Learning Goals

During this lab, you will cover:

- Mergesort and Quicksort, and how they relate to the idea of divide and conquer
- Recurrence relations and run time review

General Problem Statement

Given an array of length \( n \), sort it in ascending order (could also be descending). You cannot assume anything about the contents of the array.

Divide and Conquer: An Overview

What does it mean to have a “divide and conquer” algorithm?

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem
- **Conquer** the subproblems by solving them recursively.
- **Combine** the solutions to the subproblems into the solution for the original problem.

How do you recognize situations where “divide and conquer” might work? A natural first question is “Can I break this down into subproblems equivalent to the original problem?” You can then ask “how can I solve these problems and combine them to reach a solution for my original problem?” Usually if you can solve each subproblem and combine them, it involves some sort of recursion. In order to better understand the “divide and conquer” paradigm, we will do an in depth study on a familiar algorithm: Mergesort.

Mergesort and Divide and Conquer

We can apply the principles of Divide and Conquer when thinking about approaching this problem.

- **Divide**: Can we divide this into equivalent subproblems? Yes, we can divide this array into two halves each with \( \frac{n}{2} \) elements. Thus, each is an equivalent subproblem.
- **Conquer**: How can we recursively sort the two halves? That’s easy! Since we already broke it into subproblems, we will recurise using mergesort on the two halves until we hit the base case of a singleton element (we trivially know that a singleton element is sorted).
- **Combine**: Once we have two sorted arrays we can combine them in \( O(n) \) time by interleaving the halves!

**Running time**: We can analyze the run time of mergesort using recurrence relations. We know that the running time of mergesort, which we will denote \( T(n) \), depends on two things. First, we consider the recursive calls. We are constantly splitting the array in halves, so we have a \( 2T\left(\frac{n}{2}\right) \) term in the recurrence relation. Finally, we have to consider the interleaving of arrays, which we claimed earlier was a \( O(n) \) operation. Thus, our running time becomes \( T(n) = 2T\left(\frac{n}{2}\right) + O(n) \).

We will describe how to solve this recurrence relation below.
QuickSort

In quicksort, we first decide on the pivot. This could be the element at any location in the input array. The function Partition accomplishes the following: it places the pivot in the location that it should be in the output and places all elements that are at most the pivot to the left of the pivot and all the elements greater than the pivot to its right. Then we recurse on both parts. The pseudocode for Quicksort is as follows:

```pseudo
QSort(A[lo...hi])
    if hi <= lo then
        return
    else
        pivotIndex = floor((lo+hi)/2) // this could have been any location
        loc = Partition(A, lo, hi, pivotIndex)
        QSort(A[lo...loc-1])
        Qsort(A[loc+1...hi])
```

One possible implementation of the function Partition is as follows:

```pseudo
Partition(A, lo, hi, pIndex)
    pivot = A[pIndex]
    swap(A, pIndex, hi)
    left = lo
    right = hi-1
    while left <= right do
        if A[left] <= pivot then
            left = left + 1
        else
            swap(A, left, right)
            right = right - 1
    swap(A, left, hi)
    return left
```

The worst case running time of the algorithm is given by

\[ T(n) = \begin{cases} 
    T(n-1) + cn & n \geq 2 \\
    1 & \text{otherwise}
\end{cases} \]

Hence the worst case running time of QSort is \( \Theta(n^2) \). An instance in which the quicksort algorithm performs poorly is when the pivot is always the first element in the input array, and the input array is in descending order of its elements.

**Randomized Quicksort**

In the randomized version of quicksort, we pick a pivot uniformly at random from all possibilities. We will now show that the expected number of comparisons made in randomized quicksort is equal to \( 2n \ln n + O(n) = \Theta(n \log n) \).

**Theorem 1.** For any input array of size \( n \), the expected number of comparisons made by randomized quicksort is

\[ 2n \ln n + O(n) = \Theta(n \log n) \]

**Proof.** Let \( y_1, y_2, \ldots, y_n \) be the elements in the input array \( A \) in sorted order. Let \( X \) be the random variable denoting the total number of pair-wise comparisons made between elements of \( A \). Let \( X_{i,j} \) be the random variable denoting the total number of times elements \( y_i \) and \( y_j \) are compared during the algorithm.

We make the following observations:
• Comparisons between elements in the input array are done only in the function \texttt{Partition}.

• There are \( n - 1 \) distinct pivots chosen over the course of the algorithm, and hence \( n - 1 \) calls to \texttt{Partition}.

• Two elements are compared if and only if one of them is a pivot.

Let \( X^k_{i,j} \) be an indicator random variable that is 1 if and only if elements \( y_i \) and \( y_j \) are compared in the \( k \)-th call to \texttt{Partition}. Then we have:

\[
X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j} \quad \text{and} \quad X_{i,j} = \sum_{k=1}^{n-1} X^k_{i,j}
\]

We will now calculate \( E[X_{i,j}] \). By the linearity of expectation, we have

\[
E[X_{i,j}] = \sum_{k=1}^{n-1} E[X^k_{i,j}] = \sum_{k=1}^{n-1} \Pr[X^k_{i,j} = 1]
\]

Let \( t \) be the iteration of the first call to \texttt{Partition} during which one of the elements from \( y_i, y_{i+1}, \ldots, y_j \) is used as the pivot. From our observations above, note that for all times before \( t \), \( y_i \) and \( y_j \) are never compared, so \( X^k_{i,j} = 0 \) for all \( k < t \). If one of \( y_i \) or \( y_j \) is chosen as the \( t \)-th pivot, then \( X^t_{i,j} = 1 \), otherwise \( X^t_{i,j} = 0 \) and \( y_i \) and \( y_j \) will be separated into different sublists and hence will never be compared again. Hence \( X^k_{i,j} = 0 \) for all \( k > t \).

Now, since there are \( j - i + 1 \) elements in the list \( y_i, y_{i+1}, \ldots, y_j \), and the pivot is chosen randomly, the probability that one of \( y_i \) or \( y_j \) is chosen as the pivot is \( \frac{2}{j - i + 1} \). Hence:

\[
E[X_{i,j}] = \Pr[X^t_{i,j} = 1] = \frac{2}{j - i + 1}
\]

Now we use this result and apply the linearity of expectation to get:

\[
E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{i,j}]
\]

\[
= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}
\]

\[
= \sum_{k=2}^{n} \frac{2}{k} (n - k + 1) \quad \text{(see the note below)}
\]

\[
= (n + 1) \sum_{k=2}^{n} \frac{2}{k} - 2(n - 1)
\]

\[
= 2(n + 1) \ln n + c - 4n \quad \text{where } 0 \leq c < 1
\]

\[
= 2n \ln n + O(n)
\]

Here we have used the fact that the harmonic function, \( H(n) = \sum_{k=1}^{n} \frac{1}{k} \) is at most \( \ln n + c \) for some constant \( 0 \leq c < 1 \).
Also, the third equality follows by expanding the sum as
\[
\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \frac{2}{2} + \frac{2}{3} + \ldots + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n}
\]
\[
+ \frac{2}{2} + \frac{2}{3} + \ldots + \frac{2}{n-2} + \frac{2}{n-1}
\]
\[
+ \frac{2}{2} + \frac{2}{3} + \ldots + \frac{2}{n-2}
\]
\[
\vdots
\]
\[
+ \frac{2}{2}
\]
and grouping the columns together.

\[\square\]

Recurrences

As you have seen in class, recurrences are equations that can help us describe the running time of a recursive algorithm. You have thus far seen two different ways of solving recurrences:

| Iteration. | In this method, we expand \( T(n) \) fully by substitution and solve for \( T(n) \) directly. |
| Recursion trees. | In this method, we draw the recursive calls to \( T(n) \) in a tree format and count the amount of work done in each level of the tree. |

Let’s first go through some examples before running through problems.

Example: Method of Iteration

Let’s examine the following recurrence \( T(n) \).

\[
T(n) = \begin{cases} 
T(n-1) + n & n \geq 1 \\
0 & \text{otherwise}
\end{cases}
\]

Using the method of iteration, we expand \( T(n) \) as follows:

\[
T(n) = T(n-1) + n \\
= [T(n-2) + (n-1)] + n \\
= [T(n-3) + (n-2)] + (n-1) + n \\
\vdots \\
= \sum_{i=1}^{n} i \\
= \frac{1}{2}(n+1)(n) = \binom{n+1}{2} = \Theta(n^2)
\]
Example: Method of Recursion Trees

Let’s examine the recurrence of merge sort. For those that are unfamiliar with it by now, the algorithm works by taking an unsorted array, sorting the left and right halves of the array recursively, and then merging the two sorted halves together to end up with the final sorted list. Let $T(n)$ represent the time the algorithm takes for an input of size $n$. Since the two halves are sorted recursively by the same algorithm, but with inputs that are each half the size of the original, each half should take time $T(\frac{n}{2})$. The merging takes linear time. So we can write $T(n) = 2 \cdot T(\frac{n}{2}) + cn$ for some constant $c$. The recursion-tree is shown below.

Note that at the very top level (i.e., the end of the algorithm), it costs $cn$ to merge the two sorted halves of the array. But to get there, we needed to solve the two problems of size $\frac{n}{2}$. Each costs $c \cdot \frac{n}{2}$ to solve. Therefore, across that level, the total cost is $c \cdot \frac{n}{2} + c \cdot \frac{n}{2} = cn$. We can continue this all the way until we get to the very bottom of the tree, which are single elements. Note then that every level ends up costing $cn$. The height of the tree is $\lg n$. Therefore, the total cost is $c \cdot n \lg n = O(n \lg n)$.

To see why the height is $\lg n$, observe that subproblem sizes decrease by a factor of 2 each time we go down one level, we stop when we reach singleton elements. The subproblem size for a node at depth $i$ is $\frac{n}{2^i}$. Thus, the subproblem size hits $n = 1$ when $\frac{n}{2^i} = 1$ or, equivalently, when $i = \lg n$.

Problems

Problem 1

Given an integer array (contains positive and negative values), return the sum of the largest contiguous subarray which has the largest sum.
**Solution**

One way to solve this problem (naive method), is to use two loops. The outer loop runs through the elements in the array, while the inner loop finds the maximum sum given the current outer loop element. If this sum is bigger than the best running maximum, we update and continue through the process. This runs in $O(n^2)$.

A better solution is to apply our knowledge of the Divide & Conquer approach, and see if we can find a more efficient solution to this problem.

One thing that we can intuitively notice is that the optimal sub-sequence either lies in the left half of the array, the right half of the array, or runs along the center of the array and cuts through the middle element. Logically, these are the only three options we have. Thus, we can compute all three of these values and the maximum of them will be our solution!

To do this, we must recursively divide the array into two halves and find the maximum subarray sum in both halves. This can be done easily with two recursive calls. Lastly, we need to efficiently compute the maximum cross sum. This can be done in $O(n)$ time by starting at the middle element and calculating the maximum sum to the left of the median, doing the same with the right and combining the two!

To calculate the run time of our algorithm, we notice that it breaks down the work into two sub problems, each with half the size as input and then checks the cross sum in linear time. Thus, we get the following recurrence: $T(n) = 2T(\frac{n}{2}) + O(n)$.

Does this look familiar? It should! It’s the same recurrence you saw for the running time analysis of MergeSort! This evaluates to $O(n \log n)$.

**Problem 2**

You are given a sorted array of $n$ distinct integers $A[1...n]$. Design an $O(\log n)$ time algorithm that either outputs an index $i$ such that $A[i] = i$ or correctly states that no such index $i$ exists.

**Solution**

**Algorithm 1** Modified Binary Search

```plaintext
1: procedure FINDINDEXMATCHINGELEM(A[...])
2:     $l \leftarrow 0$ ▷ left bound pointer
3:     $r \leftarrow \text{length}(A) - 1$ ▷ right bound pointer
4:   while $l \leq r$ do ▷ ”$\leq”$ takes care of single elem case
5:     $m \leftarrow \lfloor (l + r)/2 \rfloor$ ▷ midpoint
6:     if $A[m] == m$ then
7:         return $m$
8:     else if $A[m] < m$ then
9:         $l \leftarrow m + 1$
10:    else
11:         $r \leftarrow m - 1$
12:    return $-1$
```

**Problem 3**

Solve the following recurrence using induction:

$$T(n) = \begin{cases} 
2T(n-1) + 1 & n \geq 2 \\
1 & \text{otherwise}
\end{cases}$$
Solution

Claim. For $T(n)$ above, $T(n) = 2^n - 1$.

Proof. We prove the claim by performing induction on $n$.
  
  I.H. Let $P(k)$ be the proposition that $T(k) = 2^k - 1$ for some $k \geq 1$.
  
  B.C. $P(1)$ holds, as $T(1) = 1 = 2^1 - 1$.
  
  I.S. We want to show that the claim holds for $k + 1$. 
  
  \[
  T(k + 1) = 2T(k) + 1 \\
  = 2(2^k - 1) + 1 \\
  = 2^{k+1} - 1
  \]