Instructions

- This problem set is **open book**: you may refer to the lectured material found on Canvas and the recommended books to help you answer the questions.
- This problem set is an **individual effort**. You must arrive at your answers independently and write them up in your own words. Your solutions should reflect your understanding of the content.
- **Posting questions to message boards or tutoring services including, but not limited to, Chegg, StackExchange, etc., is STRICTLY PROHIBITED. Doing so is a violation of the Honor Code.**
- Your solutions must be submitted typed in LaTeX, **handwritten work is not accepted**. If you want to include a diagram then we do accept photos or scans of hand-drawn diagrams included with an appropriate `\includegraphics` command. It is your responsibility to ensure that the photos you obtain are in a format that pdflatex understands, such as JPEG.
- The template tex file has carefully placed comments (% symbols) to help you find where to insert your answers. There is also a **STUDENT DATA** section in which you should input your name and ID, this will remove the warnings in the footer about commands which have not been edited. You may have to add additional packages to the preamble if you use advanced LaTeX constructs.
- You must CITE any outside sources you use, including websites and other people with whom you have collaborated. You do not need to cite a CA, TA, or course instructor.
- Take care with time, we do not usually accept problem sets submitted late.
- Take care to upload the correct pdf with the correct images inserted in the correct places (if applicable).
- Check your pdf before upload.
- **Check your pdf before upload.**
- **CHECK YOUR PDF BEFORE UPLOAD.**

Quicklinks: 1 (2a) (2b) (2c) (3a) (3b) (3c) (3d) (4a) (4b) (4c)

Problem 1

Provide a one-sentence description of each of the components of a divide and conquer algorithm. Recall binary search and explain how the design of it conforms to the divide and conquer paradigm.

There are three main components for a divide and conquer algorithm which are the divide step, conquer step and combine step. the divide step will break a larger or difficult problem instance into a set of smaller instances of the same problem. Following is the conquer step in which if that smaller problem instance is trivial the algorithm will solve it directly, if not it will divide again. lastly the results from the smaller instances will be combined to produce a solution for the original larger instance. Recalling the binary search psuedocode in *Lec5.pdf* we can see the inplementation of these three steps. A single ascending sorted array is used as an input in which we want to find a certain value. Initially the mid point of this array is compared to the "key", if the "key" is larger than the midpoint all values of the array smaller than the "key" will be discarded and binary search will be recursively called on the array only including larger values. If the opposite occurs where the key is larger than the wanted value only the smaller values will be considered. This would include both the divide and conquer components of a divide and conquer algorithm. Where we will disregard all values greater than or less than the key because we know the solution does not lie in that range. This will repeat until the midpoint of each instance is equal to the key or no solution is found which covers the combine component, in which every recusive call uses the solutions of smaller instances to come closer to an answer.

Problem 2

For 2a, 2b, and 2c **you must** use the pseudocode for QUICKSORT and PARTITION on page 3 of Week3.pdf of the lecture notes and the array $A = [2, 6, 5, 7, 1, 9, 4]$. Suppose that we start by calling QUICKSORT$(A, 1, 7)$.

(2a) What is the value of the pivot in the call PARTITION$(A, 1, 7)$?

Going off of the psuedocode provided on page 3 of Week3.pdf the pivot is declared on line 2 and is chosen to be the last element in array A. For the first call the pivot will be equal to A[7] or $p = 4$.

For 2a, 2b, and 2c **you must** use the pseudocode for Quicksort and Partition on page 3 of Week3.pdf of the lecture notes and the array $A = [2, 6, 5, 7, 1, 9, 4]$. Suppose that we start by calling Quicksort$(A, 1, 7)$.

(2b) What is the index of the pivot value (returned) at the end of that call to Partition?

```
Initially a call to Quicksort is made
QuickSort(A,1,7)
1 < 7 is true, so a call to Partition is made
Partition(A,1,7)
p = A[7] = 4
iteration 1:
i = 1, j = 1
A[1] <= 4, Exchange A[1] and A[1], A = [2,6,5,7,1,9,4]
i = 2
iteration 2:
i = 2, j = 2
A[2] is not less than 4
iteration 3:
i = 2, j = 3
A[3] is not less than 4
iteration 4:
i = 2, j = 4
A[4] is not less than 4
iteration 5:
i = 2, j = 5
A[5] <= 4, Exchange A[2] and A[5], A = [2,1,5,7,6,9,4]
i = 3
iteration 6:
i = 3, j = 6
A[6] is not less than 4
i = 3, j = 7
loop terminates and A[3] is exchanged with A[7] A = [2,1,4,7,6,9,5]
i = 3 is returned back to quicksort making the index of the pivot value returned 3.
```

For 2a, 2b, and 2c **you must** use the pseudocode for QUICKSORT and PARTITION on page 3 of Week3.pdf of the lecture notes and the array $A = [2, 6, 5, 7, 1, 9, 4]$. Suppose that we start by calling QUICKSORT$(A, 1, 7)$.

(2c) On the next recursive call to QUICKSORT, what subarray does PARTITION evaluate? Give the state of the whole of A at the start of this call *and* the indices specifying the subarray.

The next recursive call to quicksort will evaluate A from s to q-1. which is all elements smaller than the pivot provided in the first quicksort call, 4. The state of A at the start of this call is A = [2,1,4,7,6,9,5] and the subarray to be evaluated will be A[1 -> (3 - 1)] or A' = [2 , 1]

Problem 3

Recall that the PARTITION algorithm encodes a fixed choice of pivot: $A[e]$ is always chosen. In this problem we ask you to consider alternative implementations of PARTITION that differ in how they choose the pivot.

(3a) Given an array consisting of n distinct elements, what choice of pivot will result in the best partitioning, and which one will result in the worst partitioning? Justify your answers in terms of the shape of the recursion tree and the running time of QUICKSORT subject to these pivot choices. You do not need to give a full, formal proof of any statement about running time.

---

textbfBest Partitioning:
Using the psuedocode provided in Week3.pdf, if a pivot is chosen where the partion creates two equally sized subarrays this would result in the best case time complexity for QuickSort. The recurrsion tree below shows this pivot selection

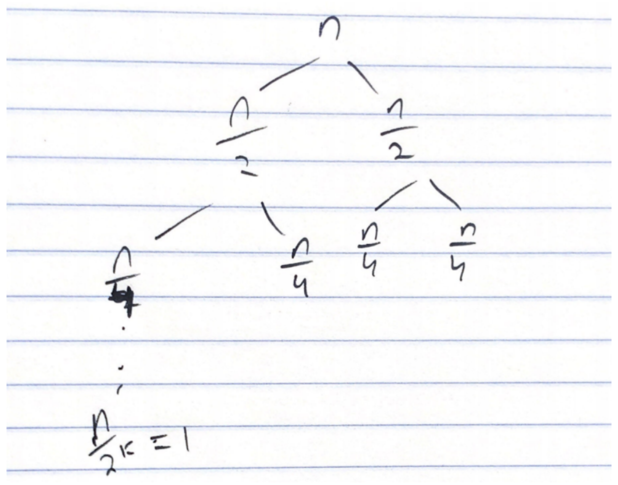

Figure 1: median pivot

As you can see, each partion with this pivot will create two equally sized sub-arrays causing the recursion tree to be balanced. The reccurence relation for runtime complexity of this pivot selection is represented as

$$T_{best}(n) = 2T(\frac{n}{2}) + \theta(n)$$

where $\theta(n)$ represents the time complexity of the pivot selection. Solving this recurrence relation using masters theorem we have, $a = b = 2$ and the reccurence follows case two. where $g(n) = \theta(n^{log_2 2}) = \theta(n)$ which corresponds to a time complexity of

$$T_{best}(n) = \theta(n^{log_2 2} log(n)n) = \theta(nlog(n))$$

Worst Partitioning:
Using the psuedo code provided in Week3.pdf, if a descending array is used as an input and the smallest element is used as a pivot for each partition call this would result in the worst time complexity for QuickSort. The recursion tree below shows this pivot selection

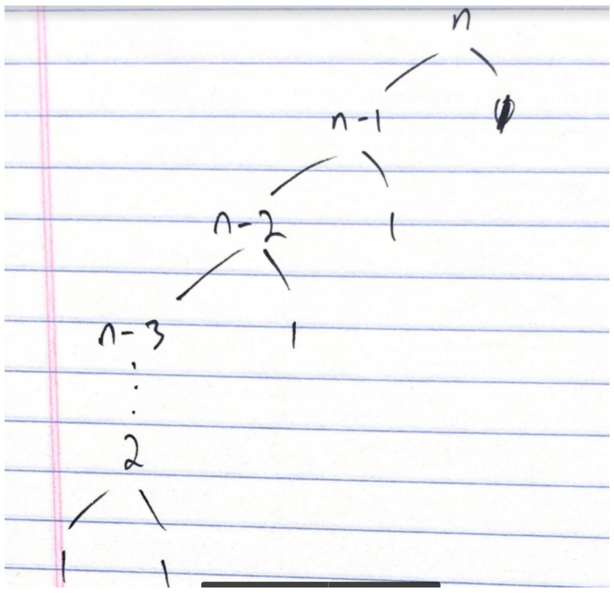

Figure 2: worst case pivot

As you can see, each partition using the minimum or maximum element will cause the recursion tree to have the largest unbalance between the size of each subarray. The recurrance relation for run time complexity of this pivot selection is shown below

$$T_{worst}(n) = T(n-1) + T(0) + \theta(n)$$

where $\theta(n)$ represents the time complexity of the pivot selection. Solving this recurrence relation using the unrolling method we obtain a run-time complexity of

$$T_{worst}(n) = \theta(n^2)$$

Recall that the Partition algorithm encodes a fixed choice of pivot: $A[e]$ is always chosen. In this problem we ask you to consider alternative implementations of Partition that differ in how they choose the pivot.

(3b) Suppose that we modify Partition(A, s, e) so that it chooses the median element of $A[s..e]$ in calls that occur in nodes of even depth of the recursion tree of a call Quicksort$(A, 1, \text{len}(A))$, and it chooses the maximum element of $A[s..e]$ in calls that occur in nodes of odd depth of this recursion tree. Assume that the running time of

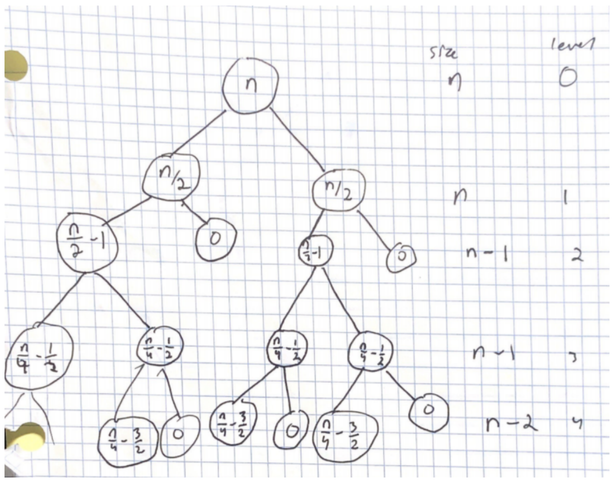

Figure 3: modified partition

this modified Partition is still $\Theta(n)$ on any subarray of length n. You may assume that the root of a recursion tree starts at level 0 (which is an even number), its children are at level 1, etc.

Write down a recurrence relation for the running time of this version of Quicksort given an array n distinct elements and solve it asymptotically, i.e. give your answer as $\Theta(f(n))$ for some function f. Show your work.

As stated in 3b, the time complexity for worst pivot selection is $T_{worst}(n) = \theta(n^2)$ and for the best case $T_{best}(n) = \theta(nlog(n))$. Below shows the recusion tree if we were to chose the median element for calls in nodes of even depth and maximum element for odd depths.

Althougth not proven, it does appear that if the maximum element is chosen and the partition between each array creates the largest difference in sub-array size, the time complexity of this modification seems be double of the best case pivot choice. Restating the recurrance relations for the best and worst pivot choices

$$T_{best}(n) = 2T(\frac{n}{2}) + \theta(n)$$

$$T_{worst}(n) = T(n-1) + T(0) + \theta(n)$$

We can combine both recurrence relations to obtain a recurrence relation for our modified partition.

$$T_{modified}(n) = 2T\left[(\frac{n}{2} - 1) + O(\frac{n}{2})\right] + O(n)$$

simplifying the above relation we notice that combining $O(\frac{n}{2} + n)$ that this relation is still bounded by $O(n)$ while the other relation $T(\frac{n}{2} - 1)$ is similar to our worst case scenerio but improved slightly making our run time complexity of $T_{modefied} = O(nlog(n))$.

Recall that the PARTITION algorithm encodes a fixed choice of pivot: $A[e]$ is always chosen. In this problem we ask you to consider alternative implementations of PARTITION that differ in how they choose the pivot.

(3c) Compare the running time of the modified version of QUICKSORT given by your answer to (3b) with the running time of the usual QUICKSORT (as stated in Week3.pdf) under the assumption that the median element is the pivot every time that we did in class.

If the two running times are Θ of each other then compare the implicit constant hidden in the Θ notation with a short non-rigorous explanation.

If the two running times are not Θ of each other, identify the one that is asymptotically larger and explain why with a short, non-rigorous explanation.

Looking at both run time complexities

$$T_{best}(n) = 2T(\frac{n}{2}) + O(n)$$

$$T_{modefied} = 2T(\frac{n}{2} - 1) + O(n + \frac{n}{2})$$

Because each running times are Θ of each other we can find the implicit constant. We know that using unrolling theorem we get that the base case for best pivot choice will occur when $\frac{n}{2^k} = 1$. so we suspect that $T_{best}(n) \leq C[log(n)] + C'$, For the modified case we will reach a base case when $\frac{n}{2^k} - \frac{2^k-1}{2^{k-1}} = 1$. Solving for n we obtain that a base case occurs when $n = 3 * 2^k - 2$ so we can suspect that $T_{mod}(n) \leq C[3log(n/2) - 2] + C'$ simplifying to

$$T_{mod}(n) \leq C[3log(n)] + C'$$

$$T_{best}(n) \leq C[log(n)] + C'$$

we can see that $T_mod(n)$ only differs by a constant of size 3, so that constant is $C = 3$.

Recall that the PARTITION algorithm encodes a fixed choice of pivot: $A[e]$ is always chosen. In this problem we ask you to consider alternative implementations of PARTITION that differ in how they choose the pivot.

(3d) Suppose now that we modify PARTITION so that the algorithm chooses the maximum as the pivot for three levels of recursion, then the median as the pivot at the next level, and this repeats so the maximum is the pivot for the next three levels, then the median for one level, etc.

What is your estimate of the asymptotic running time? We expect you to provide an explanation of your answer, but (unlike the previous subproblems) we do not require a formal proof and we do not require you to state or solve a specific recurrence relation for this subproblem.

If we were to modify partition so that the maximum is chosen as the pivot for three levels and then the median for the following level and this is repeated over time, i believe the run time complexity for this algorithm will still be $T(n) = O(nlog(n))$. Using Big-O as an asymptotic bound is not accurate but this modification is similar to the one used in 3b. It is just a slight improvement of the worst case run time complexity $T_{worst}(n)$ with a small improvement every 4th pivot selection. Looking at the recurrence tree this solution could possibly be $T_{mod}(n) = 2T(\frac{n}{6} - 1) + O(n)$.

Problem 4

Consider a chaining hash table A with b buckets that holds data from a fixed, finite universe U.

(4a) State the simple uniform hashing assumption and give one reason why this is a useful assumption and one reason why it is a poor assumption. Use full sentences.

The simple uniform hashing assumption assumes that every bucket has the same number of elements assigned to it. In other words we assume the hashing function maps the same number of keys in the universe to each bucket and the keys arrive uniformly at random from the universe.

Consider a chaining hash table A with b buckets that holds data from a fixed, finite universe U.

(4b) Recall the definition of worst-case analysis, and consider starting with A empty and inserting n elements into A under the assumption that $|U| \leq bn$. **Do not assume the simple uniform hashing assumption for this subproblem.**

(i) What is the worst case for the number of hash collisions? Give an exact answer and justify it.

(ii) Calculate the worst-case amortized cost of these n insertions into A, and give your answer as $\Theta(f(n))$ for a suitable function f. Justify your answer.

---

i) The worst case for the number of hash collisions would occur when all keys are stored in a single bucket. because we do not assume simple uniform hashing every bucket is allowed to store a non - uniform amount of keys. This makes it possible for a hash function to map all keys to a single bucket if using lists this would be a list of size n. the run time to parse through a linked list every time would be O(n).
ii) The worst-case amortized cost of these n insertions into A would be O(1), this is due to the fact that we assume the hasing function h is decent enough where the majority of insertions take O(1) time where the amount of insertions that take longer is miniscule.

Consider a chaining hash table A with b buckets that holds data from a fixed, finite universe U.

(4c) Consider the task of designing a data structure to hold the percentage score as an integer in the range 0..100 for every student in the course *DTDJ4215: Algorithms* that has 256 students. No course at *CU Cpvmefs*, where you work, can ever have more than 256 enrolled students.

The requirements are that reasonably efficient implementations of insertion and printing the scores in increasing order are important, and to have low memory overhead. Your colleague suggests that a dynamically-sized open addressing hash table with a highly sophisticated *disfrobunation* probing strategy will be a good implementation.

Your are knowledgeable enough to know of a better option. Choose an alternative data structure write a (polite) email to your colleague that explains your choice of data structure and why it fits the requirements better.

Hello fellow collegue,
It has come to my attention that you decided to not consult me about the construction of the data structure to hold our students score as an integer in the range of 1 to 100! There is a better option and will be much simpler than this dynamic gibberish you are babbling about. We could just use a 2d array aka a matrix to store both a students ID and percentage in a single column below is a representation. We already know the maximum size of the class,

index	0	1	2
Student Id	1	2	me
Percentage	90	50	10

so adding students, accessing student information or altering students information when the students id is known can be completed in constant time. using a hash table would increase the ammount of memory allocation needed to handle hash collisions and to store pointers where as with an array we could just write a simple UI to enter the students number and return the students percentage.

best, Connor