CSC2710 Analysis of Algorithms — Fall 2022
Unit 6 - Divide-and-Conquer

Book: §2.3.1; §4.1 – §4.2; §7.1; §12.1 – §12.3;

6.1 Divide-and-Conquer

- Divide-and-Conquer is perhaps the best known algorithm design technique.
- The general Divide-and-Conquer strategy works as follows:
  1. **Divide** the problem into several subproblems of the same type.
     - Ideally, each subproblem should be of about equal size.
  2. The subproblems are solved (i.e. **conquered**).
     - The subproblems are normally solved recursively.
     - Occasionally not recursive solutions are used. Especially, if the size of the subproblem is small.
  3. Solutions to the subproblems are combined to get a solution to the original problem.
     - This is only done if necessary.
- Traditionally, divide-and-conquer algorithms divide a given problem into two equally sized subproblems.
- Consider the problem of computing the sum of n numbers $a_1, a_2, \ldots, a_n$. How would we solve it using a divide-and-conquer strategy?
  - If $n > 1$ divide the problem into to two pieces
    1. Recursively compute the sum of the first $\lfloor \frac{n}{2} \rfloor$.
    2. Recursively compute the sum of the remaining $\lceil \frac{n}{2} \rceil$ numbers.
  - Once the two sub problem have been solved, we sum the two solutions to get the final solution.
- Do you think this is an efficient algorithm compute the sum of a sequence of numbers?
- Let's use a recurrence to determine the actually cost of the algorithm based on the number of additions:

$$A(n) = 2A\left(\frac{n}{2}\right) + 1$$

  - **Practice**: Solve this recurrence using the Master Method.

6.2 Mergesort

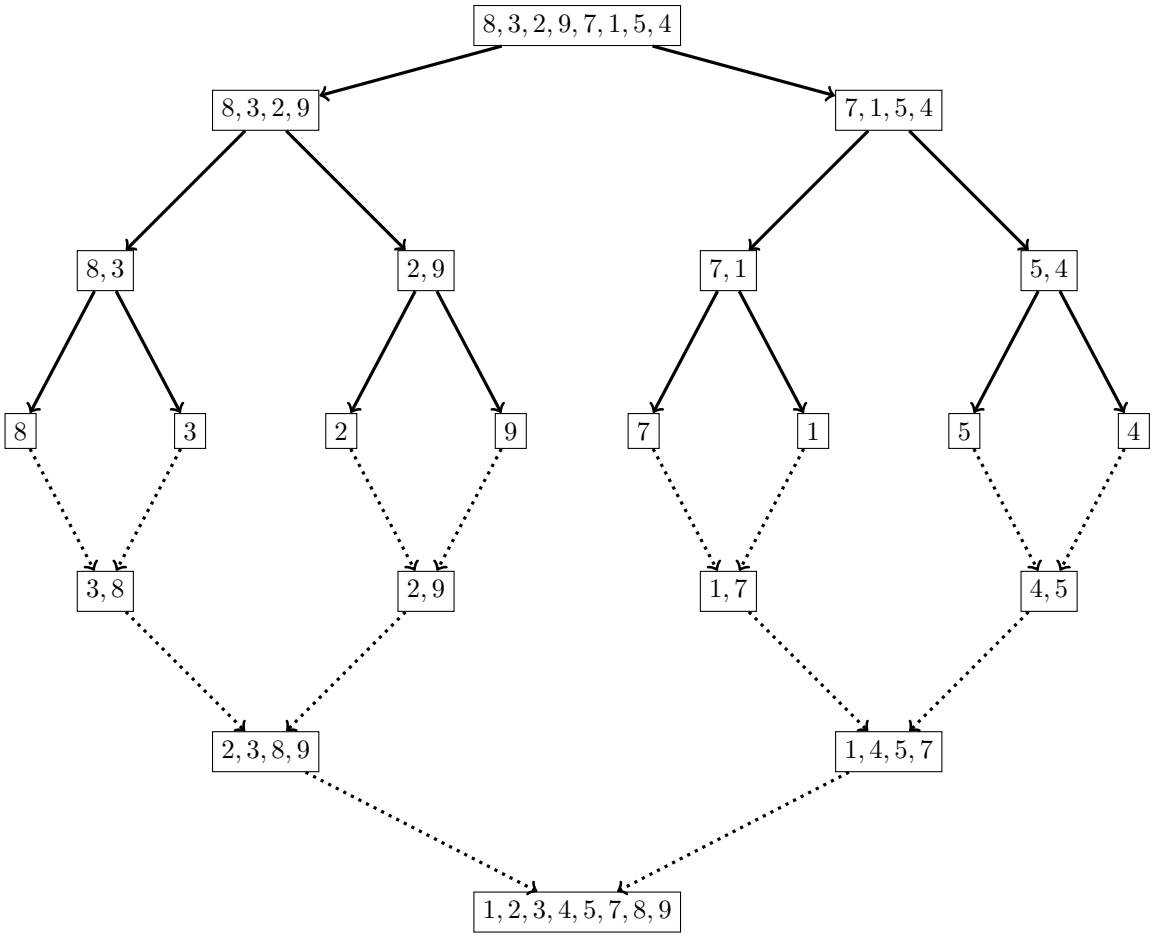
- **Practice**: what is the sorting problem again?
- Intuition: Split the array $A[0..n-1]$ to be sorted into two pieces $A\left[0..\left\lfloor\frac{n}{2}\right\rfloor-1\right]$ and $A\left[\left\lfloor\frac{n}{2}\right\rfloor..n-1\right]$; and recursively sorting them and then merging the two smaller sorted arrays together.
- Formally the MERGESORT algorithm works as follows:

```
# Input: An array A of integers
# Output: A is in sorted order
def MergeSort(A):
    if( len(A) > 1 ):                                    # Line 1
        # Divide A in half
        B = A[0 : (math.floor(len(A)/2))].copy()         # Line 2
        C = A[math.floor(len(A)/2) : len(A)].copy()      # Line 3
        MergeSort( B )                                   # Line 4
        MergeSort( C )                                   # Line 5
        Merge( A, B, C )                                 # Line 6
```

- Intuitively, the MERGE operation works as follows:
  - Two pointers are initialized to point to the first elements of the arrays being merged.
  - The elements being pointed to are compared, the smaller being added to the result array.
  - The index of the smaller element is incremented to point to its successor.
  - Repeat the operation until one array is exhausted.
  - The remaining elements of the other array are copied to the result array.
- The combine phase in MERGESORT is called MERGE. The MERGE algorithm is as follows:

```
# Input: Sorted arrays B[0 .. p-1] and C[0 .. q-1],
#    and a destination array A of size p + q
# Output: Sorted array A[0 .. p+q-1] of the elements of B and C
def Merge( A, B, C ):
    i = 0                                                # Line 1
    j = 0                                                # Line 2
    k = 0                                                # Line 3
    p = len(B)                                           # Line 4
    q = len(C)                                           # Line 5
    # Interleave the two subarrays
    while( i < p and j < q ):                            # Line 6
        if( B[i] <= C[j] ):                              # Line 7
            A[k] = B[i]                                  # Line 8
            i = i + 1                                    # Line 9
        else:                                            # Line 10
            A[k] = C[j]                                  # Line 11
            j = j + 1                                    # Line 12
        k = k + 1                                        # Line 13
    # Copy the remaining elements into the final array
    while( i < p ):                                      # Line 14
        A[k] = B[i]                                      # Line 15
        k = k+1                                          # Line 16
        i = i+1                                          # Line 17
    while( j < q ):                                      # Line 18
        A[k] = C[j]                                      # Line 19
        k = k+1                                          # Line 20
        j = j+1                                          # Line 21
```

- Let's look at an example of mergesort for the sequence $A = \langle 8, 3, 2, 9, 7, 1, 5, 4\rangle$. This is normally represented as a graph that demonstrates both the divide steps (solid arrows) *and* the conquer steps (dotted arrows).



- In the worst case what is the the cost of MERGE
  - Answer $n-1$ where $n = p + 1 - 1$
  - Does everyone see why?
- What is the recurrence relation that describes MERGESORT?
  - $T(n) \approx 2T\left(\frac{n}{2}\right) + n - 1$
- What is the closed form of this recurrence?
  - By the Master Theorem we have:
    $$T(n) \in \Theta(n \lg n)$$
  - This makes MERGESORT's running time optimal[1].
    - Note: $a = 2$, $b = 2$, and $d = 1$ so, $a = b^d$.
- Observe that MERGESORT requires extra memory, beyond what is required to hold the input, to solve the sorting problem.

[1] It turns out that if you limit yourself to comparison operations, the sorting problem (not a particular algorithm) has running time $\Omega(n \lg n)$.

6.3 Quicksort

- The Quicksort algorithm sorts an array of numbers using a divide and conquer strategy.
- The divide and conquer strategy for quicksort works by using a partitioning algorithm to split the array in two pieces and then recursively calling the partition algorithm on those two pieces.
  - Observation 1: the pivot in a partition algorithm is *always* placed in its correct final location.
  - Observation 2: All the work in the algorithm occurs in the divide phase there is no real work conquer phase.
- **Practice**: recall Lomuto's Partitioning algorithm. How does it work?
- We can formally state the Quicksort algorithm as below:

```
# Input: An array A, defined by left and right indices
# Output: A is in sorted increasing order
def QuickSort(A, left, right):
    # If our pointers haven't met yet
    if( left < right ):                                  # Line 1
        # Partition the array
        s = LumotoPartition(A, left, right)              # Line 2
        # Sort the lower half, then the upper half
        QuickSort( A, left, s - 1 )                      # Line 3
        QuickSort( A, s + 1, right )                     # Line 4
```

  - The PARTITION algorithm can be any valid partitioning algorithm. The book mentions
    - Lomuto's Partitioning
    - Hoare's Partitioning[2]
- For ease lets just look at Lomuto's Partition as the algorithm providing partition support for QUICKSORT.
- what is the worst case input for QUICKSORT?
  - What ever is the worst case input is for PARTITION.
- What is the worst case input for LOMUTOPARTITION?
  - The array is sorted in increasing order.
  - This implies that every partition will have all of the elements to one side of the pivot.
  - This means that every call to QUICKSORT will work on an array of size one less instead of half.
  - The running time, therefore, is given by $\sum_{i=0}^{n-1} i = \frac{n(n+1)}{2} \in \Theta\left(n^2\right)$
  - This makes QUICKSORT *a lot* worse than MERGESORT.
- If QUICKSORT has such bad worst case running time, why are we even bothering?
  - QUICKSORT has good average case running time.
  - Let's look at the number of comparisons made by QUICKSORT on a randomly ordered array of size n.
    - A pivot can end up in any position s after $n+1$ comparisons made by PARTITION.
    - This implies the left partition will have s elements and the right partition will have $n-s-1$ elements

[2]Due to C. A. R. Hoare, the inventor of Quicksort and many other influential algorithms.

* We can safely assume that the partition can end up in any position with equal probability.
  - **Practice**: What is that probability?
* We can write the recurrence relation as:

$$T(n) = \frac{1}{n}\sum_{s=0}^{n-1}\left((n+1) + T(s) + T(n-1-s)\right)$$

  provided $n > 1$
* The boundary conditions are: $T(0) = 0$ and $T(1) = 0$.
* This recurrence requires *significant* math to complete[3]. Therefore, I will just tell you that it works out to $T(n) \in \Theta(n \lg n)$.

- QUICKSORT is an *extremely* well studied algorithm researchers have come up with all sorts of ways to improve it.
  - Normally, by improving the PARTITION algorithm in some way.

[3]Basically, we argue the bound using expectation

6.4 Binary Tree Traversals and Related Properties

- Divide and Conquer techniques naturally apply to binary search trees.
- Formally, we define a binary search tree T as a set of nodes that is either:
  - empty or,
  - consists of a *root* and two disjoint binary search trees T_R and T_L, the right and left subtrees respectively.
- This very structure lends it self nicely to the idea of divide-and-conquer algorithms.
- Example: How do we compute the height of a binary search tree (using divide and conquer)?
  - We divide the tree into subtrees and continuously look for the maximum height subtree adding that to the count.
  - The algorithm is:

```
# Input: T is a binary tree
# Output: The height of T
def TreeHeight( T ):
    if( T == None ):                                                    # Line 1
        return -1                                                       # Line 2
    else:                                                               # Line 3
        return max(TreeHeight(T.right), TreeHeight(T.left)) + 1         # Line 4
```

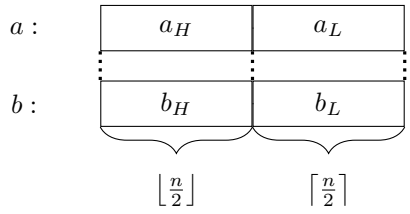
  - What is the running time of this algorithm?
    - The time is really dictated by the number of calls made to TreeHeight.
    - During each call the algorithm checks to see if T is empty.
    - To ease the analysis we will force every node of the tree to be an internal node. Specifically every node will have two children. We will call this tree an *extension tree*
      - A node is an internal node if it has at least one child.
    - This means the one call is made for every internal node *until* a leaf is reached.
    - How many leaves are there in the extension tree?
      - Every node except the root is one of two children of an internal node we have $2n + 1$ where n is the number of internal nodes.
      - This implies that the number of leaves is $n + 1$.
    - Therefore, the running time of the algorithm is $\Theta(n)$.
- The three well known traversals are also simple divide and conquer aglorithms on trees. They are:
  1. Preorder: root, left, right
  2. Postorder: left, right, root
  3. In-order: left, root, right
- Everyone remembers there traversals from CSC2820 and MTH1314?
- **Practice**: Which traversal prints the nodes of the tree out in sorted order?

6.5 Multiplication of Large Integers

- Divide and Conquer algorithms come up in performing mathematics on computers as well.
  - Example: In cryptography we often need to multiply large numbers. In other words, numbers that don't fit a standard numeric data type.
- **Practice**: If you use the traditional multiplication algorithm how long will it take to multiply two n digit numbers?
- The surprising algorithm that we will explore is called Karatsuba's Multiplication algorithm.
- Karatsuba's algorithm is based on the observation that the product of any two, two-digit numbers a_1a_0 and b_1b_0 is

  $$c_2 10^2 + c_1 10^1 + c_0,$$

  where $c_0 = a_0 b_0$, $c_2 = a_1 b_1$, and $c_1 = (a_1 + a_0)(b_1 + b_0) - (c_2 + c_0)$.
- We can think about extending this algorithm to n digit numbers by dividing the number of digits of each number in half each time we recur.
  - In other words think of $a = a_H a_L$ and $b = b_H b_L$ such that the size of $a_L = a_H = b_L = b_H \approx \lfloor \frac{n}{2} \rfloor$.
  - Graphically we have:

  - Multiplication becomes,

    $$\begin{aligned} ab &= \left(a_H 10^{\frac{n}{2}} + a_L\right)\left(b_H 10^{\frac{n}{2}} + b_L\right) \\ &= (a_H b_H)\, 10^n + (a_H b_L + a_L b_H)\, 10^{\frac{n}{2}} + (a_L b_L) \\ &= c_2 10^n + c_1 10^{\frac{n}{2}} + c_0 \end{aligned}$$

    where $c_2 = a_H b_H$, $c_0 = a_L b_L$, and $c_1 = (a_H + a_L)(b_H + b_L) - (c_2 + c_0)$.
- The goal her was to reduce the number of multiplications as that affects the running time of this algorithm.
- The running time in this case for an instance of size n is:

  $$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n)$$

  Since, we have three instances of a subproblem of size $\frac{n}{2}$ and $\Theta(n)$ time for the constant number of n-bit additions.
- What is the running time (closed form)?
  - By the Master Theorem we have $a = 3$, $b = 2$ and $d = 1$ which means $a > b^d$ so our running time is $\Theta\left(n^{\lg 3}\right)$.

Challenge Problems

- Develop a divide-and-conquer algorithm (in Python) to find the **mean** of a list of numbers `A`.
  - Is there a decrease-and-conquer strategy to finding the mean of a set of numbers? Why or why not?
  - What is the running time of your algorithm?
- Develop a divide-and-conquer algorithm to find the **median** of a list of numbers `A`. This is *non-trivial* but give it a shot. You don't need to write it in Python.
  - Is there a decrease-and-conquer strategy to finding the median of a set of numbers? Why or why not?
  - What is the running time of your algorithm?
- Is there a divide-and-conquer strategy to find the **mode** of a list of numbers `A`? If so, find it. If not, why not?
- Develop a divide-and-conquer algorithm to multiply two matrices together. What is the running time of your algorithm?