Learning Goals

During this lab, you will cover:

- Insertion sort and selection sort
- Mergesort and how it relates to the idea of divide and conquer
- Recurrence relations and run time

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.

Basic Sorting Algorithms: Insertion Sort and Selection Sort

Insertion Sort

Insertion sort is a simple solution to sorting. It is useful for smaller data sets and can be implemented very easily. Insertion sort is a stable sort, can be done in-place ($O(1)$ space), and can be considered an “online” algorithm, meaning it can sort a list as it receives it!

Recall:

1. We recall the invariant in place: We will keep a sorted side of the array and have some function to insert a value into the sorted sequence at the beginning of the array. Essentially, it will begin at the end of the sorted sequence and shift each element one place to the right until a suitable position is found for the element.

2. We begin at the leftmost element in the array, and call insert in order to position each element in its proper position in the array. The ordered sequence is built up from the left to the right.

Refer to the java implementation below:

```java
for (int i = 1; i < A.length; i++) {
    int j = i;
    while (j > 0 && A[j-1] > A[j]) {
        swap(A[j], A[j - 1]);
        j--;
    }
}
```

Running time: We can see that the outer loop executes $n - 1$ times. In the worst case, the inner while loop makes at most $n$ swaps. Thus, we can see that the worst case running time for this algorithm is $O(n^2)$. 


Selection Sort

Selection sort is another simple algorithm to sorting. It has many of the same characteristics of insertion sort: can be done in place ($O(1)$ space) and can be implemented very easily. However, given the same inputs, selection sort generally performs worse than insertion sort.

Recall:

1. Selection sort divides the input list into two parts: it holds the invariant that we keep a sorted side of the array, which is built up from left to right at the beginning of the array, and maintains a subarray of items remaining to be sorted that occupy the rest of the array.

2. The algorithm proceeds by finding the smallest element in the unsorted portion of the array and exchanges it with the leftmost unsorted element (putting the smallest element in sorted order), and moving the subarray boundaries one element to the right.

Refer to the java implementation below:

```java
for (j = 0; j < a.length - 1; j++) {
    int min = j;
    for (i = j + 1; i < a.length; i++) {
        if (a[i] < a[min]) {
            min = i;
        }
    }
    if (min != j) {
        swap(a[j], a[min]);
    }
}
```

Running time: We can see that the outer loop executes $n - 1$ times. We also notice that the inner loop checks at most $n - 1$ elements. Thus, we can see that the worst case running time for this algorithm is $O(n^2)$.

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 recurse 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.

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:

\[
T(n) = T(n - 1) + n \\
= [T(n - 2) + (n - 1)] + n \\
= [[T(n - 3) + (n - 2)] + (n - 1)] + n \\
\vdots \\
= \sum_{i=1}^{n} i \\
= \frac{1}{2}(n + 1)(n) = \left( \frac{n + 1}{2} \right) = \Theta(n^2)
\]

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 \( T \left( \frac{n}{2} \right) \). The merging takes linear time. So we can write \( T(n) = 2 \cdot T \left( \frac{n}{2} \right) + cn \) 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 \( \log n \). Therefore, the total cost is \( c \cdot n \log n = O(n \log n) \).

To see why the height is \( \log n \), 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 \( i = \log n \).

Problems

Problem 1

Assume you have an array of \( n \) comparable elements. Design an algorithm that removes all duplicates in \( O(n \log n) \) time. You may not use a Hash Map or a BST to solve this question.

Solution

When we do the merge step we check to see if \( a[i] = b[j] \), if it is, then the first time we encounter this, we add the number to the final list. Let us temporarily store this number as \( x \). We then increment both \( i \) and \( j \) until \( a[i] \neq x \) and \( b[j] \neq x \).

Alternate Solution

Run mergesort on the list. Then go through the sorted list again, and remove any duplicates by adding only one of each element to the final list.
Problem 2

Implement a `merge()` function that might be utilized in mergesort. Assume you are given the correct length of the output array parameter, `merged`.

```java
private static void merge(int[] A, int[] B, int[] merged) {...}
```

Solution

```java
private static void merge(int[] A, int[] B, int[] merged)
{
    int i = 0; // pointer to position in A
    int j = 0; // pointer to position in B
    int k = 0; // pointer to position in merged

    while (i < A.length && j < B.length) {
        if (A[i] < B[j]) {
            merged[k++] = A[i++];
        } else {
            merged[k++] = B[j++];
        }
    }

    // if A or B are not the same size we will take care of this
    // equivalently, we can use Arrays.copy() function
    while (i < A.length) {
        merged[k++] = A[i++];
    }

    while (j < B.length) {
        merged[k++] = B[j++];
    }

    // merged is now sorted
}
```

Problem 3

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: \( T(n) = 2T\left(\frac{n}{2}\right) + O(n) \).

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 4**

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**

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: \( m \leftarrow (l + r)/2 \) \> 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 5**

Solve the following recurrence:

\[
T(n) = \begin{cases} 
2T\left(\frac{n}{2}\right) + n^2 & n \geq 1 \\
1 & \text{otherwise}
\end{cases}
\]

**Solution**

We first solve by iteration. First, we may assume that \( n \) is some power of 2 such that \( n = 2^k \implies k = \log n \).

\[
T(n) = 2T\left(\frac{n}{2}\right) + n^2 = 2 \left[ 2T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)^2 \right] + n^2
\]

\[
= 2 \left[ 2 \left[ 2T\left(\frac{n}{2^3}\right) + \left(\frac{n}{2^2}\right)^2 \right] + \left(\frac{n}{2}\right)^2 \right] + n^2
\]
We see that if we continue to expand the recurrence, we will end up with a $2^k T\left(\frac{n}{2^k}\right) = 2^k$ term in the base case. Fully expanded, we get:

$$T(n) = 2^k + n^2 + 2 \left(\frac{n}{2}\right)^2 + 2^2 \left(\frac{n}{2^2}\right)^2 + \cdots + 2^{k-1} \left(\frac{n}{2^{k-1}}\right)^2$$

$$= 2^k + n^2 + \frac{n^2}{2} + \frac{n^2}{2^2} + \cdots + \frac{n^2}{2^{k-1}}$$

$$= 2^k + n^2 \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i$$

$$= 2^k + n^2 \frac{1 - \left(\frac{1}{2}\right)^k}{1 - \frac{1}{2}}$$

geometric series

$$= 2^{\lg n} + 2n^2 \left[1 - \frac{1}{n}\right]$$

substituting $k = \lg n$

$$= n + 2n^2 - 2n = \Theta(n^2)$$

That’s a lot of tedious algebra, but it gets the job done. Now, let’s try a different approach—we’ll expand the recurrence using a recursion tree:

On the right side, we can see the total amount of work done at each level of the tree. Our sum becomes immediately apparent without all the initial algebraic soup! I would argue that this approach helps form a much more intuitive understanding of the problem.

**Problem 6**

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 $k + 1$.

\[
T(k + 1) = 2T(k) + 1 = 2(2^k - 1) + 1 = 2^{k+1} - 1
\]

$\square$