Theory of Algorithms Homework 4

Evan Dreher

October 2022

1 Problem 1

Suppose we have an N-element array being passed back and forth in some recursive function. We usually assume it's passed *by reference* - that is, a pointer is passed, taking contant time. But it might be passed in other ways. We might, for instance, pass the array *by value* - copying the entire array, in $\Theta(N)$ time, and passing that into the recursive call. We might also copy *only the portion of the array* of interest - we might say that this takes $\Theta(m)$ times, where m is the length of the portion of interest.

(a) Suppose we have a binary search algorithm similar to the following. What will its runtime be under the three strategies? Why?

Algorithm 1 Binary-Search(A, x, low, high)

```
if low ≥ high then
    return high
medium = ⌊(low+high)/2⌋
if x ≤ A[medium] then
    return Binary − Search(A, x, low, medium)
else
    return Binary − Search(A, x, medium + 1, high)
```

- ***By pointer in*** $\Theta(1)$ ***time***
  As established in previous homeworks, the loop for a binary search algorithm will execute $\lg(n)$ times. It may be less than that, but since this implementation doesn't have an early termination condition it will always be $\lg(n)$. Everything performed in the pseudocode above is done in constant time (if-else checks, calculating the medium, etc), so if passing the array is also constant time then the runtime will simply be the number of loops times the work per loop which is $\lg(n)$ times a constant making it $\Theta(lg(n))$.
- ***By value in*** $\Theta(n)$ ***time***
  With this implementation, we are increasing the work per loop by $\Theta(n)$. Thus the new runtime is the number of loops times the work per loop making it $\Theta(n\ lg(n))$.
- ***By portion of array in*** $\Theta(m)$ ***times***
  With this implementation, every iteration (except the last one) we make on recusrive call on half the array of the current iteration and the work per loop is proportional to half the size of the array in the current iteration. This gives us the recurrence $T(n) = T(\frac{n}{2}) + n$.

$a = 1$
$b = 2$
$n^{\log_2(1)} = n^0$
$f(n) = n^{0+\epsilon}$ for $\epsilon = 1$
$\therefore$ by MM case 1, $T(n) = \Theta(n)$

(b) Now consider merge sort as we've discussed in the notes. What will its runtime be under the three strategies? Why?

- ***By pointer in*** $\Theta(1)$ ***time***
  The work per recursive call in mergesort is $\Theta(n)$. Adding a constant amount of work to that will not make the Theta bounds change, so the runtime will simply be $\Theta(n\ lg(n))$ (the regular runtime of merge sort).
- ***By value in*** $\Theta(n)$ ***time***
  The merge sort recursion is $T(n) = 2T(\frac{n}{w}) + n$. Thus it creates a recursion tree of height $lg(n)$. Each level of the tree has a number of nodes equal to twice the number of nodes in the previous level. Thus the total number of nodes (i.e calls to merge sort) is $2^{\log_2(n)} - 1 = n - 1$. Each call now has to do an extra n work so the total work of the algorithm equals the total number of recursive calls times the work to copy by value per recursive call plus the $n\ lg(n)$ work that mergesort normally does.

$$
\begin{aligned}
f(n) &= n^2 - n + n\ lg(n) \\
&\leq n^2 + n\ lg(n) \\
&\leq n^2 + n^2 && \text{since } n > lg(n) \text{since } n > 2 \\
&= 2n^2
\end{aligned}
$$

Thus, $f(n) = O(n^2)$ for $c = 2$, and $n_0 = 2$

$$
\begin{aligned}
f(n) &= n^2 - n + n\ lg(n) \\
&\geq n^2 - n \\
&= n^2 - \frac{1}{2}2n \\
&\geq n^2 - \frac{1}{2}n^2 && \text{since } n > 2 \\
&= \frac{1}{2}n^2
\end{aligned}
$$

Thus, $f(n) = \Omega(n^2)$ for $c = 2$, and $n_0 = \frac{1}{2}$

$$\therefore f(n) = \Theta(n^2)$$

- ***By portion of array in*** $\Theta(m)$ ***times***
  With this implementation, every iteration (except the last one) we make two recusrive calls on each half the array of the current iteration and the work per loop is copying

the left half, copying the right half, and then merging both halves. This gives us the recurrence $T(n) = 2T(\frac{n}{2}) + \frac{1}{2}n + \frac{1}{2}n + n = 2T(\frac{n}{2}) + 2n$.

$a = 2$
$b = 2$
$n^{\log_2(2)} = n^1$
$f(n) = 2n^1 = \Theta(n)$
$\therefore$ by MM case 2, $T(n) = \Theta(n\ lg(n))$

2 Problem 2

Consider the following array A,

$$[5, 3, 2, 7, 8, 1, 6, 4]$$

We have an initial mergesort call for this function of mergesort(A, 1, 8). List all calls that get made to mergesort or merge, in order. Every time a call to merge is made, rewrite the entire eight element array that results from the call to merge. highlight or bold any values that changed since your last array.

MS for mergesort call
M for merge call

$1.MS(A, 1, 8)$	
$2.MS(A, 1, 4)$	
$3.MS(A, 1, 2)$	
$4.MS(A, 1, 1)$	
$5.M(A, 1, 1)$	$\{5, 3, 2, 7, 8, 1, 6, 4\}$
$6.MS(A, 2, 2)$	
$7.M(A, 2, 2)$	$\{5, 3, 2, 7, 8, 1, 6, 4\}$
$8.M(A, 1, 2)$	$\{\mathbf{3,\ 5}, 2, 7, 8, 1, 6, 4\}$
$9.MS(A, 3, 4)$	
$10.MS(A, 3, 3)$	
$11.M(A, 3, 3)$	$\{3, 5, 2, 7, 8, 1, 6, 4\}$
$12.MS(A, 4, 4)$	
$13.M(A, 4, 4)$	$\{3, 5, 2, 7, 8, 1, 6, 4\}$
$14.M(A, 3, 4)$	$\{3, 5, 2, 7, 8, 1, 6, 4\}$
$15.M(A, 1, 34)$	$\{\mathbf{2,\ 3,\ 5}, 7, 8, 1, 6, 4\}$
$16.MS(A, 5, 8)$	
$17.MS(A, 5, 6)$	
$18.MS(A, 5, 5)$	
$19.M(A, 5, 5)$	$\{2, 3, 5, 7, 8, 1, 6, 4\}$
$20.MS(A, 6, 6)$	

21.$M(A, 6, 6)$	$\{2, 3, 5, 7, 8, 1, 6, 4\}$
22.$M(A, 5, 6)$	$\{2, 3, 5, 7, \mathbf{1,\ 8}, 6, 4\}$
23.$MS(A, 7, 8)$	
24.$MS(A, 7, 7)$	
25.$M(A, 7, 7)$	$\{2, 3, 5, 7, 1, 8, 6, 4\}$
26.$MS(A, 8, 8)$	
27.$M(A, 8, 8)$	$\{2, 3, 5, 7, 1, 8, 6, 4\}$
28.$M(A, 7, 8)$	$\{2, 3, 5, 7, 1, 8, \mathbf{4,\ 6}\}$
29.$M(A, 5, 8)$	$\{2, 3, 5, 7, \mathbf{1,\ 4,\ 6,\ 8}\}$
30.$M(A, 8, 8)$	$\{\mathbf{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8}\}$

3 Problem 3

Consider the array from the previous problem. Show the arrays that result from calling Build-Max-Heap on that array, showing every step; at each step, highlight or bold any values that have changed since the last step.

1.$\{5, 3, 2, 7, 8, 1, 6, 4\}$
2.$\{5, 3, \mathbf{6}, 7, 8, 1, \mathbf{2}, 4\}$
3.$\{5, \mathbf{8}, 6, 7, \mathbf{3}, 1, 2, 4\}$
4.$\{\mathbf{8,\ 5}, 6, 7, 3, 2, 1, 4\}$
5.$\{8, \mathbf{7}, 6, \mathbf{5}, 3, 2, 1, 4\}$

4 Problem 4

Consider the following array A,

$$\{8, 7, 6, 5, 4, 3, 2, 1\}$$

Assum this array was constructued using Build-Max-Heap as the first step of heapsort. Show the array that will result after each change that heapsort will make to the array after that point; again, highlight or bold each change.

1.$\{8, 7, 6, 5, 4, 3, 2, 1\}$
2.$\{\mathbf{1}, 7, 6, 5, 4, 3, 2, \mathbf{8}\}$
3.$\{\mathbf{7}, \mathbf{1}, 6, 5, 4, 3, 2, 8\}$
4.$\{7, \mathbf{5}, 6, \mathbf{1}, 4, 3, 2, 8\}$
5.$\{\mathbf{2}, 5, 6, 1, 4, 3, \mathbf{7}, 8\}$
6.$\{\mathbf{5}, \mathbf{2}, 6, 1, 4, 3, 7, 8\}$

7.$\{5, \mathbf{4}, 6, 1, \mathbf{2}, 3, 7, 8\}$
8.$\{\mathbf{6}, 4, \mathbf{5}, 1, 2, 3, 7, 8\}$
9.$\{\mathbf{3}, 4, 5, 1, 2, \mathbf{6}, 7, 8\}$
10.$\{\mathbf{4}, \mathbf{3}5, 1, 2, 6, 7, 8\}$
11.$\{\mathbf{5}, 3, \mathbf{4}, 1, 2, 6, 7, 8\}$
12.$\{\mathbf{2}, 3, 4, 1, \mathbf{5}, 6, 7, 8\}$
13.$\{\mathbf{3}, \mathbf{2}, 4, 1, 5, 6, 7, 8\}$
14.$\{\mathbf{4}, 2, \mathbf{3}, 1, 5, 6, 7, 8\}$
15.$\{\mathbf{1}, 2, 3, \mathbf{4}, 5, 6, 7, 8\}$
16.$\{\mathbf{2}, 1, \mathbf{3}, 4, 5, 6, 7, 8\}$
17.$\{\mathbf{3}, 1, \mathbf{2}, 4, 5, 6, 7, 8\}$
18.$\{\mathbf{2}, 1, \mathbf{3}, 4, 5, 6, 7, 8\}$
19.$\{\mathbf{1}, \mathbf{2}, 3, 4, 5, 6, 7, 8\}$
20.$\{1, 2, 3, 4, 5, 6, 7, 8\}$

5 Problem 5

Consider the following array A,

$$\{5, 7, 8, 2, 3, 1, 9, 4, 6\}$$

Show the array that will result after each change that one call to partition will make on it (in other words, don't fully resolve quicksort, but do show each swap that one run of partition will make. Again, highlight or bold each chage.

Pivot = 6 (highest index in the range of quicksort call)

1.$\{5, 7, 8, 2, 3, 1, 9, 4, 6\}$
2.$\{5, \mathbf{2}, 8, \mathbf{7}, 3, 1, 9, 4, 6\}$
3.$\{5, 2, \mathbf{3}, 7, \mathbf{8}, 1, 9, 4, 6\}$
4.$\{5, 2, 3, \mathbf{1}, 8, \mathbf{7}, 9, 4, 6\}$
5.$\{5, 2, 3, 1, \mathbf{4}, 7, 9, \mathbf{8}, 6\}$
6.$\{\mathbf{5}, 2, 3, 1, 4, \mathbf{6}, 9, 8, \mathbf{7}\}$

6 Problem 6

Consider the following array A,

$$\{3, 8, 2, 3, 1, 8, 5, 3, 2, 6\}$$

Show the cumulative array that will be produced by Counting Sort. Then show how this cumulative array changes as the sorted array is filled in; that is, after each number is added to the

final sorted array, show both that sorted array and the modified cumulative array. Bold/hightlight any changes.

First, it will generate the keys as shown in the following array

$$\{1, 2, 3, 0, 1, 1, 0, 2\}$$

Then it will sum those numbers to get the cumulative array as follows

$$B = \{1, 3, 6, 6, 7, 8, 8, 10\}$$

Then it will start filling in C based on the values in B (C will be filled with 0s for visualization)

B (cumulative array)	C (sorted array)	N
{1,3,6,6,7,8,8,10}	{0,0,0,0,0,0,0,0,0,0}	init
{1,3,6,6,7,**7**,8,10}	{0,0,0,0,0,0,0,**6**,0,0}	10
{1,**2**,6,6,7,7,8,10}	{0,0,**2**,0,0,0,0,6,0,0}	9
{1,2,**5**,6,7,7,8,10}	{0,0,2,0,0,**3**,0,6,0,0}	8
{1,2,5,6,**6**,7,8,10}	{0,0,2,0,0,3,**5**,6,0,0}	7
{1,2,5,6,6,7,8,**9**}	{0,0,2,0,0,3,5,6,0,**8**}	6
{**0**,2,5,6,6,7,8,9}	{**1**,0,2,0,0,3,5,6,0,8}	5
{0,2,**4**,6,6,7,8,9}	{1,0,2,0,**3**,3,5,6,0,8}	4
{0,**1**,4,6,6,7,8,9}	{1,**2**,2,0,3,3,5,6,0,8}	3
{0,1,4,6,6,7,8,**8**}	{1,2,2,0,3,3,5,6,**8**,8}	1
{0,1,**3**,6,6,7,8,8}	{1,2,2,**3**,3,3,5,6,8,8}	2

7 Problem 7

You're back in Shinjuku, with your tuples representing the (ID, entrance time, exit time) of each of the 3.6 million times a person passed through the station in a week. Assume that the tuples were recorded in order of people's entrance into the station. You want to sort them by exit time. Assess your major algorithmic sorting choices (mergesort, heapsort, quicksort, counting sort) for this problem. Discuss the appropriateness of each for this specific data set. State any assumptions clearly.

Assumptions

1. The times are stored as an integer representing the number of seconds since 12:00 AM of the given week.
2. A tuple is stored as the following data type (64-bit int, 32-bit int, 32-bit int), for a total of 128-bits.

- **Counting sort**

  Counting sort runs in $\Theta(n + k)$, so with $n = 3.6$million and $k = 604,800$ (number of seconds in a week) it will run faster than an $n\ lg(n)$ algorithm. The memory consumption is also $\Theta(n + k)$ so with 3.6 million 128-bit tuples and a keys array of 604,800 32-bit integers, the total memory will be about 60 Megabytes of RAM which should be fine given the amount of RAM we have on our laptops. Therefore counting sort would likely be the best algorithm for this problem.

- **Merge sort**

  Merge sort runs in $\Theta(n\ lg(n)$, so it will take longer than counting sort. Furthermore it takes $\Theta(n)$ auxillary memory space so it will take about 57.6 MB which is not much less than counting sort and it will run significantly slower meaning this is likely a poor choice of algorithm for this data set.

- **Quick sort**

  Quick sort, like merge sort, runs in $\Omega(n\ lg(n))$ as an average case. But it has a worst case run time of $O(n^2)$ which happens when it tries to sort and already sorted array. Since the tuples are sorted by entry times, and each passenger spends similar amounts of time in the station, it is reasonable to assume that the array is close to being sorted by exit time already meaning quick sort would likely be picking bad pivots and rune close to $O(n^2)$ and so it will likely also not be a great pick.

- **Heap sort**

  Of the $n\ lg(n)$ algorithms, heap sort is the way to go despite its paging issues. However to save some time it should be implemented to build a min-heap rather than a max heap since the data is already approximately a min heap so less swaps will have to be done to configure it in such a manner.

8 Problem 8

Consider the partition code we presented in class,

Algorithm 2 Partition(A,p,r)

```
x = A[r]
i = p − 1
for j=p to r-1 do
    if A[j] ≤ x then
        i = i + 1
        swap A[i] with A[j]
exchange A[i + 1] with A[r]
```

We gave the following invariant for that code: *At the beginning of the loop where j = k, all elements of A from p...i are less than or equal to x, and all elements of A from i+1...j-1 are greater than x.*

- **Loop Invariant:** At the beginning of the loop where j = k, all elements of A from p...i are less than or equal to x, and all elements of A from i+1...j-1 are greater than x.

- **Initialization:** $j = p$. At the initialization $i = p - 1$ (i.e $j - 1$). The range $A[p...i]$ has no elements in it therefore it is trivially true that all elements in that range are less than x. Likewise $i + 1 = p$ and $j - 1 = p - 1$ so the range $A[i + 1...j - 1]$ also has no elements in it is trivially true that all elements in that range are greater than x.

- **Maintenance:** Assume the LI holds for the loop where $j = k$. If $A[k] \geq x$, then i will increment thus incrementing the range $A[p...i]$ as well. $A[i]$ and $A[k]$ are then swapped. By the LI, all elements in $A[p...i-1]$ are less than or equal to x. Since $A[k] \leq x$ and $A[k]$ is now stored at $A[i]$, then all elements in $A[p...i]$ are less than or equal to x. By the LI, at the start of this loop all elements in $A[i+1...k-1]$ are greater than x. Before i incremented and $A[k]$ and $A[i]$ were swapped, we know by the LI that all the elements in $A[i+1...k-1] \geq x$. After i incremented but before the swap, we knew that all elements in $A[i...k-1] \geq x$. After $A[i]$ and $A[k]$ were swapped we can now know that all elements in $A[i+1...k]$ are greater than x. Thus the LI will hold for the next loop. If instead $A[k] < x$, then nothing will happen for the rest of this loop. By the LI all elements in $A[p...i] \leq x$ and all elements in $A[i+1...k-1] \geq x$. Since $A[k] > x$, then all elements in $A[i+1...k] > x$ thus in either case the LI holds for the next loop.

- **Termination:** The loop ends when $j = r$. By the LI we know that all elements in $A[p...i] \leq x$ and all elements $A[i+1...r-1] \geq x$. The method then swaps $A[i+1]$ and $A[r]$ (where A[r] = x), thus all elements in $A[p...i+1] \leq x$ and all elements in $A[i+2] \geq x$. The method then returns $i+1$ which is the index that recursive calls of quicksort will sort around.