CompSci 330 Design and Analysis of Algorithms
Assignment 6, Fall 2024 Duke University

TODO: Add your name(s) here

Due Date: October 28, 2024

How to Do Homework. We recommend the following three step process for homework to help you learn and prepare for exams.

1. Give yourself 15-20 minutes per problem to try to solve on your own, without help or external materials, as if you were taking an exam. Try to brainstorm and sketch the algorithm for applied problems. Don't try to type anything yet.
2. After a break, review your answers. Lookup resources or get help (from peers, office hours, Ed discussion, etc.) about problems you weren't sure about.
3. Rework the problems, fill in the details, and typeset your final solutions.

Typesetting and Submission. Your solutions should be typed and submitted as a single pdf on Gradescope. Handwritten solutions or pdf files that cannot be opened will not be graded. LaTeX[1] is preferred but not required. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Any applied problems will request that you submit code separately on Gradescope to be autograded.

Writing Expectations. If you are asked to provide an algorithm, you should clearly and unambiguously define every step of the procedure as a combination of precise sentences in plain English or pseudocode. If you are asked to explain your algorithm, its runtime complexity, or argue for its correctness, your written answers should be clear, concise, and should show your work. Do not skip details but do not write paragraphs where a sentence suffices.

Collaboration and Internet. If you wish, you can work with a single partner (that is, in groups of 2), in which case you should submit a single solution as a group on Gradescope. You can use the internet, but looking up solutions or using LLMs is unlikely to help you prepare for exams. See the homework webpage for more details.

Grading. Theory problems will be graded by TAs on an S/I/U scale. Applied problems typically have a separate autograder where you can see your score. See the course policy webpage for details about homework grading.

[1] If you are new to LaTeX, you can download it for free at latex-project.org or you can use the popular and free (for a personal account) cloud-editor overleaf.com. We also recommend overleaf.com/learn for tutorials and reference.

Problem 1 (Recurrences Zoo). Solve the following recurrence relations and give a Θ bound for each of them. If you use the master theorem, state the values involved and which the recurrence statisfies in addition to the conclusion. For parts (g)–(k) in particular, consider unrolling/expanding[2] the recurrence to observe a pattern then simplify the resulting sum.

(a) $T(n) = 2T(n/3) + 1$

(b) $T(n) = 5T(n/4) + n$

(c) $T(n) = 7T(n/7) + n$

(d) $T(n) = 9T(n/3) + n^2$

(e) $T(n) = 8T(n/2) + n^3$

(f) $T(n) = 49T(n/25) + n^{3/2} \log n$

(g) $T(n) = T(n-1) + 2$

(h) $T(n) = T(n-1) + n^c$ where $c \geq 1$ is a constant

(i) $T(n) = T(n-1) + c^n$ where $c > 1$ is a constant

(j) $T(n) = 2T(n-1) + 1$

(k) $T(n) = T(\sqrt{n}) + 1$

Solution 1. We justify all answers using the recursion tree method.

(a) $\Theta(n^{\log_3 2})$. There are $\log_3 n$ levels, with 2^i work at each level. The total work is described by the sum $\sum_{i=0}^{\log_3 n} 2^i$, which is an increasing geometric sum, so it is dominated by the last term in the sum: $\Theta(2^{\log_3 n}) = \Theta(n^{\log_3 2})$.

(b) $\Theta(n^{\log_4 5})$. There are $\log_4 n$ levels, with $5^i(n/4^i) = n(5/4)^i$ work at each level. The total work is described by the sum $\sum_{i=0}^{\log_4 n} n(5/4)^i = n\sum_{i=0}^{\log_4 n}(5/4)^i$, which is an increasing geometric sum, so it is dominated by the last term in the sum: $\Theta(n \cdot (5/4)^{\log_4 n}) = \Theta(n^{\log_4 5})$.

(c) $\Theta(n \log n)$. There are $\log_7 n$ levels, with $7^i(n/7^i) = n$ work at each level. The total work is described by the sum $\sum_{i=0}^{\log_7 n} n = n \log_7 n = \Theta(n \log n)$.

(d) $\Theta(n^2 \log n)$. There are $\log_3 n$ levels, with $9^i(n/3^i)^2 = n^2$ work at each level. The total work is described by the sum $\sum_{i=0}^{\log_3 n} n^2 = n^2 \log_3 n = \Theta(n^2 \log n)$.

(e) $\Theta(n^3 \log n)$. There are $\log_2 n$ levels, with $8^i(n/2^i)^3 = n^3$ work at each level. The total work is described by the sum $\sum_{i=0}^{\log_2 n} n^3 = n^3 \log_2 n = \Theta(n^3 \log n)$.

(f) $\Theta(n^{3/2} \log n)$. There are $\log_{25} n$ levels, with

$$49^i(n/25^i)^{3/2} \log(n/25^i) = n^{3/2} \log(n/25^i) \cdot (49/25^{3/2})^i$$

work at each level. We upper bound this by replacing the $\log n/25^i$ term with $\log n$: $n^{3/2} \log n \cdot (49/25^{3/2})^i$. The total work is described by the sum

$$n^{3/2} \log n \sum_{i=0}^{\log_{25} n} (49/25^{3/2})^i.$$

[2]All of the recurrences can be solved without the master theorem, which itself can be proven with this technique. This technique is often visualized as a "recursion tree" as seen in lecture for the runtime analysis of mergesort.

This is a decreasing geometric sum since $49 < 25^{3/2}$, so is dominated by its first term: $O(n^{3/2} \log n \cdot (49/25^{3/2})^0) = O(n^{3/2} \log n)$. The first work in $T(n)$ is $n^{3/2} \log n$ itself and thus is $\Omega(n^{3/2} \log n)$, so we conclude $T(n) = \Theta(n^{3/2} \log n)$.

(g) $\Theta(n)$. This recurrence has 2 work per level and n levels, resulting in $\sum_{i=0}^{n} 2 = \Theta(n)$.

(h) $\Theta(n^{c+1})$. This recurrence has n levels with $(n-i)^c$ in the i-th level: $\sum_{i=0}^{n}(n-i)^c$. Consider the $n/2$ largest terms in this sum:

$$\sum_{i=0}^{n/2}(n-i)^c \geq \sum_{i=0}^{n/2}(n-n/2)^c = (n/2)(n/2)^c = (n/2)^{c+1} = \Omega(n^{c+1}).$$

Furthermore, we can upper bound each term in the original sum as n^c, resulting in at most $n \cdot n^c = O(n^{c+1})$. Therefore $T(n) = \Theta(n^{c+1})$.

(i) $\Theta(c^n)$. This recurrence has n levels and c^{n-i} work in the i-th level: $\sum_{i=0}^{n} c^{n-i} = \sum_{i=0}^{n} c^i$. Since $c > 1$ this is an increasing geometric sum, dominated by its last term: $\Theta(c^n)$.

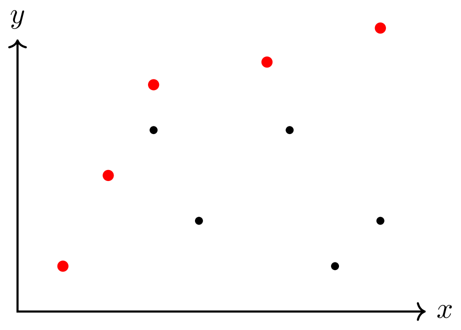
(j) $\Theta(2^n)$. This recurrence has n levels and 2^i work in the i-th level: $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 = \Theta(2^n)$.

(k) $\Theta(\log(\log n))$. Let k be the number of levels of recursion to be solved next, with 1 work per level, so the total work is k. The input value at the i-th level i is $n^{1/2^i}$. The base case occurs when the input is a constant $c \geq 1$. Thus the number of levels k is where

$$c = n^{1/2^k} \implies \log c = 1/2^k \log n \implies 2^k = \log n / \log c \implies k = \log(\log n / \log c)).$$

We conclude that $T(n) = \Theta(\log \log n))$.

Problem 2 (Northwest) Let $S = \{p_i = (x_i, y_i) : 1 \le i \le n\}$ be a set of n points on a 2D plane. We say that a point p_i is *northwest* (NW) of p_j if $x_i \le x_j$ and $y_i \ge y_j$, treating the $(-x)$-direction (resp., $(+y)$-direction) as west (resp., north). A point p_i is a *northwesterly* point of S if there are no other points in S NW of p_i. In the example below, the larger red points are the northwesterly points of S.



Describe an algorithm that returns the northwesterly points of S in $O(n \log n)$ time. You can assume[3] that the points of S are already sorted in increasing order of x-coordinate, breaking ties by the greater y-coordinate; e.g., $(1, 5)$ comes before $(1, 3)$. Justify the correctness of your algorithm, then state and justify its runtime, e.g., by unrolling/expanding, the recursion tree method, or applying master theorem.

[Hint: Divide S into two sets based on the x-coordinates of points, and "conquer." What is the appropriate merge/combine step to obtain the overall solution?

Solution 2. **Algorithm:** We first sort the array S in increasing order of x-coordinate. If two points have the same x-coordinate, we place the one with higher y-coordinate first.

1. Our algorithm $FindNW(S[1..n])$ takes in an array of points, sorted using the above procedure, and will output an array consisting of its northwesterly points.

2. If $n = 1$, return S, composed of a single point. Otherwise, we (recursively) find the northwesterly points of the points left and right of the median x-coordinate, respectively. Specifically, we set $m \coloneqq \lfloor n/2 \rfloor$, $L \coloneqq S[1..m]$ and $R \coloneqq S[m + 1..n]$. Then we compute

$$NW_L \coloneqq FindNW(L) \text{ and } NW_R \coloneqq FindNW(R).$$

3. Then we compute

$$y^* = \max_{(x,y) \in L} y,$$

   the maximum y-coordinate among of the left half of the points, L, by iterating through them.

4. Next, we find the subset of points $NW'_R \subseteq NW_R$ that have y-coordinate **strictly** greater than y^*. This is achieved by iterating through NW_R.

5. Finally we output $NW_L \cup NW'_R$ as the northwesterly points in S.

[3]This ordering could be obtained in $O(n \log n)$ time, e.g., using mergesort.

Correctness: For completeness, we give a full proof by induction, though an informal justification suffices for the problem.

Let $S[1..n]$ be an arbitrary set of points, sorted as described above. Assume that $FindNW$ returns the set of all northwesterly points on inputs smaller than S.

- *Base Case:* When $n = 1$, the algorithm simply returns S, which is indeed northwesterly as no point is to the northwest of it.

- *Inductive Case:* Consider a sorted input of length $n \geq 2$. By the induction hypothesis, $FindNW(S[1...m])$,
  $FindNW(S[m+1...n])$ correctly compute the northwesterly points NW_L, NW_R in the left and right halves, as their input is sorted. Consider the following four cases for a point $p \in S$:
  - $p \in (L \setminus NW_L) \cup (R \setminus NW_R)$. If $p \in L \setminus NW_L$ (resp., $p \in R \setminus NW_R$) then there is a point $q \in L$ (resp., $q \in R$) that is northwest of p, due to the correctness of NW_L, NW_R. $q \in S$ so p is not northwesterly in S.
  - $p \in NW_L$. We claim p is northwesterly in S. Indeed, by the correctness of NW_L there is no point in L that is northwest of p, and all points in R either have strictly greater x-coordinate or strictly smaller y-coordinate (due to how we break ties in the initial sorting). So p is northwesterly in S.
  - $p \in NW'_R$. We claim p is northwesterly in S. Indeed, there is no point in R that is northwest of it by the correctness of NW_R, and all points in L have strictly smaller y-coordinate than that of p (since it is in NW'_R). So p is northwesterly in S.
  - $p \in NW \setminus NW'_R$. Then p is not northwesterly, as follows. Let (q_x, q_y) be the point with greatest y-coordinate in $S[1...m]$, so $q_y = y^*$, and let $(x_p, y_p) = p$. Then we have $x_q \leq x_p$ (due to sorting) and $y_q \geq y_p$ (by definition of NW'_R), so q is northwest of p.

  Any point $p \in S$ satisfies exactly one of the four cases above. Thus, we have established that $NW_L \cup NW'_R$ is the set of all northwesterly points for S, which is returned by the algorithm.

Runtime Analysis: We obtain the following recurrence for the runtime $T(n)$ of $FindNW(S[1...n])$.

$$T(n) = 2T(n/2) + O(n).$$

Since we first recurse on two subproblems with half the input size, we have $2T(n/2)$. Computing y^L_{max} from L and NW'_R from NW_R take $O(n)$ time as they can each be done in a single for-loop over at most n elements. The solution to this recursion is $T(n) = O(n \log n)$, either by observing that this is the same recurrence as mergesort, or by noting that the recursion tree has depth $O(\log n)$ and has total work $O(n)$ at each level. Since the initial sorting can be done in $O(n \log n)$ time, the total running time is $O(n \log n)$.

Problem 3 (Twirling) An array $A[1..n]$ of n distinct integers (no duplicates) is considered *twirled* if it consists of two non-empty contiguous sequences, L and R, that are sorted in increasing order and all elements in L are greater than those in R. In other words, if R instead appeared left of L, the entire resulting sequence would be sorted in increasing order.

For example, in the following twirled array, the left subsequence is underlined and has length 6, the right subsequence is overlined and has length 4, and indeed all elements in the left subsequence are greater than those in the right subsequence:

$$[\underline{12,14,19,23,25,27},\overline{2,3,5,7}]$$

Describe an $O(\log n)$-time algorithm to return the length of the left subsequence of a given twirled array A. Justify its correctness, then state and justify its runtime, e.g., by unrolling/expanding, the recursion tree method, or applying master theorem.

[Hint: Consider a variant of binary search. Make sure that your recursive calls are always made on twirled arrays, as your recursive algorithm will likely assume and make use of this fact.]

Solution 3. The main idea is to determine whether the median element in the current subarray is the last element of the left subsequence. If yes, then we solve for the left subsequence's length using the median index and return it; in fact, the length is exactly the median index. Otherwise, we determine which side of the median element contains the last element of the left subsequence, then recursively search for it.

```
procedure LeftLength($A[1..n]$)
    if $n = 2$ then
        return 1                          ▷ Both subseq. are non-empty, so each has size 1.
    $m \leftarrow \lfloor n/2 \rfloor$
    if $A[m] > A[m+1]$ then               ▷ $A[m]$ is last element of last subseq. in $A$
        return $m$
    else if $A[1] \leq A[m]$ then         ▷ $A[m]$ lies in left subseq. of $A$
        return $m +$ LeftLength($A[m+1..n]$)
    else                                  ▷ $A[m]$ lies in right subseq. in $A$
        return LeftLength($A[1..m]$)
```

Correctness: For completeness, we give a full proof by induction, though an informal justification suffices for the problem.

Let $A[1..n]$ be an arbitrary twirled array. Assume that *LeftLength* returns the length of the left subsequence in any input twirled array smaller than A.

- *Base Case:* When $n = 2$, the left and right subsequences of A each have length 1 (since they are non-empty), and 1 is returned.
- *Inductive Case:* The only element in a twirled array greater than its next (to the right) is the last in the left subsequence. The if-condition after setting m checks if $A[m]$ is this element of A, and if yes, then m is correctly returned. Otherwise, there are two cases for $A[m]$. Both subsequences of A are sorted in increasing order and all elements of the right subsequence of A are less than every element, including $A[1]$. Thus, if $A[1] \leq A[m]$ then $A[m]$ is in the left subsequence of A, otherwise it is in the right subsequence of A. Let j be the length of the left subsequence in A.

- If $A[m]$ is in the left subsequence, then $j > m$ since $A[m]$ is not the last element of the left subsequence. It follows that $A[m+1..n]$ is twirled with left subsequence $A[m+1..j]$ and right subsequence $A[j+1..n]$. By the IH, the call LeftLength($A[m+1..n]$) correctly returns the length of the left subsequence in $A[m+1..n]$, $j-m$, so m plus the returned length is the desired length, which is returned.
- Otherwise $A[m]$ is in the right subsequence and $j < m$. Then $A[1..m]$ is indeed twirled with left subsequence $A[1..j]$ and right subsequence $A[j+1..m]$. By the IH, the call LeftLength($A[1..j]$) correctly returns the length of the left subsequence in A, j.

Runtime Analysis: The algorithm performs at most one recursive call with input size roughly $n/2$ and the non-recursive work is $O(1)$. Hence the runtime of LeftLength on a twirled array of length n is described by $T(n) = T(n/2) + 1$, which solved to $O(\log n)$ using the recursion tree method (or observing that the recurrence is the same as for binary search and other examples).

Problem 4 (Package Managing)   ** Autograder available 10/22 @ 5pm **

The FlipMaster is a new two-armed robot to organize packages on a conveyor belt, where the packages are sequenced one after another. (If their manufacturer has any luck, they will be acquired by Amazon for an enormous sum.)

The setting in which the FlipMaster is to be used is as follows. Every package on the conveyor has an identifier (ID) on it, which is a positive integer. The FlipMaster can interact with the packages with two basic operations:

- `peek(i)`: It returns the ID of the i-th package from the left in $O(1)$ time, and
- `flip(i,j)`: For two indices $i < j$, it uses its two arms to squeeze the i-th to $(j-1)$-th packages together, flips them around to reverse their ordering, then sets them back down. In effect, this *reverses* the subsequence of packages from index i to $j-1$, inclusive[4].

Example: Let $S = [4, 5, 2, 1, 3]$ be the sequence of IDs of the packages from left to right on the conveyor belt. Then `peek(1)` returns 5 and `flip(1,4)` flips the 1st through 3rd packages so that the IDs become $[4, \underline{1, 2, 5}, 3]$, where the underlined symbols are those in the affected subsequence.

Let `f` be a FlipMaster object which has n packages on its conveyor belt, let i, j be indices between 0 and n, and let p be a positive integer (the "pivot" ID). Design and implement a **divide-and-conquer** recursive method `flartition(f,i,j,p)` using only `peek` and `flip` operations on S, and returns an index k where:

- all packages with indices i to $k-1$ have IDs ***less than*** p, and
- all packages with indices k to $j-1$ have IDs ***at least*** p.

In other words, the returned index k is the first index of a package on the FlipMaster's conveyor belt that is at least p. If all packages have IDs less than p, then $k = j$, and if all packages have IDs at least p, then $k = i$.

Example: For the sequence of package IDs $S = [4, 9, 330, 1, 2, 5, 8, 16]$, a valid value of S after `flartition(f,2,7,6)` is $[4, 9, \underline{2, 1, 5, 330, 8}, 16]$, where the underlined elements are those in the affected subsequence. The return value of the call is 5, which is the index of the first element (330) at least the pivot value of 6. The orderings of the elements less than/at least the pivot value 6 are arbitrary.

[Hint: Split the problem into two halves, conquer with recursion, then combine their results into a solution. The median index of the i-th through $(j-1)$-th packages can be computed as $(i + j)/2$, rounded down.]

Assume that `peek(i)` takes $O(1)$ time and `flip(i,j)` takes $O(\max\{1, j-i\})$ time. The empirical running time of your solution will be compared to an algorithm with worst-case asymptotic running time $O(n \log n)$ for calls of the form `flartition(f,0,n,p)` on a FlipMaster f with a conveyor belt of n packages.

Language-specific details follow. You can use whichever of Python or Java you prefer. You will receive automatic feedback when submitting, and you can resubmit as many times as you like up to the deadline.

[4]Note that the j-th is not included in the flip.

- **Python.** Download the template file `flartition.py` and FlipMaster class file `flipmaster.py` from Canvas. Complete the method `flartition.py`, **without ever directly accessing the `_c` or `_s` variables of a FlipMaster**. Only its methods should be used. Note that the number of packages on the FlipMaster's conveyor belt can be accessed with `len()`. See `flipmaster.py` for all of its methods.

  You should submit only `flartition.py` to the Gradescope item "Homework 6 (Python)" with the complete function `flartition`, a top-level function that takes as input a FlipMaster object, two indices, and a positive integer. The function returns an integer k, as defined above. The function header is:

  ```
  def flartition(fm:FlipMaster, i:int, j:int, p:int):  -> int
  ```

- **Java.** Download the template file `flartition.py` and FlipMaster class file `flipmaster.py` from Canvas. You should submit only `Flartition.java` to the Gradescope item "Homework 6 (Java)." Note that the number of packages on the FlipMaster's conveyor belt can be accessed with `.length()`. See `FlipMaster.java` for all of its methods.

  You should submit only `Flartition.java` to the Gradescope item "Homework 6 (Java)" with the complete method `flartition`, a top-level method that takes as input a FlipMaster object, two indices, and a positive integer. The method returns an integer k, as defined above. The method header is:

  ```
  public int flartition(int[] s, int i, int j, int p);
  ```

Solution 4. We briefly describe the high-level idea of the recursive case. Suppose we call FLARTITION on the left and right halves of the given array, with the given pivot value. Let k_L, k_R be the returned indices for the first and second halves, respectively. By correctness of the recursive calls, the left and right halves are correctly partitioned by the pivot p. Thus, it suffices to move all elements in the left half that are at least p with those in the right half that are less than p. These elements all lie in the subarray between indices k_L and $k_R - 1$, so a single FLIP operation suffices. After this flip, the array is correctly partitioned by pivot p. The runtime is $T(n) = 2T(n/2) + O(n)$ which solves to $O(n \log n)$, as desired.