Lab03

Here are my responses for Lab 3! This is in PDF format because I'm trying out a new markdown editor called Obsidian.

1. Binary Search

Part A: Correctness Proof

Invariant: If the target value exists in the array, it exists in the slice passed to this call.

Initialization:
The entire array is passed, so if the target value exists in the array, it exists in the slice given.

Maintainence:
Because the array is sorted, if the target value is less than the midpoint value of any given slice, then the target value must exist before the midpoint of that slice. Similarly, if the target value is greater than the midpoint value of any given slice, then the target value must exist after the midpoint of that slice.

Because the slice passed to the function must contain the target value if the target value exists, we know that the upper or lower half of the given slice (determined as explained above) must also contain the target value, if it exists.

Termination:
Binary search cuts the size of the slice in half at each recursion, so eventually the size of the slice will be one. Since the target value always exists in the given slice, if it exists in the array, then it must be the one value in that slice of length 1.
Since the algorithm terminates when the slice length is 1, the slice will never be empty, so the slice must always contain the target value.

Part B: Time Complexity

At each recursion, the list length to search gets multiplied by $\frac{1}{2}$, and the algorithm terminates when the list length is 1. Each recursion is constant time, since all that happens is a list lookup, numerical comparison, and single function call. So the overall time complexity is $O(\lg n)$

2. Bubble Sort

Code, for reference (copied from assignment sheet):

```
def bubble_sort(A)
    n = len(A)
        for i = 1 to n
                for j = 1 to n-i
                        if A[j] > A[j + 1]
                                swap A[j] and A[j + 1]
```

Part A: Inner loop

Invariant: `A[j+1]` is guaranteed to be the largest element in `A[1:j+1]`.

Initialization: the invariant is true at the beginning, since `j` starts at 1,

Maintainence: for each iteration of the loop, `A[j]` is moved up to `A[j+1]` if it is greater than `A[j+1]`

Termination: The invariant is true at the beginning, so it must be true at subsequent steps. The loop terminates when `j` is `n-i`, which is always in bounds, so the invariant is true at the end of the loop.

Part B: Correctness Proof

Loop invariant: `A[n-i:n]` is sorted and contains the `i` largest elements of `A`.

Initialization: at the beginning, `i` is 0, so the length of `A[n-i:n]` is zero. This means that it contains the 0 largest elements of A and is sorted.

Maintainence: See part A; after the inner loop terminates, `A[j+1]` is the greatest element in the unsorted portion of the array (`A[1:j+1]`). Since `j` is `n-i` when the loop terminates, `A[n-i]` must be larger than all elements before it, yet smaller than all elements after it, since the inner loop terminates before reaching any larger values, and since `A[n-i:n]` is already sorted with the largest elements of `A`. This means that it is in the correct sorted location, and is the largest element of `A` other than the elements that have already been sorted.

Termination: The invariant is true at the beginning, so it must be true at each subsequent step. At the end of the algorithm, `i` is `n+1`, and since $n - (n - 1) = 1$, `A[n-(n+1):n]` is `A[1:n]`, so the whole list is sorted.

Part C: Time Complexity

Bubblesort's worst-case time complexity is $O(n^2)$, because for each element in the list, it needs to iterate through the rest of the list to put it into the correct sorted location.

$$\begin{aligned} T(n) &= n + (n-1) + (n-2) + \ldots + (n-(n-1)) \\ &= n\left(\frac{n + (n-(n-1))}{2}\right) \\ &= n\left(\frac{n+1}{2}\right) \\ &= \frac{n^2}{2} + \frac{n}{2} \\ &= O(n^2) \end{aligned}$$

Insertion sort is also $O(n^2)$, but bubble sort is much slower, because there are far more actual swaps required for bubble sort, since each iteration must check every single item in the unsorted list, no matter what.

3. Inversions

Part A: Inversions of $[2 \quad 3 \quad 8 \quad 6 \quad 1]$

The five inversions are: $\{(0,4),(1,4),(2,3),(2,4),(3,4)\}$.
These were found manually with the mental equivalent of two nested for loops, essentially checking all pairs of indices.

Part B: Maximizing inversions

The array A with elements $\{1 \quad 2 \quad \ldots \quad n\}$ with the most inversions is $[n \quad n-1 \quad \ldots \quad 2 \quad 1]$.
This is because for every i, all $j > i$ represent a valid inversion. This maximizes the number of inversions because there are no other possible pairs of numbers which satisfy the requirement of inversions that $i < j$.

Part C: Connection to insertion sort

The number of inversions that exist for a given array is the same as the number of elements insertion sort will need to insert. For each inversion, insertion sort will need to swap $j - i$ times to get `A[j]` into its correct position.
This is true because the entire premise of insertion sort is that it looks through the array for inversions then corrects the ones it finds.

Part D: Calculating the number of inversions

Here is the algorithm. It just counts how many times things are out of order when merging. For example, when merging $[10 \quad 20 \quad 30]$ and $[15 \quad 25 \quad 35]$, we know that the original order was $[10 \quad 20 \quad 30 \quad 15 \quad 25 \quad 35]$, so whenever we insert something from the second array into the merged array before the end of the first array, we know there had to have been an inversion.
In this case, the merged array is $[10 \quad 15 \quad 20 \quad 25 \quad 30 \quad 35]$, so there were two inversions, since the 15 and 25 had to be inserted early.

Here is the pseudocode/almost python:

```
def count_inversions(arr)
        _, inversions = partition(arr)
        return inversions

def partition(arr)
        if len(arr) <= 1
                return arr, 0
        mid = len(arr)//2
        (sorted_left, l_inversions) = partition(arr[:mid])
        (sorted_right, r_inversions) = partition(arr[mid:])
        return merge(sorted_left, sorted_right, l_inversions+r_inversions)

def merge(left, right, inversions)
        inversions_total = inversions
        merged = []
        (lptr, rptr) = (0, 0)
        while lptr<len(left) and rptr<len(right)
                if left[lptr]>right[rptr]
                        merged.append(right[rptr])
                        rptr++
                        inversions_total++
                else:
                        merged.append(left[lptr])
                        lptr++
        return merged, inversions
```

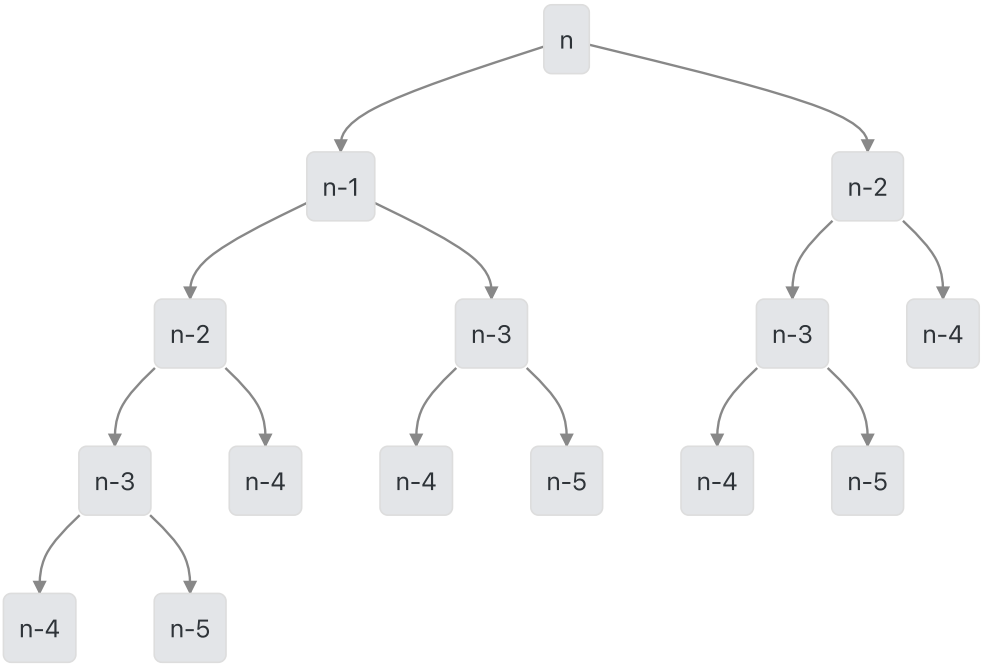
4. Fibonacci

Part A: Naive implementation

Recurrence relation and time complexity:

$$T(n) = \begin{cases} O(1) & x \leq 1 \\ T(n-1) + T(n-2) & x > 1 \end{cases} \qquad T(n) = O(2^n)$$

The time to divide and combine are both constant, so all that's left are the subproblems. Each node creates two other nodes, so the width roughly doubles for every level down. Since there are between n and $\frac{n}{2}$ levels, the runtime is somewhere between 2^n and $2^{\frac{n}{2}}$, which generalizes to $O(2^n)$. This is visible in the recurrence tree. Here it is when $n = 5$:

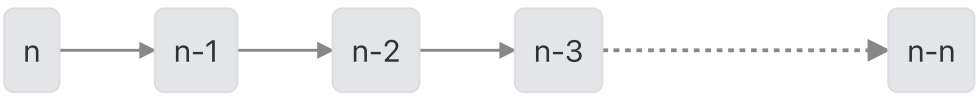


Part B: Improved implementation

Recurrence relation and time complexity:

$$T(n) = \begin{cases} O(1) & x \leq 1 \\ T(n-1) & x > 1 \end{cases} \qquad T(n) = O(n)$$

The time is just $O(n)$ because each recursion just calls one more.
Recurrence tree (sideways to save some space):



5. Permutation Sort

Code:

```
def permutationSort(arr):
    while True:
        # shuffle
        for i in range(len(arr)):
            randidx = randint(0, i)
            arr[i], arr[randidx] = arr[randidx], arr[i]

        # check if it's sorted
        for i in range(len(arr)-1):
            if arr[i]>arr[i+1]: break
        else:
            return True
```

Performance graph: