Algorithms (6470) HW02

Alex Darwiche

June 28, 2024

Answers

Q1

(a) Show how partition function would work on Array A<13,19,9,5,12,8,7,4,21,2,6,11>.

A =	13	19	9	5	12	8	7	4	21	2	6	11

i	p & j											r, x
	13	19	9	5	12	8	7	4	21	2	6	11

i	p	j										r, x
	13	19	9	5	12	8	7	4	21	2	6	11

i	p,i+1		j									r, x
	9	19	13	5	12	8	7	4	21	2	6	11

	p, i	i+1		j								r, x
	9	5	13	19	12	8	7	4	21	2	6	11

	p	i			j							r, x
	9	5	13	19	12	8	7	4	21	2	6	11

	p	i	i+1			j						r, x
	9	5	8	19	12	13	7	4	21	2	6	11

	p		i	i+1			j					r, x
	9	5	8	7	12	13	19	4	21	2	6	11

	p			i	i+1			j				r, x
	9	5	8	7	4	13	19	12	21	2	6	11

	p				i				j			r, x
	9	5	8	7	4	13	19	12	21	2	6	11

	p				i	i+1				j		r, x
	9	5	8	7	4	2	19	12	21	13	6	11

	p					i	i+1				j	r, x
	9	5	8	7	4	2	6	12	21	13	19	11

	p						i	i+1			j	r, x
	9	5	8	7	4	2	6	11	21	13	19	12

Q2

(a) Show that Quicksort's best case scenario is $\Omega(nlogn)$

(b) First, we need to show that $T(n) \geq cnlogn$

(c) This can be done by finding a c and n where: $T(n) = cnlogn + (something)$

(d) The best case scenario for quicksort is when each of the paritions is equal sized at each step. This would leave a recurrence relationship equivalent to: $T(n) = 2 * T(\frac{n}{2}) + \theta(n)$

(1) $T(n) \geq cnlogn$

(2) $T(\frac{n}{2}) \geq c\frac{n}{2}log\frac{n}{2}$

(3) Sub this back into original formula: $T(\frac{n}{2}) = 2 * (c\frac{n}{2}log\frac{n}{2}) + \theta(n)$

(4) Simplify: $T(\frac{n}{2}) = cn * (logn - log2) + n$

(5) Simplify: $T(\frac{n}{2}) = cnlogn - cn + n = cnlogn + n(1 - c)$

(6) Now we need to solve for where $(1 - c) > 0$ or $c < 1$ for all $n > 0$

Q3

(a) What do you expect the performance of these algorithms to be on the above dataset?

(1) Merge Sort: This sort is $O(nlogn)$ for all cases (Best, Worst, Average). Given this is a partially sorted dataset, this algorithm will do less comparisons than the average case, generally moving closer to the best

case scenario. This is because the merge operation can take a short cut when the first partition is all higher/lower than the second partition.

(2) Insertion Sort: This sort will do less comparisons than the average case, given the data is partially sorted. Given the data is almost completely sorted, I believe this will be $O(n)$.

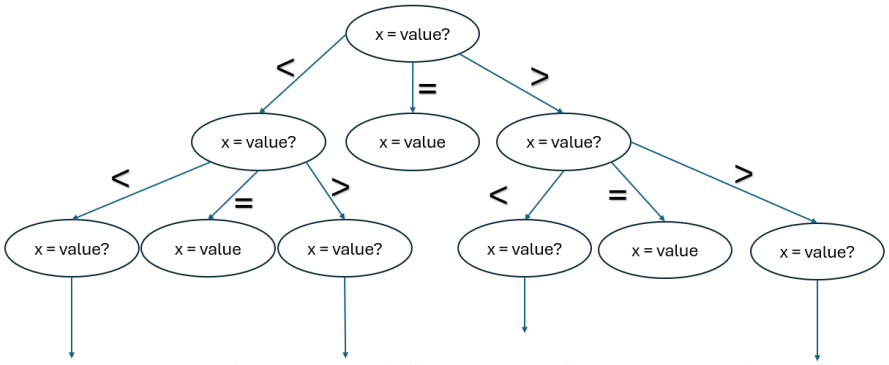
(3) Heap Sort: This data can quickly be arranged as a heap (there is no build heap needed). This however is still a $O(nlogn)$ and will does the same amount of comparisons as an other array.

(4) Quicksort: This sort will likely perform worse than average, given the data is already nearly sorted. Assuming we were to use the first or last element as a pivot, we would be getting heavily unbalanced partitions at each step. This might cause this sort to perform $O(n^2)$. This can be improved likely, with a better pivot selection algorithm if one exists.

(b) Given the data is nearly sorted, I would recommend Insertion Sort for this array. This should operate as $O(n)$.

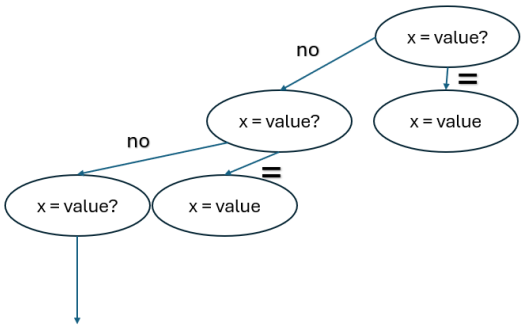
Q4

(a) Sorted List Decision Tree complexity



For a sorted list, there are 3 possibilities at each node. X is equal to the value or its greater than or less than that value. So, we can cut down the search space (greater than or less than) and continue searching. This "halving" of the search space at each level, will mean that we take log(n) steps to search the sorted list. This is shown by if there are 2 non-leaf nodes at each level, then there will be 2^i = n, where the number of levels i = logn.

(b) Unsorted List Decision Tree complexity

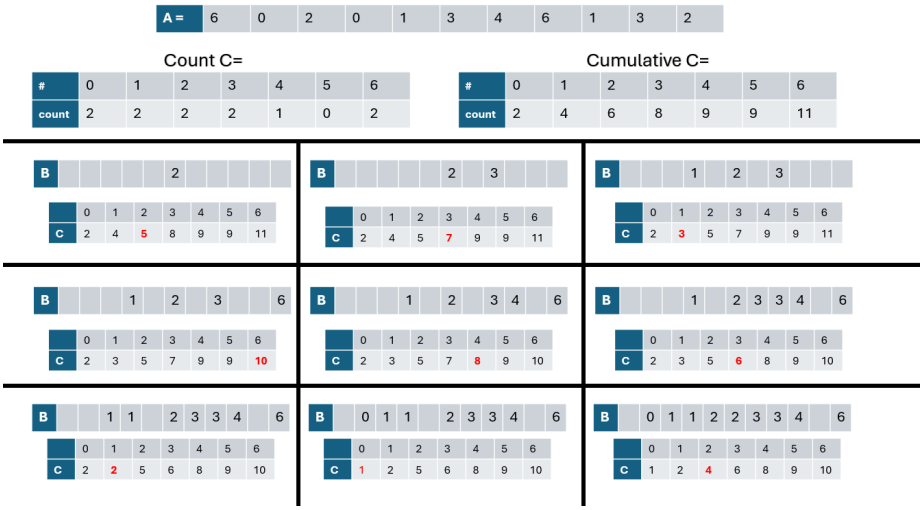


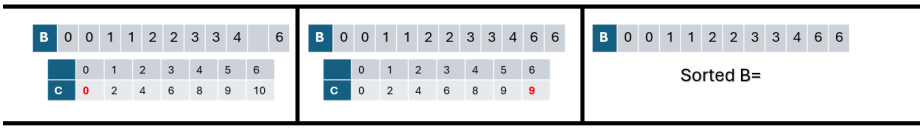
For an unsorted list, there are 2 possibilities at each node. X is equal to the value or its not equal to the value. This means that a search would need to iterate through the entire list to determine if each value is equal to the key.

(c) The lower bound of the complexity is $\Omega(log(n))$ for a sorted list, assuming that each decision point exactly splits the data in half at each step. For an unsorted list, you either need to search each value individually, which is $\Omega(n)$ or you need to sort the list first. Either way, the lower bound is greater than $\Omega(n)$ for an unsorted list. Thus, regardless of if the data is sorted or unsorted, the lower bound complexity will be $\Omega(log(n))$.

Q5

(a) Counting sort on Array A.

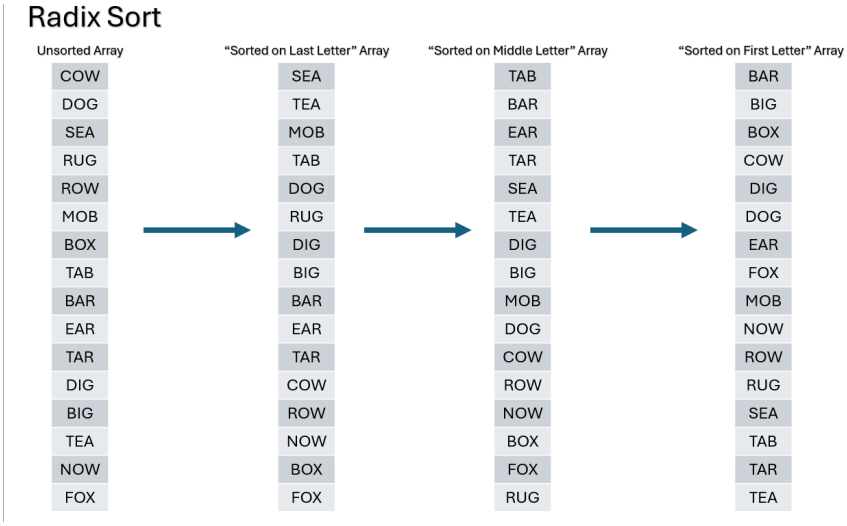




Q6

(a) Radix Sort on 3-letter words array.

(b)



Q7

(a) First, prove that groups of 9 would be $O(n)$ in the worst case.

(1) Assumptions: $g \leq \frac{n}{9}$, $9g$ = max total elements, n.

(2) Now, to find the worst case scenario, we will find the minimum number of elements less than our median of median x. This should be $\lfloor \frac{g}{2} \rfloor$ groups, with 5 elements in each group. We need to remove a few elements from this though, to include a partial column and the median itself. So total elements smaller than the median are $5 * \lfloor \frac{g}{2} \rfloor - (4 + 1)$.

(3) Next, Subtract from total $(9g - 4) - [5 * (\frac{g}{2}) - 5] \approx \frac{18g}{2} - \frac{5g}{2} = \frac{13g}{2}$

(4) We can sub back in $g \leq \frac{n}{9}$ to get $\frac{13n}{18}$

(5) Now, we rewrite our recurrence relationship as: $T(n) = T(\frac{n}{9}) + T(\frac{13n}{18}) + \Theta(n)$

(6) To show this is $O(n)$ we need to show that $T(n) \leq cn$

(7) $T(\frac{n}{9}) \leq c * \frac{n}{9}$

(8) $T(\frac{13n}{18}) \leq c * \frac{13n}{18}$

(9) Subbing back in: $T(n) = c * \frac{n}{9} + c * \frac{13n}{18} + n = c * \frac{15n}{18} + n = cn - c * \frac{3n}{18} + n = cn - n * (\frac{3c}{18} - 1)$

(10) Lastly, show that $\frac{3c}{18} - 1 > 0$, select $c > \frac{18}{3}$ for all n $>$ 0.

(b) Second, show that groups of 3 would not be $O(n)$ in the worst case, repeating all the steps in the first part of this problem.

(1) Assumptions: $g \leq \frac{n}{3}$, $3g$ = max total elements, n.

(2) Now, to find the worst case scenario, we will find the minimum number of elements less than our median of median x. This should be $\lfloor \frac{g}{2} \rfloor$ groups, with 5 elements in each group. We need to remove a few elements from this though, to include a partial column and the median itself. So total elements smaller than the median are $2 * \lfloor \frac{g}{2} \rfloor - (2 + 1)$.

(3) Next, Subtract from total $(3g - 2) - [2 * (\frac{g}{2}) - 3] \approx 3g - g = 2g$

(4) We can sub back in $g \leq \frac{n}{3}$ to get $\frac{2n}{3}$

(5) Now, we rewrite our recurrence relationship as: $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + \Theta(n)$

(6) To show this is not $O(n)$ we need to show that $T(n) > cn$.

(7) $T(\frac{n}{3}) \leq c * \frac{n}{3}$

(8) $T(\frac{2n}{3}) \leq c * \frac{2n}{3}$

(9) Subbing back in: $T(n) = c * \frac{n}{3} + c * \frac{2n}{3} + n = c * n + n$

(10) This shows that we will be unable to make this $T(n) \leq cn$, so groups of 3 are not linear.

Q8

(a) Demonstrate the Hoare Partition:

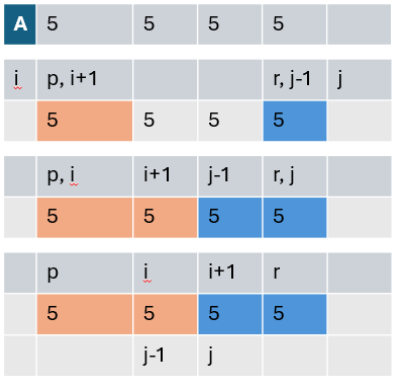
A	13	19	9	5	12	8	7	4	11	2	6	21	
i	p, i+1										j-2	r, j-1	j
	6	19	9	5	12	8	7	4	11	2	13	21	
	p, i	i+1								j-1	j	r	
	6	2	9	5	12	8	7	4	11	19	13	21	
	p	i	i+1	i+2	i+3	i+4	i+5	i+6	i+7	i+8		r	
	6	2	9	5	12	8	7	4	11	19	13	21	
									j-1	j			

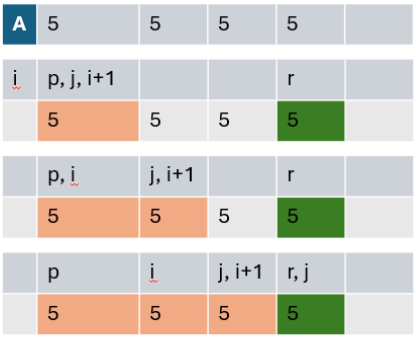
(b) Compare Hoare Partition to normal Partition.

(1) The Hoare partition includes the pivot in the arrays that it returns. The return "j" is simply the bound of the lower pivot in Hoare

partition, rather than the position of the pivot value in the normal partition. The Hoare partition works from the front and the back at the same time, whereas the other partition function starts from p and goes till r-1. The Hoare Partition compares each A[i] and A[j] to a value x, rather than comparing two elements of the array together at each step, a subtle difference in their comparisons.

(c) Compare when all the elements are equal.





(1) The difference in the total number of steps is equivalent in these two algorithms. The difference in is the returned arrays that get called in the next recurrsive call. Hoare partition appears to return 2 equally (within +1) sized arrays, whereas the normal partition function will return one array that is n-1 length and another that is empty. This will cause the Hoare Partition to perform near best case for quicksort, around $O(nlogn)$ and will cause a Normal Partition quicksort to perform near an $O(n^2)$ complexity.

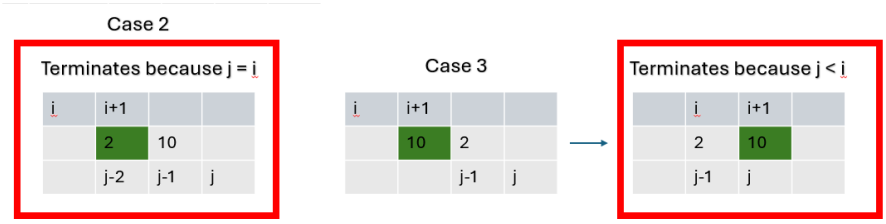
(2) This likely means that a HOARE-PARTITION is good in practice on datasets that have many duplicates.

(d) Assuming A[p..r] contains at least two elements, prove i and j never leave A.

(1) The Hoare Partition returns when i is greater than or equal to j. So, this means that there are 3 cases can occur in an array.

(2) Case 1: The array contains many elements and there are swaps that occur between the indices of i and j. These two indices will eventually crossover one another and terminate at that point, somewhere in the middle of the Array A.

(3) Case 2+3: These cases are the edge cases where there are only 2 elements in the array. As you can see in the picture below, the indexing of i and j will cause the recursion to terminate before either i or j exit the bounds of Array A. The red boxes indicate a "terminating" state.



(e) Prove the return j value is: $p \leq j < r$.

(1) Given i and j never leave the Array A, and that j is incremented down by 1 (at least) each loop. We can say that any Array of size greater than 2, j will be less than r and greater than or equal to p.

(2) The above graphic proves this point for arrays that are size 2, and the same intuition can be extended to larger arrays. j will increment down at least twice in all HOARE-PARTITION'ing.

(3) Perhaps more explicity, in the first iteration, i will always be stopped at the pivot (pivot = A[p]). This means that j will move left until it finds a value that is less than the pivot. If that first value is less than or equal to the pivot, a swap will occur. This then means that that the next iteration, j will decrement at least 1 more, and it will be less than r. If that first value is not less than or equal to the pivot, then j will have been decremented at least twice, again being lower than r. If j is unable to find a value that is less than or equal to the pivot, it will continue until the pivot. This will mean that i and j are equal (in the first iteration) and the Partition will return a j that is equal to p.

Q1 (Graduate students only)

(a) What happens to QUICKSORT if all values are the same.

(1) As seen in the above Question 8, we see that the PARTITION function returns 2 partitions, one that is size n-1 and another that is size 0.

(2) This means that the time complexity is $T(n) = T(n-1) + T(0) + \Theta(n) = T(n) = T(n-1) + \Theta(n)$

(3) Per the master theorem, we can see this is $O(n^2)$, where a = 1, b = 1, and d=1.

(4) To show that $O(nlogn)$ does nto work, we'd use $T(n) \leq cnlogn$ and $T(n-1) \leq c(n-1)log(n-1)$.

(5) Subbing back in: $T(n) = c(n-1)log(n-1) + kn$. Given $nlog(n-1) \leq nlogn$, we can sub that in and get $T(n) = cnlog(n) + kn$.

(6) As we can see here, there is no way to make $cnlog(n) + kn \leq cnlog(n)$.

(b) How to improve the complexity of the QUICKSORT so it is still $O(nlogn)$.

(1) Rather than using the less than or equal to in PARTITION, we could instead create 3 different different brackets where 1. Less Than Pivot, 2. Equal to Pivot, and 3. Greater than Pivot. This will allow the PARTITION function to pass back "complete" when it finds that the size of partitions 1 and 3 are both empty in part a. In my data structures course, we learned quicksort with this approach, essentially grouping all duplicates of the pivot in the middle of the array.

Q2 (Graduate students only)

(a) Placeholder

```
def RANDOMIZED-SELECT-ITERATIVE(A, p, r, i):
FLAG = TRUE

while (FLAG == TRUE)
    if p == r:
        return A[p]
    q = RANDOMIZED-PARTITION(A, p, r)
    k = q - p + 1
    if i == k:
        FLAG = FALSE
        value = A[q]
    else if i < k:
        p = p
        r = q-1
        i = i
    else:
        p = q+1
        r = r
        i = i-k
return value
```