Instructions. Limited collaboration is allowed while solving problems, but you must write solutions yourself. List collaborators on your submission.

You can choose which problems to complete, but must submit at least one problem per assignment. See the course page for information about how challenge problems are graded and contribute to your homework grade. Since you don’t need to complete every problem, you are encouraged to focus your efforts on producing high-quality solutions to the problems you feel confident about. There is no benefit to guessing or writing vague answers.

If you are asked to design an algorithm, please (a) give a precise description of your algorithm using either pseudocode or language, (b) explain the intuition of the algorithm, (c) justify the correctness of the algorithm; give a proof if needed, (d) state the running time of your algorithm, (e) justify the running-time analysis.

Submissions. Please submit a PDF file. You may submit a scanned handwritten document, but a typed submission is preferred. Please assign pages to questions in Gradescope.

1 Challenge Problems

Problem 1. Array medians. There are two sorted arrays $A$ and $B$ of $n$ numbers. You want to find the median of the overall set of $2n$ numbers, which we will define as the $n$th smallest value — you may assume all $2n$ numbers are distinct. Give an algorithm that finds the median in $O(\log n)$ time.

Solution: The high-level approach is similar to binary search: we will make a constant-time query to reduce the problem by half, and then recurse on the smaller problem. A key observation is that we can safely eliminate some numbers from consideration if we eliminate an equal number that are smaller than the median and greater than the median; then the median of the remaining numbers is the same as the median of the original numbers. The difficulty is doing this when the numbers are divided into two arrays $A$ and $B$.

We can query the median in each array (which takes constant time since they are sorted), compare them, and then eliminate the smallest half of the array with the smaller median and the largest half of the array with the larger median.

Assume for now $n$ is even and let $k = n/2$, so $k$ is the index of the median of each array. Let $m_A = A[k]$ be the median of $A$ and let $m_B = B[k]$ be the median of $B$. If $m_A < m_B$, then the overall median is between $m_A$ and $m_B$. Specifically, if $m$ is the overall median, then $m_A < m \leq m_B$. To see this, note that:

- There are $n/2$ entries in $A$ that are greater than $m_A$. There are an additional $n/2$ entries in $B$ that are greater than $m_B$, and since $m_B > m_A$, these are also greater than $m_A$. The median element of $B$ is also greater than $m_A$. Therefore, there are at least $n + 1$ numbers that are greater than $m_A$, so the overall median $m$ is greater than $m_A$. (To understand this: if we created a sorted list of all of the numbers from $A$ and $B$, the median $m$ would appear at position $n$; if there are $n + 1$ numbers greater than $m_A$, then $m_A$ appears at some index less than $n$, so must be less than $m$.)

- There are $n/2$ entries in database $A$ that are less than or equal to $m_A$, and since $m_A < m_B$, these are all less than $m_B$. There are $n/2 - 1$ entries in database $B$ that are less than $m_B$. Therefore, there are at least $n - 1$ entries that are less than $m_B$, so the median is less than or equal to $m_B$.

By the argument above, the first $n/2$ values of $A$ are less than $m_A$, which is less than the median. Similarly, the last $n/2$ values of $B$ are greater than $m_B$, which is greater than or equal to the median. So, we can eliminate all of these values, and, because we eliminate an equal number of elements that are greater than
the median and less than the median, the median of the remaining values is the same as the median of the original values. So it suffices to (recursively) find the median of the smaller set of values.

The recursive principle is therefore to recurse on the second half of $A$ and on the first half of $B$. We need to do this “in place”, since it would take $\Omega(n)$ time to copy the arrays and we want to use $O(\log n)$ time. This is accomplished by the following algorithm, which also correctly handles the case where $n$ is odd.

The algorithm $\texttt{median}(n, a, b)$ finds the median of $\{A[a + 1], \ldots, A[a + n], B[b + 1], \ldots, B[b + n]\}$. Here, $a$ and $b$ (initially 0) are the starting offsets in databases $A$ and $B$, and $n$ is the number of elements left to consider in each database (initially set to $n \leftarrow \text{length}(A)$).

$$\text{median}(n, a, b)$$

if $n = 1$ then return $\min(A[a + 1], B[b + 1])$  \hspace{1cm} \text{\triangleright} \text{Base case}$

end if

$k = \lceil n/2 \rceil$

$m_A = A[a + k]$

$m_B = B[b + k]$

if $m_A \leq m_B$ then return $\text{median}(k, a + [n/2], b)$ \hspace{1cm} \text{\triangleright} \text{median of remaining elements of } A$

else return $\text{median}(k, a, b + [n/2])$ \hspace{1cm} \text{\triangleright} \text{median of remaining elements of } B$

end if

Each recursive step divides the input size in half and makes one recursive call. There are $\lceil \log n \rceil$ levels of recursion, and each level makes two queries. The overall number of queries is therefore $O(\log n)$.

Alternate solution (sketch): A student described a clever alternate solution. It’s actually the same algorithm but with a different interpretation and implementation. The idea is to maintain two indices $a$ and $b$ and adjust them until either $A[a]$ or $B[b]$ is the median. Assume for now that $n$ is a power of 2. We’ll set $a = b = n/2$ and maintain the invariant that $a + b = n$. The invariant implies that there are exactly $n$ values in the set $L = \{A[1], \ldots, A[a], B[1], \ldots, B[b]\}$ and exactly $n$ values in the set $R = \{A[a + 1], \ldots, A[n], B[b + 1], \ldots, B[n]\}$. Furthermore, if we can find $a$ and $b$ such that $A[a]$ and $B[b]$ are “equal”, then all values in $R$ are greater than all values in $L$, so $L$ contains the $n$ smallest values, and the largest value in $L$ is the median. But the largest value in $L$ is just the larger of $A[a]$ and $B[b]$, so we could just check these two values and return the larger. Also, we don’t really need $A[a]$ and $B[b]$ to be equal, we just need both $A[a + 1]$ and $B[b + 1]$ to be larger than both of $A[a]$ and $B[b]$ so that all numbers in $R$ are greater than all numbers in $L$. We can adjust $a$ and $b$ in the pattern of a binary search until this is true. This gives an algorithm like the following:

$$\text{median}(A, B)$$

$n \leftarrow \text{length}(A)$

Set $a = b = n/2$

while it is not true that $A[a + 1]$ and $B[b + 1]$ are both greater than or equal to $A[a]$ and $B[b]$ do

if $A[a] < B[b]$ then

$a \leftarrow a + n/2$

$b \leftarrow b - n/2$

else

$a \leftarrow a - n/2$

$b \leftarrow b + n/2$

end if

$n \leftarrow n/2$

end while

return The larger of $A[a]$ and $B[b]$

This algorithm works if the original lists have size that is a power of 2 and can be modified by rounding for the case when $n$ could become odd (details omitted).

Problem 2. Butterflies (Again?). You have collected $n$ butterfly specimens of different species. You want to know if there is a single species that accounts for more than half of the specimens. You don’t know butterflies well enough to name the species of any single specimen, but you can carefully compare any two specimens and judge (correctly) if they are from the same species or not. Design an algorithm that returns
“true” if there are more than \( n/2 \) specimens from one species, and returns “false” otherwise, using only \( O(n \log n) \) pairwise comparisons.

**Solution:** Split the specimens into two equal-size piles \( A \) and \( B \). Assume for simplicity that \( n \) is a power of two. Suppose more than \( n/2 \) of all specimens are of species \( S \). Then more than half of the specimens of *either* \( A \) or \( B \) (or possibly both) will also be of species \( S \). So, recursively test for a majority species in \( A \) and \( B \), and assume that the recursive call returns a specimen from the majority species if one exists. If either recursive call returns a majority specimen, take that specimen and compare it to each other specimen in the overall set. Return “true” if it matches more than \( n/2 \).

Running time: this procedure makes two recursive calls to problems of half the size, and then does \( O(n) \) work outside the recurrence, so its running time satisfies the recurrence \( T(n) \leq 2T(n/2) + O(n) \), which solve to \( T(n) = O(n \log n) \).

**Problem 3.** Shelving books. Books numbered 1 to \( n \) are to be shelved in order on consecutive bookshelves at the library. Book \( i \) has height \( h_i \). Each shelf can hold up to \( L \) books (but you are allowed to place fewer) and its height can be adjusted to accommodate the tallest book placed on it. However, you want to keep the shelf heights as small as possible to use space efficiently — give an algorithm to find the smallest sum of shelf heights required to shelve all \( n \) books.

**Solution:** Define \( \text{OPT}(i) \) to be the minimum cost to shelve books starting with book \( i \), i.e., to shelve the books \( i, i+1, \ldots, n \). The original problem is to compute \( \text{OPT}(1) \). The recursive principle is to first determine the number of books \( j \) to place on the first shelf. If we place \( j \) books on the first shelf, the cost would be:

\[
\text{OPT}(i) = \min_{1 \leq j \leq L} \left\{ \frac{\max\{h_i, \ldots, h_{i+j-1}\}}{\text{height of first shelf}} + \frac{\text{OPT}(j+1)}{\text{total height of remaining shelves}} \right\}
\]

The first term is the height of the first shelf, and the second term is the total height required to shelve the remaining books starting from \( j + 1 \). We need to search over all possible values of \( j \), and generalize our recursive principle to start at an arbitrary book \( i \). This gives the recurrence:

\[
\text{OPT}(i) = \min_{1 \leq j \leq L} \left\{ \max\{h_i, \ldots, h_{i+j-1}\} + \text{OPT}(i+j) \right\}
\]

The base case corresponds to having only the \( n \)th book remaining: we could also use the base case \( \text{OPT}(n+1) = 0 \). To turn this recurrence into a dynamic programming algorithm we construct a memoization array \( M \) with \( j \) ranging from 1 to \( n \), and fill it starting with the base case. This means iterating from the end of \( M \) to the beginning.

1. Initialize arrays \( M[1 \ldots n] \) and num-books-on-first-shelf[\( i \)]
2. Set \( M[n] = h_n \)
3. for \( i = n - 1 \) down to 1 do
   4. \( M[i] \leftarrow \infty \)
   5. first-shelf-height \( \leftarrow 0 \)
   6. for \( j = 1 \) to \( L \) do
      7. shelf-height \( \leftarrow \max(\text{first-shelf-height}, h_{i+j-1}) \)
      8. if \( \text{first-shelf-height} + M[i+j] < M[i] \) then
         9. \( M[i] \leftarrow \text{first-shelf-height} + M[i+j] \)
         10. num-books-on-first-shelf[i] \( \leftarrow j \)
      11. end if
   12. end for
13. end for
14. return \( M[1] \)

The running time is \( \Theta(nL) \). This algorithm computes the height of the first shelf, which is the maximum of \( \{ h_i, \ldots, h_{i+j-1} \} \) as a running maximum. A direct translation of the recurrence might use an additional nested loop to compute this maximum for each value of \( j \), and run in time \( \Theta(nL^2) \); it would also get full credit.

The array num-books-on-first-shelf stores how many books to place on the first shelf when starting from any book index \( i \). To reconstruct the solution and shelf the books, trace through this array starting from index 1:
Set $i = 1$

while $i < n$ do

    $j \leftarrow \text{num-books-on-first-shelf}[i]$

    Place books $i, \ldots, i + j - 1$ on the next shelf

    $i \leftarrow i + j$

end while

Problem 4. Chicken Wings. The image in Figure 1 is a real restaurant menu. The goal of this problem is to find the cheapest way to buy $V$ chicken wings for some integer $V$ given a menu like this one. Assume the menu is given as a list of $n$ menu items $(v_1, w_1), (v_2, w_2), \ldots, (v_n, w_n)$, where $w_i$ is the price to buy $v_i$ wings and $v_i$ and $w_i$ are both integers. In the example, assuming costs are computed in cents, we would have:

$$(v_1, w_1) = (4, 455)$$
$$(v_2, w_2) = (5, 570)$$
$$(v_3, w_3) = (6, 680)$$

You are allowed to choose any combination of orders whose quantities add up to $V$, including ordering the same quantity multiple times. You can assume there is always some combination of orders to buy exactly $V$ wings.

(a) There is a natural greedy algorithm where you first buy the largest quantity $v_i$ such that $v_i \leq V$, and then repeat on the remaining $V - v_i$ wings. Show that this algorithm is not optimal for the menu shown in Figure 1.

(b) Write a dynamic programming algorithm to find the cost of the cheapest set of orders to buy exactly $V$ wings.
(c) Modify your algorithm to also return the set of orders you could make to achieve the smallest cost.

**Solution:** Let $OPT(v)$ be the minimum cost for a combination of orders to buy $v$ chicken wings. A combination of orders for a total of $v$ wings consists of one order for $v_i$ wings (at cost $w_i$), for some $i$, plus a combination of orders for $v - v_i$ wings. The recurrence is

\[
OPT(0) = 0 \\
OPT(v) = \min_{i: v_i \leq v} \{ w_i + OPT(v - v_i) \}
\]

The algorithm will initialize an array $M[0..V]$ and then use the recurrence above to fill the array, for $v$ ranging from 0 to $n$. The time to fill each entry is $O(n)$. The total running time is $O(nV)$.

Here is the detailed algorithm:

To find the optimal value:
1. Initialize an array $M[0..V]$ and an array first-order[0..V]
2. Set $M[0] = 0$
3. for $v = 1$ to $V$ do
   - $M[v] = \infty$
   - for $i = 1$ to $n$ do
     - if $v_i \leq v$ then
       - $M[v] = \min\{M[v], w_i + M[v - v_i]\}$
     - end if
   - end for
4. end for
5. Return $M[V]$ as the cheapest cost to buy $V$ wings

To return the set of orders:
1. Set $v = V$
2. Initialize the set $O$ to be empty
3. while $v > 0$ do
   - Let $i =$ first-order[$v$]
   - Set $O = O \cup \{i\}$
   - Set $v = v - v_i$
4. end while
5. Return $O$ as the optimal set of orders

An alternate correct solution uses the following recurrence. Let $OPT(v, j)$ be the minimum cost to buy $v$ chicken wings using only orders placed from the first $j$ menu items. We can consider the binary choice: the optimal set of orders does not include the $j$th menu item, or it includes at least one order of the $j$th menu item. This gives the recurrence:

\[
OPT(0, j) = 0 \\
OPT(v, j) = \begin{cases} \min\{OPT(v, j - 1), w_j + OPT(v - v_j, j)\} & \text{if } v_j \leq v \\ OPT(v, j - 1) & \text{if } v_j > v \end{cases}
\]

This can be converted into a dynamic programming algorithm in the usual way. The memoization array $M[0..V, 0..n]$ has $O(nV)$ entries, and there are at most two options on the RHS of the recurrence, so filling each entry takes constant time. The running time will be $O(nV)$.