CSC 6013 Algorithms and Discrete Structures   Michael Bradley

Class Notes for Week 8:

Divide-and-Conquer Algorithms

Read these sections from the textbook: §12.1 - 12.3

Divide-and-Conquer

The divide-and-conquer algorithm design technique is one of the most common design techniques. It is similar in form to the decrease-and-conquer but, instead of decreasing the size of the problem by a factor, constant, or variable amount, the divide-and-conquer strategy divides the problem into multiple subproblems that are then conquered.

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.

A. Array Sum

Previously, you wrote code to for a recursive function that calculates the sum of the entries in an array of n numbers [a1, a2, … , an] as follows:

If n > 1 divide the problem into two pieces

1. Recursively compute the sum of the first $\left\lfloor \frac{n}{2} \right\rfloor$ numbers.
2. Recursively compute the sum of the remaining $\left\lceil \frac{n}{2} \right\rceil$ numbers.
3. Once the two sub-problem have been solved, add the two solutions to get the final solution.

This is a divide-and-conquer strategy.

Think of the problem this way. Your boss asks you to add up a list of numbers. You give the first half of the numbers to one of your assistants and tell him to add them up for you. You give the second half of the numbers to your other assistant and tell her to add them up for you. When your assistants give you the answers to their sub-problems, you add those two numbers together and give the answer to your boss (with a big smile). Your assistants can use the same strategy if they each have two assistants who work for them.

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:

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

Solve this recurrence using the Master Method to show that this is O(n).

For example, to sum the elements of the array A = [80, 30, 10, 60, 70, 40, 50], we would essentially

return ArraySum([80, 30, 10]) + ArraySum([60, 70, 40, 50]).

These recursive calls would generate the calls

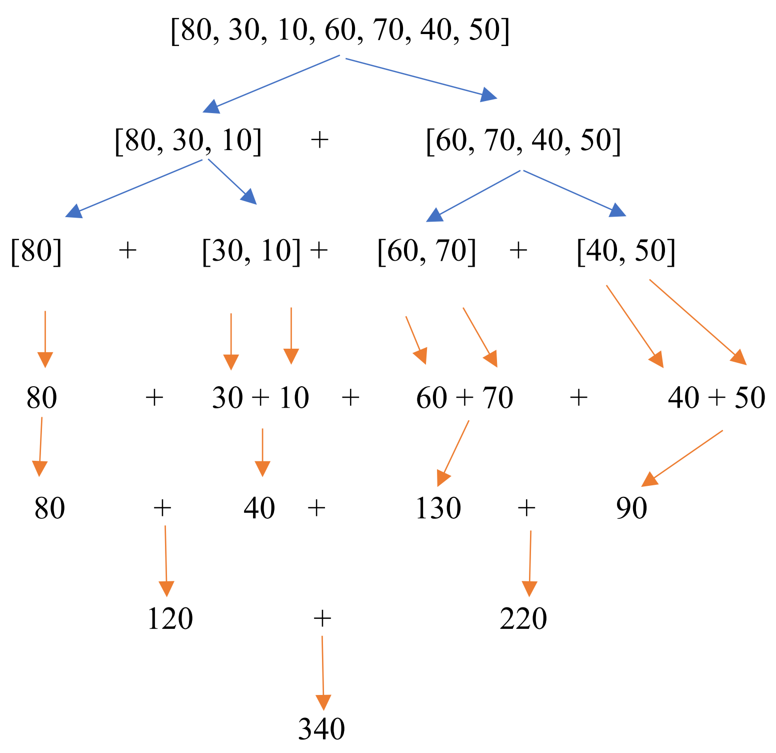
return ArraySum([80]) + ArraySum([30, 10])

and

return ArraySum([60, 70]) + ArraySum([40, 50]).

The first of these four calls returns the value 80, while the other three calls collectively generate six calls with arrays of size 1 that return individual values.

Without all the words, we can use a graph that demonstrates both the divide steps (blue arrows) and the conquer steps (red arrows).



B. Mergesort

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 $A\left[\left\lfloor\frac{n}{2}\right\rfloor..\,n-1\right]$

recursively sort them, and then merge the two smaller sorted arrays together.

Formally the Mergesort algorithm works as follows:

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

Note 1: You may have to import math.

Note 2: To follow the logic of the MergeSort algorithm, it might be helpful to insert commands

print(“B= “, B)

and

print(“C= “, C)

between lines 7 and 8 to see what portions of the original array are being sent to the recursive calls in lines 8 and 9.
With the 8-element array A = <8, 3, 2, 9, 7, 1, 5, 4> that is used to trace the code two pages later in these notes, the first recursive calls are with the first four elements B = <8, 3, 2, 9> and the last four elements C = <7, 1, 5, 4>.

Intuitively, the Merge operation works as follows:
Two pointers are initialized to point to the first elements of the arrays being merged (B and C).
The elements being pointed to are compared, the smaller being added to the result array (A).
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:

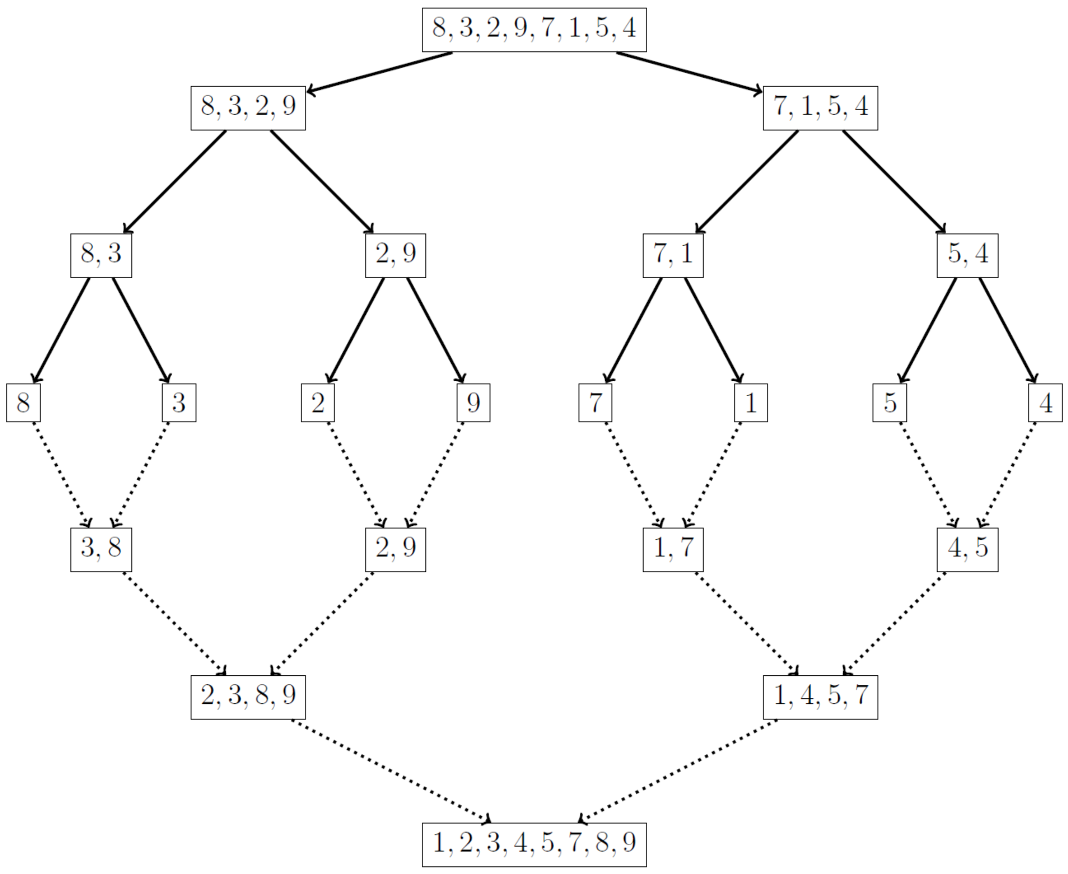
```
# Input: Sorted arrays B[0..p-1] and C[0..q-1].
# Output: Sorted array A[0..p+q-1] of elements of B and C.
def Merge(A, B, C):
    i = 0
    j = 0
    k = 0
    p = len(B)
    q = len(C)

    #Interleave the two subarrays
    while i<p and j<q:
        if B[i] <= C[j]:
            A[k] = B[i]
            i = i+1
        else:
            A[k] = C[j]
            j = j+1
        k = k+1

    # Copy the remaining elements into the final array
    while i<p:
        A[k] = B[i]
        i = i+1
        k = k+1
    while j<q:
        A[k] = C[j]
        j = j+1
        k = k+1
```

Notice that the three while loops are all indented the same, so that the first loop must finish its execution before the second one starts, and the second loop must finish its execution before the third one starts.

Example B1: Let's look at an example of Mergesort for the sequence A = <8, 3, 2, 9, 7, 1, 5, 4>. 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 cost of Merge?
Answer: n - 1 where n = p + q - 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) = O(n lg n)
This makes Mergesort's running time "optimal", meaning that no sorting algorithm does less work than n lg n.
Notice however, that Mergesort uses a lot of extra space to make working copies of the sub-arrays. How much space is required in the worst case? What is the worst case?

C. Quicksort

The Quicksort algorithm sorts an array of numbers using a divide and conquer strategy by using a partitioning algorithm to split the array in two pieces and then recursively calling the partition algorithm on those two pieces.

Version A: using Lomuto partition

```
# Input: Array A defined by left and right indices.
# Output: A[left .. right] sorted in increasing order.
def QuickSort(A, left, right):
 if left < right:
   s = LomutoPartition(A, left, right)
   QuickSort(A, left, s)
   QuickSort(A, s + 1, right)
```

Example C1: Walk through Quicksort Version A with the example:

A = [85, 21, 63, 24, 88, 45, 17, 31, 96, 50]

Pass #1: 85 is the pivot. Using Lomuto partitioning, compare each element with the pivot, and swap two elements when one is less than the pivot and the other is greater.

[50, 21, 63, 24, 45, 17, 31, 85, 96, 88]

Pass #2: Now call quicksort for the subarrays

[50, 21, 63, 24, 45, 17, 31] and [96, 88].

50 is the pivot in the left "half". 96 is the pivot in the right "half". These become

[31, 21, 24, 45, 17, 50, 63] and [88, 96].

Pass #3: Now call quicksort for the subarrays

[31, 21, 24, 45, 17] [63] and [88].

Taking these one at a time:
In

[31, 21, 24, 45, 17]

with 31 as the pivot, this partitions as

[17, 21, 24, 31, 45]

For

[63] and [88]

there is no recursive call because left == right

[63] and [88] are sorted.

Pass #4: For

[17, 21, 24]

17 is the pivot and the partition produces

[17, 21, 24]

For [45]

there is no recursive call.

[45] is sorted.

Pass #5: For

[21, 24]

21 is the pivot and the partition produces

[21, 24]

Pass #6: For B = [24]

there is no recursive call.

[24] is sorted.

Consider the following visualization for this sequence of steps in a different presentation with notes. Each array entry is colored red when it is placed in its final sorted slot after being used as a pivot or is colored blue when it is placed in its final sorted slot after becoming the only element in a subarray of size 1:

```
[85, 21, 63, 24, 88, 45, 17, 31, 96, 50]
                              85 is the pivot (9 comparisons)
[50, 21, 63, 24, 45, 17, 31]  85  [96, 88]
                              50 and 96 become pivots (6 + 1 = 7 comparisons)
[31, 21, 24, 45, 17]  50  [63]  85  [88] 96
                              31 becomes a pivot; 63 and 88 are done (4 comparisons)
[17, 21, 24]  31  [45]  50  63  85  88 96
                              17 becomes a pivot; 45 is done (2 comparisons)
17  [21, 24]  31  45  50  63  85  88  96
                              21 becomes a pivot; (1 comparison)
17  21  [24]  31  45  50  63  85  88  96
                              24 is done
17  21  24  31  45  50  63  85  88  96
```

There were 6 recursive calls using the 6 red elements as the pivots.

ARRAY IS SORTED with a total of 23 comparisons.

Version B: using a different partitioning algorithm and twice as much memory.
Note that line 7 selects the **last** element in each section of the array as the pivot, rather than the **first** element as in Lomuto partitioning.

```
1    # Input: Array A defined by left and right indices.
2    # Output: A[left..right] sorted in increasing order.
3    def Quicksort (A, left, right):
4    # If left > right, there are no elements to be sorted.
5    # If left == right, there is only one array element, so it is sorted.
6        if left < right:
7            pivot = right
8    # partition elements of A by copying into B
9            L = left
10           R = right -1
11           for j in range(left, right)
12   # if A[j] smaller than pivot, copy A[j] into left half of B
13               if A[j] < A[pivot]:
14                   B[L] = A[j]
15                   L = L + 1
16   # if A[j] larger than pivot, copy A[j] into right half of B
17               else:
18                   B[R] = A[j]
19                   R = R - 1
20   # copy pivot in between both "halves" of B
21           B[L] = A[pivot]
22           pivot = L
23   # now recursively quicksort both portions of B if there is more than one element
24           if left < pivot-1:
25               Quicksort(B, left, pivot-1)
26           if pivot+1 < right:
27               Quicksort(B, pivot+1, right)
```

Example C2: Walk through Quicksort using this alternative pivoting method with the example:

A = [85, 21, 63, 24, 88, 45, 17, 31, 96, 50]

Pass #1: 50 is the pivot. compare each element with the pivot and copy it into the left half or the right half of the second array

B = [21, 24, 45, 17, 31, 50, 96, 88, 63, 85]

Pass #2: Now call quicksort for the subarrays

A = [21, 24, 45, 17, 31] and A = [96, 88, 63, 85].

31 is the pivot in the left half. 85 is the pivot in the right half. In each subarray, compare each element with the pivot and copy it into the left half or the right half of the sub-array. These become

B = [21, 24, 17, 31, 45] and B = [63, 85, 88, 96].

Pass #3: Now call quicksort for the subarrays

A = [21, 24, 17] A = [45] A = [63] A = [88, 96].

Taking these one at a time:
In

A = [21, 24, 17],

17 is the pivot. After two comparisons and copies we get

B = [17, 24, 21]

For

A = [45]

there is no recursive call because left == right

A = [45] is sorted.

Similarly, for

A = [63]

there is no recursive call because left == right

A = [63] is sorted.

For

A = [88, 96]

96 is the pivot. After one comparison and copy we get

B = [88, 96].

Pass #4: For

[24, 21]

21 is the pivot. This becomes

[21, 24]

For

    [88]

there is no recursive call.

    [88] is sorted.

Pass #5: For

    [24]

there is no recursive call

    [24] is sorted.

Consider the following visualization for this sequence of steps in a different presentation with notes. Each array entry is colored red when it is placed in its final sorted slot after being used as a pivot or is colored blue when it is placed in its final sorted slot after becoming the only element in a subarray of size 1:

```
[85, 21, 63, 24, 88, 45, 17, 31, 96, 50]
                              50 becomes the pivot (9 comparisons)
[21, 24, 45, 17, 31]  50  [96, 88, 63, 85]
                              31 and 85 become pivots (4 + 3 = 7 comparisons)
[21, 24, 17]  31  [45]  50  [63]  85  [88, 96]
                              17 and 96 become pivots; 45 and 63 are done (2 + 1 = 3 comparisons)
17  [24, 21]  31  45  50  63  85  [88]  96
                              21 becomes a pivot; 88 is done (1 comparison)
17  21  [24]  31  45  50   63  85  88  96
                              24 is done
17  21  24  31  45  50  63  85  88  96
```

There were 6 recursive calls using the 6 red elements as the pivots.

ARRAY IS SORTED with a total of 20 comparisons.

Asymptotic analysis of Quicksort

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" (partition) phase; there is no real work in the "conquer" phase.

Q: What is the worst-case input for Quicksort?

A: The fundamental unit of work is the comparison of array elements is the if statement in line 13. The maximum amount of work occurs when the array is already sorted in increasing order. This implies that every partition will put all of the elements on 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.

Q: If Quicksort has such bad worst-case running time, why are we even bothering?

A: Quicksort has good average case running time. In the best case, every pivot splits the remaining array elements into two equal subarrays, which are then recursively sorted. This involves lg(n) "halving" steps (divide array into 2 halves, then into 4 quarters, then into 8 eights, etc) and each set of "array halves" require < n keys to be compared to the new pivots (compare n keys to the one pivot, compare n-3 keys to the 2 pivots, compare n-7 keys to the 4 picots, etc) so the work in the best case is B(n) = O(n lg(n)). The average case is quite close to the best case, so typically quicksort runs in O(n lg(n)) time.

Q: What additional space is needed?

A: Lomuto partitioning needs only one memory location to implement the swaps. The alternative partitioning algorithm (method B) requires enough space in the computer's memory to make an extra copy of the array in each partitioning step, so O(n) space is needed.

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 TR and TL, the right and left subtrees respectively.

This very structure lends itself nicely to the idea of divide-and-conquer algorithms.

D. Height of Binary Tree

Q: How do we compute the height of a binary search tree (using divide and conquer)?

A: We divide the tree into subtrees and continuously look for the maximum height subtree, adding that to the count. Basically, the height of a non-empty tree is one more than the height of its tallest subtree.

The algorithm is:

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

Notice that an empty tree has height -1. A tree that consists of just a single node, the root of the tree, has height 0. This is consistent with the definition of the height of a tree as the length of the longest path from the root to a leaf node.

Q: What is the running time of this algorithm?

A: 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 by adding a "dummy" node every time a "real" node has a nil pointer. Then every node will have two children (either real or dummy). 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.

Q: How many leaves are there in the extension tree?

A: Every node except the root is one of two children of an internal node, so we have 2n + 1 total nodes, where n is the number of internal nodes. Since there are n nodes in the tree, this implies that the number of leaves is n + 1. Therefore, the running time of the algorithm is O(n).

Looking at this another way, each node of the original tree will have two children – actual or dummy – in the extension tree, and each dummy node that was introduced will be a leaf node with no children of its own. In this extension tree, if the current node is an internal node, the "return max" command will make a recursive call to TreeHeight once for each child; if the current node is a dummy/leaf node, the call will return the value -1 without making any new recursive calls. So, TreeHeight will be called once for every node in the extension tree, for a total of 2n+1 calls, so the worst case will be in class O(n).

E. Tree Traversals – Preorder, Postorder, Inorder, Levelorder

A tree traversal is an algorithm that provides an orderly way to visit/process each node in the tree exactly once. “Processing” a node could mean any of a variety of tasks. If the employee records of a large company are stored as a binary tree, processing each node could mean printing out every employee’s name, giving every employee a 5% raise, counting how many employees work in a particular department, calculating this week’s payroll, etc. In all these situations, it is important not to miss anyone and not to double count anyone.

There are three well known traversals that are simple divide and conquer algorithms. Given a binary tree of height n, its left subtree and its right subtrees each have heights at most n-1. At each stage, these algorithms do one unit of work and break the problem down into two smaller problems. This is a classic divide-and-conquer strategy.

E1. Preorder traversal of a tree

```
process the root,
traverse the left subtree,
traverse the right subtree
```

The basic principle is to process each node before traversing its two subtrees
In short, a preorder traversal goes through the tree as “root, left, right”

E2. Postorder traversal of a tree

```
traverse the left subtree,
traverse the right subtree
process the root,
```

The basic principle is to process each node after traversing its two subtrees
In short, a postorder traversal goes through the tree as “left, right, root”

E3. Inorder traversal of a tree

```
traverse the left subtree,
process the root,
traverse the right subtree
```

The basic principle is to process each node in between traversing its two subtrees
In short, an inorder traversal goes through the tree as “left, root, right”

Q: Which traversal prints the nodes of a binary search tree out in sorted order?

Q: What is the worst-case performance of each algorithm – preorder, postorder, inorder – using recursive call as the fundamental unit of work?

E4. Levelorder traversal of a tree

Although not a divide-and-conquer algorithm, it is interesting to discuss this algorithm at this time. It accomplishes a result that is equivalent to the three prior algorithms (process every node in the tree once)but it uses a queue to implement a brute-force approach.

```
Given a tree and an empty queue,
    enqueue the root of the tree.
    While the queue is not empty,
        dequeue the node at the head of the queue and process this node,
        enqueue its left child,
        enqueue its right child.
```

Note that this processes the root (level 0), then all of its children (level 1), then their children (level 2), etc. The nodes in the tree are processed by level from top down, and from left to right within each level.

Note 1: This process also works for multi-trees where nodes can have more than two children.

Note 2: Recursive algorithms are implemented on a machine using a stack to stack up the recursive calls. The levelorder traversal algorithms uses a queue instead of a stack to process the nodes in the tree.