single row. This means that at any level of the recursion, all the submatrices have disjoint columns, meaning if the different submatrices have n_k columns, then $\sum_k n_k \leq n$. The total number of steps required to split these matrices to go to the next level of recursion is then just $\sum_k n_k \leq n$. So there are $\log m$ levels of recursion, each taking total time n, for a grand total of $n \log m$.

Actually to be more accurate, when m=1, $\log m = 0$, so the expression should be $n(\log m + 1)$ to get the base case right.

For a "proof by picture", consider the following picture, where the grey boxes represent the submatrices we're solving the problem for at each level of recursion, and the blue lines represent the rows we're scanning at each level of the recursion. We can see the total length of the blue lines in each level of the recursion is $O(n)$.



Another way to analyze the running time is by writing and solving a recurrence relation (though again, this isn't necessary for full credit). The recurrence is easy to write out: let $T(m, n)$ be the number of steps to solve the problem for an $m \times n$ array. It takes n time to find the minimum of row $m/2$. If this row has minimum at column j, we recurse on submatrices of size at most $m/2$-by-j and $m/2$-by-$(n - j)$. So we can write the following recurrence relation: $T(m, n) \leq T(m/2, j) + T(m/2, n - j) + n$.

This does not directly look the recurrences in the master theorem — it is "2-D" since it depends upon two variables. You need some inspiration to guess the solution. We will guess that $T(m, n) \leq n(\log m + 1)$. We can prove this by strong induction on m.

Base case: $T(1, n) = n = n(\log 1 + 1)$.
Induction step:

$T(m,n) \leq T(m/2,j) + T(m/2,n-j) + n \leq j\log(m/2) + 1 + (n-j)\log(m/2) + 1 + n$

(by the induction hypothesis)

$$= n(\log(m/2) + 1 + 1) = n(\log m + 1).$$