SIENAcollege
Computer Science

Computer Science 385
Design and Analysis of Algorithms
Siena College
Spring 2018

Lab 7: Decrease and Conquer and Divide and Conquer

Due: Start of your next lab session

You will be assigned a partner to work with on this lab. Only one submission per group is needed.

Group members: _____

Learning goals:

1. to understand the source-removal algorithm for topological sort
2. to be able to apply the concept of binary search to new problems
3. to better understand Mergesort and Quicksort and their running times

---

Submitting

Once all written items are initialed to indicate completion, turn in one copy of this handout. Be sure names of all group members are clearly on the first page.

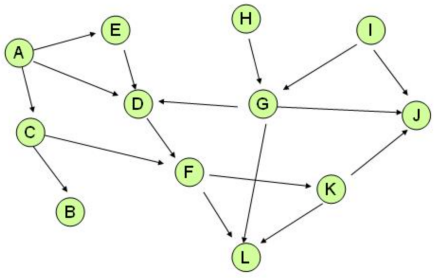
Grading

Score: ____ / 100

Question 1:

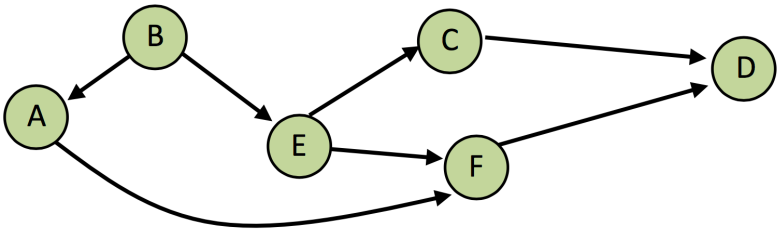
Recall that a topological ordering is a list of a directed graph's vertices in which for every edge in the graph, the vertex where the edge starts comes before the vertex where the edge ends. Below is pseudocode for the Source-Removal Algorithm which computes a topological ordering of the vertices in a directed acyclic graph (DAG). Apply the algorithm to the DAG below. Give the vertices in the order in which they are added to the list o by the algorithm. (5 points)

ALGORITHM SOURCEREMOVAL(G)
//Input: a dag $G = \{V, E\}$
$o \leftarrow$ a new empty list
while $\exists$ a source vertex in G **do**
$v \leftarrow$ a source vertex from G
$o.append(v)$
$G.remove(v)$ // also removes all incident edges
if G is empty **then**
return o
else
flag error // G was not a dag



Question 2:

The Source-Removal algorithm gives one topological ordering of the vertices, but there may be others. For the graph below, list all of the valid topological orderings of its vertices. (5 points)



For the questions on this page, we consider the familiar problem of finding the maximum value in an array A of n integers.

Question 3:
If A is **unordered**, how quickly can you find the maximum? Describe your algorithm in one sentence and give its asymptotic running time. (3 points)

Question 4:
If A is **sorted in ascending order**, how quickly can you find the maximum? Describe your algorithm in one sentence and give its asymptotic running time. (3 points)

Question 5:
Now for a challenge. Suppose A contains unique sorted values, but the values have been shifted some number of positions to the right, with wrap around. For example, `[19, 30, 35, 2, 3, 7, 9, 10, 15]` is a sorted array that has been shifted 3 positions. As another example, `[18, 20, 21, 24, 25, 27, 29, 35, 42, 5, 15]` is a sorted array that has been shifted 9 positions. Given an array shifted by some unknown amount, the problem is to develop an efficient algorithm to find the maximum value in it. Below, design and write pseudocode for an algorithm whose worst case running time is asymptotically faster than linear. (9 points)

ALGORITHM FINDMAX(A, l, r)
//Input: a an array $A[0..n-1]$
//Input: l, r, the left and right indices of the subarray to search
//On the first call, l would be 0 and r would be $n-1$

The pseudocode for mergesort that we studied in class is included near the end of this packet. You may remove it for easy reference as you complete the questions on this page.

Question 6:
Show the recursive call tree when applying `MergeSort` on the array `A = [90, 70, 40, 20, 30, 50, 80, 10]`. Show all calls to `MergeSort` and `Merge`, similar to how the first call is done below, where `ms` is short for `MergeSort`, `m` is short for `Merge`, and the values in arrays B and C when used as input to `Merge` still need to be filled in. (5 points)

```
ms(A=[90, 70, 40, 20, 30, 50, 80, 10])
```

```
m(B=[                    ], C=[                    ], A[...])
```

Question 7:
How many calls to `MergeSort` were made (including the first call)? (2.5 points)

Question 8:
How many calls to `Merge` were made? (2.5 points)

The pseudocode for quick sort that we would have studied in class if we did not have a snow day is included near the end of this packet. But fear not, you will study it plenty in lab today. You may remove it for easy reference as you complete the remaining questions.

Question 9:

In `QuickSort`, the `Partition` step picks a pivot and then rearranges the array elements so that those smaller than the pivot come first, followed by the pivot, followed by those larger than the pivot. Suppose the array A below is given as input to the `Partition` algorithm with $lt = 0$ and $rt = 9$. Show the contents of the array after each iteration of the **outer** `repeat` **loop** (12 points)

12	3	1	19	25	8	34	7	45	23

After the first iteration of the outer `repeat` loop: $i =$ ____ and $j =$ ____

After the second iteration of the outer `repeat` loop: $i =$ ____ and $j =$ ____

After the third iteration of the outer `repeat` loop: $i =$ ____ and $j =$ ____

After the outer `repeat` loop has completed, and the last swap has happened, just before the `return`

Question 10:
What value does `Partition` return in this specific example? (6 points)

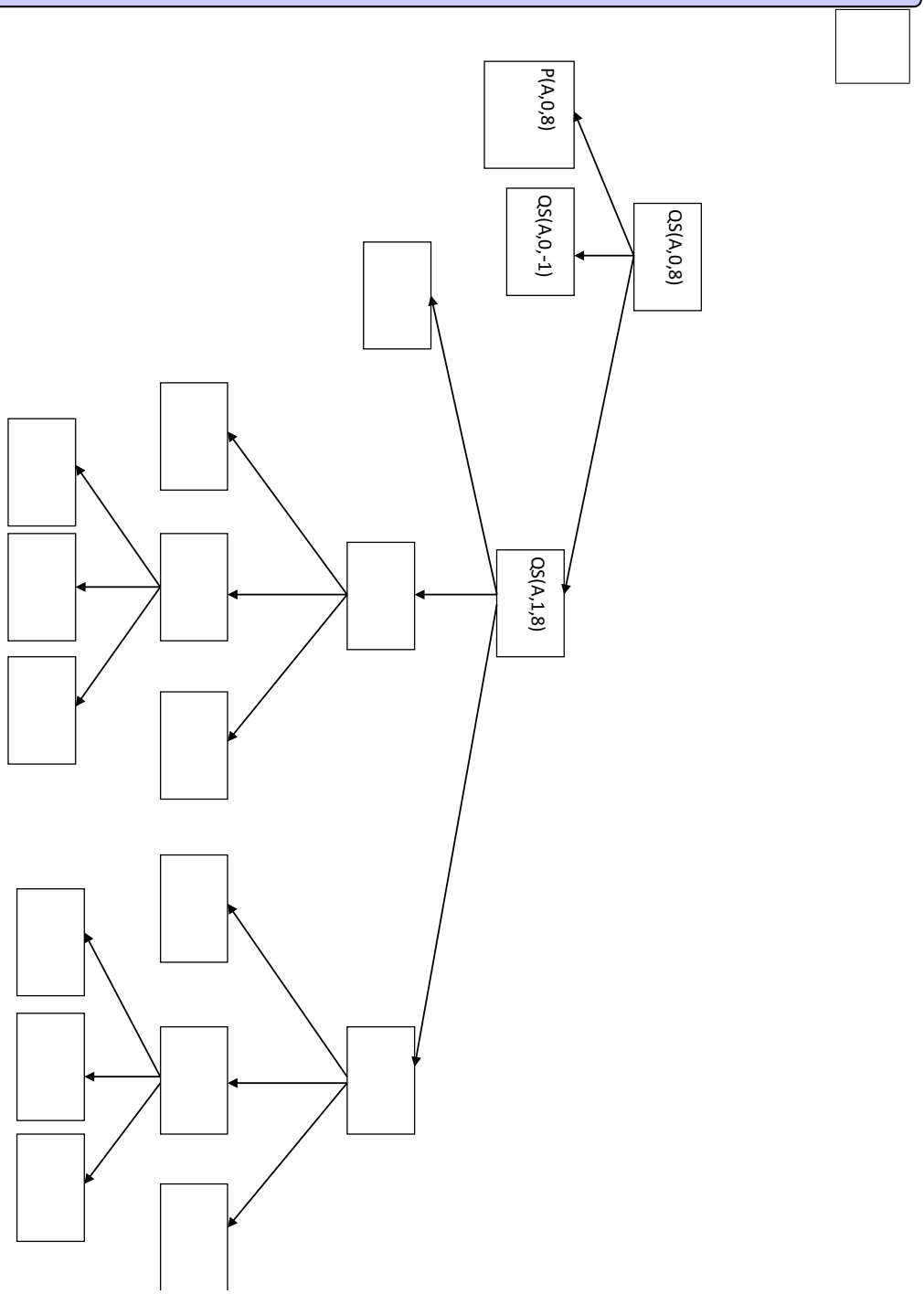
Question 11:
Describe in words, in general, what the last swap in `Partition` does. (5 points)

Question 12:
Describe in words, in general, the meaning of the value returned by `Partition`. (4 points)

Question 13:
What is the worst case efficiency class of running time of a call to `Partition`? Express your answer in terms of n where n is the size of the subarray between lt and rt. Explain your answer. (10 points)

Question 14:

Fill in the tree of subroutine calls made when `QuickSort(A,0,8)` is called with `A = [11, 55, 22, 77, 88, 44, 99, 33, 66]`. Show calls to both `Partition` (as `P`) and `QuickSort` (as `QS`). The first few are filled in for you. Show the return value of each `Partition` call. (20 points)



Question 15:

Write a recurrence formula for `QuickSort` when the pivot in every call to `Partition` is the smallest value among all values in `A` in the range lt to rt. This is the worst case for Quicksort. Find a closed form for your recurrence using any method you wish (no need to show your work here). Then give the efficiency class for the worst case running time for Quicksort. (4 points)

Question 16:

Write a recurrence formula for `QuickSort` when the pivot is the median value in every call to `Partition`. To keep things simple, you may assume that there are exactly $\frac{n}{2}$ items to the left of the median value and $\frac{n}{2}$ to the right. This is the best case for `QuickSort`. Find a closed form for your recurrence using any method you wish (no need to show your work here). Then give the efficiency class for the best case running time for Quicksort. (4 points)

This page has a copy of the merge sort algorithm from class. You may remove it for easy reference as you complete the relevant questions.

ALGORITHM MERGESORT(A)
//Input: an array $A[0..n-1]$
if $n > 1$ **then**
copy first half of array A into a temp array B
copy second half of array A into a temp array C
$MergeSort(B)$
$MergeSort(C)$
$Merge(B, C, A)$

ALGORITHM MERGE(B, C, A)
//Input: a sorted array $B[0..p-1]$
//Input: a sorted array $C[0..q-1]$
//Output: a sorted array $A[0..(p+q-1)]$
$i \leftarrow 0$
$j \leftarrow 0$
$k \leftarrow 0$
// take smallest item from B and C and put into A
while $i < p$ **and** $j < q$ **do**
if $B[i] < C[j]$ **then**
$A[k] \leftarrow B[i]$
$i \leftarrow i + 1$
else
$A[k] \leftarrow C[j]$
$j \leftarrow j + 1$
$k \leftarrow k + 1$
// if items remain in B, put them in A
while $i < p$ **do**
$A[k] \leftarrow B[i]$
$i \leftarrow i + 1$
$k \leftarrow k + 1$
// if items remain in C, put them in A
while $j < q$ **do**
$A[k] \leftarrow C[j]$
$j \leftarrow j + 1$
$k \leftarrow k + 1$

This page has a copy of the quicksort algorithm from class. You may remove it for easy reference as you complete the relevant questions.

ALGORITHM QUICKSORT(A, lt, rt)
//Input: an array $A[0..n-1]$
//Input: lower lt and upper rt bounds of the subarray to sort
//The initial call would be with $lt = 0$ and $rt = n - 1$
if $lt < rt$ **then**
$s \leftarrow Partition(A, lt, rt)$
// s is pivot element location
$QuickSort(A, lt, s-1)$
$QuickSort(A, s+1, rt)$

where the partition is accomplished by:

ALGORITHM PARTITION(A, lt, rt)
//Input: an array $A[0..n-1]$
//Input: lower rt and upper rt bounds of the subarray
// to partition
$p \leftarrow A[lt]$ // select pivot
$i \leftarrow lt$
$j \leftarrow rt + 1$
repeat
repeat
$i \leftarrow i + 1$
until $i = rt$ **or** $A[i] \geq p$
repeat
$j \leftarrow j - 1$
until $j = lt$ **or** $A[j] \leq p$
$swap(A[i], A[j])$
until $i \geq j$
$swap(A[i], A[j])$ // undo last swap
$swap(A[lt], A[j])$ // place pivot
return j