CSCI211: ALGORITHM DESIGN

JANEET BAJRACHARYA

1. Divide and Conquer Algorithms:

1.1. Find Max and Min:

- Given a unsorted array, we can find the max and min element in $\Theta(n)$ time.

```python
def minMax(a):
    n = len(a)
    if n == 1:
        return a[0], a[0]
    m = n // 2
    min1, max1 = minMax(a[0:m])
    min2, max2 = minMax(a[m:])
    return min(min1, min2), max(max1, max2)
```

- You can use this to find the Max-Difference. Just $\max - \min$ from this result.
- The Recurrence for this: $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + \Theta(1)$

1.2. Finding the r th smallest item:

1.2.1. Quick-Select:

- Input$(A[1...n], r)$

1. Pick some random pivot a_p
2. Call Partition such that a_p is correctly placed at index p such that all elements left are smaller or equal and all elements right are greater.
3. If p is r then we are done
4. If $p < r$, recurse on the right half $A[p+1...n]$
5. If $p > r$, recurse on the left half A$[1...p]$

1.3. Find Median:

1.4. Peasant Multiply:

```
PeasantMultiply(x, y):
  if x = 0
      return 0
  else
      x′ ← ⌊x/2⌋
      y′ ← y + y
      prod ← PeasantMultiply(x′, y′)    ⟨⟨Recurse!⟩⟩
      if x is odd
          prod ← prod + y
      return prod
```

1.5. Hanoi:

```
HANOI(n, src, dst, tmp):
  if n > 0
      HANOI(n − 1, src, tmp, dst)    ⟨⟨Recurse!⟩⟩
      move disk n from src to dst
      HANOI(n − 1, tmp, dst, src)    ⟨⟨Recurse!⟩⟩
```

1.6. Merge Sort/ Count Inversions:

```python
def mergeSort(arr, count):
    def merge(arr1, arr2):
        if len(arr1) == 0:
            return arr2
        elif len(arr2) == 0:
            return arr1
        elif arr1[0] <= arr2[0]:
            return [arr1[0]] + merge(arr1[1:], arr2)
        else:
            count[0] += len(arr1)
            return [arr2[0]] + merge(arr1, arr2[1:])

    if len(arr) < 2:
        return arr
    else:
        h = len(arr) // 2
        return merge(mergeSort(arr[:h], count), mergeSort(arr[h:], count))


def inversionCount(arr):
    count = 0
    for i in range(len(arr) - 1):
        for j in range(i + 1, len(arr)):
            if arr[i] > arr[j]:
                count += 1
    print(f"Brute Force: {count}")
```

1.7. Quick Sort:

```
QuickSort(A[1..n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← Partition(A, p)
      QuickSort(A[1..r − 1])    ⟨⟨Recurse!⟩⟩
      QuickSort(A[r + 1..n])    ⟨⟨Recurse!⟩⟩
```

```
Partition(A[1..n], p):
  swap A[p] ↔ A[n]
  ℓ ← 0                ⟨⟨#items < pivot⟩⟩
  for i ← 1 to n − 1
      if A[i] < A[n]
          ℓ ← ℓ + 1
          swap A[ℓ] ↔ A[i]
  swap A[n] ↔ A[ℓ + 1]
  return ℓ + 1
```

Figure 1.8. Quicksort

```python
def quickSort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[0]

    left = quickSort([x for x in arr[1:] if x <= pivot])
    right = quickSort([x for x in arr[1:] if x > pivot])

    return left + [pivot] + right
```

2. Master's Theorem:

For Recurrences of the Form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is asymptotically positive. The three cases are:

1. If $f(n) = O\big(n^{\log_b(a-\varepsilon)}\big)$ for some constant $\varepsilon > 0$, then $T(n) = \Theta\big(n^{\log_b a}\big)$
2. If $f(n) = \Theta\big(n^{\log_b a}\big)$, then $T(n) = \Theta\big(n^{\log_b a} \log n\big)$
3. If $f(n) = \Omega\big(n^{\log_b a+\varepsilon}\big)$ for some constant $\varepsilon > 0$, and if $af\big(\frac{n}{b}\big) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

2.1. Examples:

1. $T(n) = 9T\big(\frac{n}{3}\big) + n$. Since $n < n^{\log_3 9} = n^2$, $T(n) = \Theta\big(n^2\big)$
2. $T(n) = T\big(\frac{2n}{3}\big) + 1$. Since $1 = n^{\log_{\frac{2}{3}} 1} = n^0 = 1$, $T(n) = \Theta\big(n^0 \log n\big) = \Theta(\log n)$
3. $T(n) = 3T\big(\frac{n}{4}\big) + n \log n$. Since $n \log n > n^{\log_4 3}$, we claim that $f(n) = \Omega\big(n^{\log_4 3+\varepsilon}\big)$ and $3\big(\frac{n}{4}\log\big(\frac{n}{4}\big)\big) \leq \big(\frac{3}{4}\big)n \log n \leq cn \log n$. So $c = \frac{3}{4}$ as $\frac{3}{4} < 1$. Therefore, $T(n) = \Theta(n \log n)$
4. $T(n) = 2T\big(\frac{n}{2}\big) + n \log n$. Since $n \log n > n^{\log_2 2} = n^1$. We know that $f(n) = \Omega(n)$. Now $2\frac{n}{2}\log\big(\frac{n}{2}\big) \leq n \log(n) \leq cn \log n$. This does not follow Master Theorem.

2.2. Other Examples:

1. $T(n) = 2T(\frac{n}{2}) + \log n$ is $\Theta(n)$ as the last layer of the work tree has exactly $2^{\log_2(n)}$ nodes multiplied by $\log_2 \frac{n}{2^{\log_2(n)}} = 0$

3. Majority Element:

```
def majority_element(A, left, right):
    # Base case: If there's only one element, it's the majority of itself
    if left == right:
        return A[left]

    mid = (left + right) // 2
    left_majority = majority_element(A, left, mid)
    right_majority = majority_element(A, mid + 1, right)

    # If both halves agree on the majority element, return it
    if left_majority == right_majority:
        return left_majority

    # Otherwise, count occurrences of both candidates
      left_count  =  sum(1  for  i  in  range(left,  right  +  1)  if  A[i]  ==
left_majority)
      right_count  =  sum(1  for  i  in  range(left,  right  +  1)  if  A[i]  ==
right_majority)

    # Check if either candidate appears more than n/2 times
    majority_threshold = (right - left) // 2
    if left_count > majority_threshold:
        return left_majority
    if right_count > majority_threshold:
        return right_majority

    return None  # No majority element
```

3.1. Missing Item:

```python
# given 0-n-1 numbers where one of the numbers is missing in an unsorted list, we find it using
def missing(a, pPivot=0):
    if len(a) == 0:
        return pPivot  # Base case: the missing number is found

    pivot = a[0]  # Choose pivot (this choice is arbitrary)

    left = [i for i in a if i < pivot]
    right = [i for i in a if i > pivot]

    left_size = len(left)  # Number of elements in left partition
     expected_left_size = pivot - pPivot  # Expected count of elements in left

    if left_size == expected_left_size:
        # If left partition has the expected count, search in right
        return missing(right, pivot + 1)
    else:
        # Otherwise, search in left
        return missing(left, pPivot)
```

3.2. Count Zeros in bit string:

```python
# Counts number of zeros in a 1^m 0^k string where m+k = n
def bitString(a):
    n = len(a)
    lo = 0
    hi = n - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        # bsearch until you are a 1 and the next element is a 0 then the answer is just the difference
        if a[mid] == "1" and a[mid + 1] == "0":
            return n - mid - 1
        if a[mid] == "0":
            hi = mid - 1
        else:
            lo = mid + 1
    return None
```

4. Dump

- If you some how know that the number of inversions is very low in this unordered list, then using insertion sort is good because it is almost linear and has to loop back very little as most items are already in a relatively close place to their final destinations in the sorted array

Generalizing the Pattern

After k steps, we get:

$$T(n) = T(n^{1/2^k}) + k$$

We stop when $n^{1/2^k}$ reaches a base case (e.g., when it becomes a constant, say $T(1)$).
This happens when:

$$n^{1/2^k} = 1$$

Taking the log on both sides:

$$\frac{1}{2^k} \log n = 0$$

$$\log n = 2^k$$

$$k = \log \log n$$

Final Complexity

Since the number of steps is $O(\log \log n)$, we conclude:

$$T(n) = \Theta(\log \log n)$$

Rule 1: $\log_b(M \cdot N) = \log_b M + \log_b N$

Rule 2: $\log_b\left(\frac{M}{N}\right) = \log_b M - \log_b N$

Rule 3: $\log_b\left(M^k\right) = k \cdot \log_b M$

Rule 4: $\log_b(1) = 0$

Rule 5: $\log_b(b) = 1$

Rule 6: $\log_b\left(b^k\right) = k$

Rule 7: $b^{\log_b(k)} = k$

Where:

$b > 0$ but $b \neq 1$, and M, N, and k are real numbers but M and N must be positive!

Washington and Lee University
Email address: jbajracharya@mail.wlu.edu