EE4371

NOHAN JOFMON

CH19B072

ASSIGNMENT - 3

(Everywhere, log n means $\log_2 n$)

1) To prove: Running time of merge-sort algorithm is $O(n \log n)$ for all n

Psuedocode of Merge-sort

```
merge (L, R):
    result = an array of length len (L) + len (R)   [n = len (L) + len (R)]

    i=0, j=0
    for k in [0, .... n-1]:
        if L[i] < R[j]:
            result [k] = L[i]
            i += 1
        else:
            result [k] = R[j]
            j += 1
    return result
```

↓

Running time of merge function:

First two steps $= O(1)$ [Assignment steps]

for loop $= O(n)$ [Loops through each value]

$\Rightarrow O(n) = O(\text{len}(L) + \text{len}(R))$

mergesort (A) :

n = length of ~~A~~ A

if n <= 1 :

return A

L = mergesort $(A[0 : \lfloor \frac{n}{2} \rfloor])$

R = mergesort $(A[\lfloor \frac{n}{2} \rfloor : n])$

return merge (L, R)

~~Running time of merge sort~~

~~First three lines → O(1) (Assignment steps)~~

Running time of merge sort (Recurrence relation)

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n)$$

Explanation:

$T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ = time for sorting left half

$T\left(\left\lceil \frac{n}{2} \right\rceil\right)$ = time for sorting right half

$O(n)$ = time for performing merge function

(merge (L, R) is of $O(\text{len}(L) + \text{len}(R))$

which is equal to $\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil$

$\Rightarrow O(n)$ )

Our next step is find explicit running time from the recurrence relation.

We split it into 2 cases:

Case 1: n is a power of 2

In this case $\lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil = \frac{n}{2}$

$\Rightarrow$ Our recurrence relation becomes:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

(we take $O(n)$ as $cn$)

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

$$T(n) = 2\left(2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right) + cn = 4T\left(\frac{n}{4}\right) + 2cn$$

$$T(n) = 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn = 8T\left(\frac{n}{8}\right) + 3cn$$

$\vdots$

iterating through the relation, we find that

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + icn \quad (i \rightarrow \text{iteration variable})$$

$\Downarrow$

when will this stop?

This stops when we reach ~~to~~ one element case (ie when no further iteration is possible).

Once it reaches the one element case, mergesort ~~[illegible]~~ function returns the array A itself.

In that case, last step is :

$$T(n) = 2^i T(1) + icn$$

ie $1 = \frac{n}{2^i} \Rightarrow i = \log_2 n$

(I denote it as simply $\log n$)

$$\Rightarrow T(n) = 2^i T\left(\frac{n}{2^i}\right) + icn$$

$$= 2^{\log n} T\left(\frac{n}{2^{\log n}}\right) + (\log n)cn$$

$$T(n) = nT(1) + cn\log n$$

We know that $T(1)$ is a constant, $(b)$ which is the time required to return the same array A (which has one element now)

$$\Rightarrow T(n) = nb + cn\log n$$

$$\Rightarrow T(n) = O(n\log n)$$ → for all $n$ which is a power of 2

Case 2 : $n$ is not a power of 2

(OR $n$ is any number)

Here, we can prove it by induction hypothesis.

As we already proved it for $n$ = power of 2, we can use that information as base case

~~[We take~~

We need to prove $T(n) = O(n \log n)$

$\Rightarrow T(n) \leq c\, n \log n$ for $n \geq n_0$

[Here $c$ is just a constant.

We take $c = 2k$

and $n_0 = 1$ ]

Induction Hypothesis :

$T(n) \leq 2k\, n \log n$ for all $n \geq 1$

Base case :

we have already proved in case 1 that

$T(n) = O(n \log n)$ for powers of 2

$\therefore$ We take base case as $n = 2$ (or 4, 8, 16 ....)

Induction step :

Assume that the Induction hypothesis holds true for all $m < n$

Now, we need to prove that it holds true for $m = n$

Proof

$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + kn$

(By induction,

$T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \leq 2k \left\lfloor \frac{n}{2} \right\rfloor \log\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$

& $T\left(\left\lceil \frac{n}{2} \right\rceil\right) \leq 2k \left\lceil \frac{n}{2} \right\rceil \log\left(\left\lceil \frac{n}{2} \right\rceil\right)$

as $\lceil \frac{n}{2} \rceil , \lfloor \frac{n}{2} \rfloor < n$ )

$$\Rightarrow T(n) \leq 2k \lfloor \tfrac{n}{2} \rfloor \log \lfloor \tfrac{n}{2} \rfloor + 2k \lceil \tfrac{n}{2} \rceil \log \lceil \tfrac{n}{2} \rceil + kn$$

$$\leq 2k \lfloor \tfrac{n}{2} \rfloor \log \lceil \tfrac{n}{2} \rceil + 2k \lceil \tfrac{n}{2} \rceil \log \lceil \tfrac{n}{2} \rceil + kn$$

$$\leq 2k \left( \lfloor \tfrac{n}{2} \rfloor + \lceil \tfrac{n}{2} \rceil \right) \log \lceil \tfrac{n}{2} \rceil + kn$$

$$\leq 2kn \log \left( \lceil \tfrac{n}{2} \rceil \right) + kn$$

[As $\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil = n$ ]

$$\Rightarrow T(n) \leq 2kn \log \left( \frac{2n}{3} \right) + kn$$

[As $\lceil \frac{n}{2} \rceil \leq 2n/3$ for all $n \geq 2$ ]

$$\Rightarrow T(n) \leq 2kn \log n + kn \left(1 - 2\log(3/2)\right)$$

$$\Rightarrow T(n) \leq 2kn \log n$$

[As $1 - 2\log\left(\frac{3}{2}\right) < 0$ ]

ie our hypothesis holds true for $m = n$

$\Rightarrow$ Our hypothesis is correct

$\Rightarrow$ By IH, we prove that

$T(n) = O(n \log n)$ for any $n$

2) Consider a modification of the deterministic version of the quick sort algorithm where we choose the element at index $\lfloor n/2 \rfloor$ as our pivot :

PSEUDOCODE :

```
Quicksort (A) :
        if len (A) <= 1 :
                return A
        pivot = element at index A ⌊n/2⌋
        Partition A into :
                L (less than pivot)   and
                R (greater than pivot)
        Replace A with [L, pivot, R]
        Quick sort (L)
        Quick sort (R)
```

Recurrence Relation for Quicksort

$$T(n) = T(|L|) + T(|R|) + \Theta(n)$$

Explanation :

$T(|L|) \longrightarrow$ Time for sorting L part

$T(|R|) \longrightarrow$ Time for sorting R part

$\Theta(n) \longrightarrow$ Time for partitioning A into [L, pivot, R]

( other ~~steps~~ steps are assignment steps that take constant time (independent of n) )

Also ,

$$T(0) = T(1) = O(1)$$

( If 0 or 1 elements in the array, we simply return the array $\longrightarrow$ constant time )

Now, we need a sequence that would cause this version of quicksort to run in $\Omega(n^2)$ time

Consider the situation when pivot = max (A) for all recursive calls

That means, we will have

$n-1$ terms in L

1 term in pivot (maximum of A)

0 terms in R

$\Rightarrow$ Recurrence relation becomes:

$$T(n) = T(0) + T(n-1) + \theta(n)$$

$$T(n) = T(n-1) + \theta(n)$$

[As $T(0) = O(1)$ and can be included in $\theta(n)$]

Solving the above recurrence relation (iterative method)

$$T(n) = T(n-2) + 2\theta(n)$$

$$T(n) = T(n-3) + 3\theta(n)$$

$$\vdots$$

this stops at

$$T(n) = T(n-n) + n\theta(n)$$

ie $T(n) = T(0) + n\theta(n)$

$$\Rightarrow T(n) = \theta(n^2)$$

Therefore, such a sequence will have complexity $\Omega(n^2)$ [ Specifically $O(n^2)$ ]

## Example of such a sequence

We need

$$\left.\begin{aligned} \text{pivot} &= \max(A) \\ \text{element at } \lfloor \tfrac{n}{2} \rfloor &= \max(A) \end{aligned}\right\} \text{ for all recursive calls}$$

(Note: We consider our index from 0 to n-1)

$$\underset{0}{[4}, \underset{1}{6}, \underset{2}{8}, \underset{3}{10}, \underset{4}{9}, \underset{5}{7}, \underset{6}{5}]$$

**Call - 1**

$L = [\underset{0}{4}, \underset{1}{6}, \underset{2}{8}, \underset{3}{9}, \underset{4}{7}, \underset{5}{5}]$ pivot = [10]

**Call - 2**

$L = [\underset{0}{4}, \underset{1}{6}, \underset{2}{8}, \underset{3}{7}, \underset{4}{5}]$ pivot = [9]

**Call - 3**

$L = [\underset{0}{4}, \underset{1}{6}, \underset{2}{7}, \underset{3}{5}]$ pivot = [8]

**Call - 4**

$L = [\underset{0}{4}, \underset{1}{6}, \underset{2}{5}]$ pivot = [7]

**Call - 5**

$L = [\underset{0}{4}, \underset{1}{5}]$ pivot = [6]

**Call - 6**

$L = [4]$ pivot = [5]

(In each case, circled element is the element at $\lfloor \frac{n}{2} \rfloor$ and is the pivot for the next call)

(In all cases $R = null$)

~~we observe that~~

In fact, in all cases when pivot (element at $\lfloor \frac{n}{2} \rfloor$) is equal to max (A) for each call, complexity is $\Omega(n^2)$

Alternatively

We can choose sequences whose pivot (element at $\lfloor \frac{n}{2} \rfloor$) is equal to min (A) for each call.

In this case, L will have 0 elements and R will have n-1 elements and therefore, it will still be of $\Omega(n^2)$

eg of such a sequence : [7, 5, 3, 1, 2, 4, 6]

↓

can be explained in a similar manner

3) In assignment 2 → question 6, I already provided a method to remove duplicates from an array A in time $O(n \log n)$, which is done in an efficient manner.

HENCE I AM NOT REPEATING THE SAME ANSWER

4) We have an array A ($len(A) = n$) with integer values in range $[0, n^2-1]$.

<u>Method</u>

We can use a modification of the popular Radix sort here

↓

How to ~~radi~~ modify Radix sort ?

Running time of Radix sort is $O(d(b+n))$

$d$ = no. of digits

$b$ = base of the numbers

In our case,

$d$ = no. of digits

= no. of digits of largest number possible

= no. of digits of $n^2 - 1$

$\therefore d \leq$ no. of digits of $n^2$

ie $d \leq \log_b(n^2)$ or $d \leq 2\log_b(n)$

ie $d$ can be maximum $2\log_b n$

(eg. if $n = 100$, ~~d can be~~

integers can range from 0 to $100^2 - 1$

0 to ~~10000~~ 9999

$d$ can be maximum $2\log_{10}(100) = 4$ )

$\Rightarrow$ Running time becomes $O(2\log_b(n)(b+n))$

But we need a running time $O(n)$

$\boxed{\therefore \text{ We choose } \quad b = n}$

$\Rightarrow$ Running time is $O(2 \log_n(n) \; (n+n))$

$O(2(2n))$

$O(n)$

Python implementation

```
def countsort (arr, n, exp) :
        output = [0] * n
        count  = [0] * n

        for  i  in range (n) :
            count [(arr[i] // exp) % n ] += 1

        for  i  in  range (1, n) :
            count[i] += count [i-1]

        for  i  in  range (n-1, -1, -1) :
            output [count [(arr[i] // exp) % n] - 1 ] = arr [i]
            count [(arr [i] // exp) % n] -= 1

        for  i  in  range (n) :
            arr [i] = output [i]


def  sort (arr, n) :

        count sort (arr, n, 1)
        count sort (arr, n, n)
```

~~temp~~

Testing code

```
arr = [1, 35, 14, 26, 11, 8]
n = len(arr)
print("Given array :", arr)
sort(arr, n)
print("Sorted array :", arr)
```

OUTPUT

```
Given array : [1, 35, 14, 26, 11, 8]
Sorted array : [1, 8, 11, 14, 26, 35]
```

Explanation

→ sort function

1: countsort(arr, n, 1) sorts arr based on Most significant digit (MSD).

Then,

2: countsort(arr, n, n) sorts arr based on Least significant digit (LSD)

⇒ Array gets sorted after the above two calls

(Note: Note that if $b = n$,

no. of digits $d$ can be maximum

$= 2\log_n(n) = 2$

(proved earlier)

Therefore, we need 2 calls for 2 digits of the number)

## → count sort function

We have three 'for' loops :

1: After implementation of first for loop, count[i] stores the no. of occurrences of the digit i at MSD or LSD

2: After 2nd for loop, count[i] is updated so that it now stores the position of the digit i in the sorted (output) array

3: Now we have the position of each digit in the output. Using that position, we build the output array from arr

## Running time of sort

$T(n)$ = time for 2 calls of count sort

For each call of countsort, there are three for loops $(O(n))$

$\Rightarrow T(n) = 2(O(n)) = \underline{\underline{O(n)}}$

## Example for illustration

array = [1, 35, 14, 26, 11, 8]

$n = 6$ (range : 0 to 35)

array = [ 1, 35, 14, 26, 11, 8 ]

$\Downarrow$ base 6

[01, 55, 22, 42, 15, 12]

After countsort (arr, n, 1) :

sorts based on digit

[01, 22, 42, 12, 55, 15] (base 6)

After countsort (arr, n, n) :

sorts based on digit

[01, 12, 15, 22, 42, 55] (base 6)

$\Downarrow$ base 10

Final result :

[1, 8, 11, 14, 26, 35]

## Conclusion

By using b=n in the Radix sort algorithm ; we can build an algorithm that sorts an array A of n integers in the range $[0, n^2-1]$ in $O(n)$ time

5) Quick sort - Pseudo code

```
Quick sort (A) :
    if   len (A) <= 1 :
            return A
    pivot = element at index q
    Partition A into :
            L (less than pivot) and
            R (greater than pivot)
    Replace A with [L, pivot, R]
    Quick sort (L)
    Quick sort (R)
```

Recurrence Relation

L → has q elements [0 to q-1]

R → has n-q-1 elements [q+1 to n-1]

pivot → has 1 element [q]

- Partitioning is $\Theta(n)$ because we should go through each element and check if they are greater than or less than pivot.
- other steps are simple $O(1)$ steps

$$\boxed{T(n) = \underbrace{T(q)}_{\text{sorting } L} + \underbrace{T(n-q-1)}_{\text{sorting } R} + \underbrace{\Theta(n)}_{\text{Partitioning}}}$$

( sorting pivot is $T(1) = O(1)$ )

Best case running time = min T(n)
(denoted by $T_b(n)$)

$$\boxed{T_b(n) = \min_{1 \le q \le n-1} \left( T(q) + T(n-q-1) \right) + \Theta(n)}$$

We guess that $T(n) \ge cn\log n$ for some const $c$

$$\Rightarrow T(n) \ge \min_{1 \le q \le n-1} \left( cq\log q + c(n-q-1)\log(n-q-1) \right) + \Theta(n)$$

$\hookrightarrow$ equation (1)

→ Finding $\min_{1 \le q \le n-1} \left( cq\log q + c(n-q-1)\log(n-q-1) \right)$

$$f(q) = cq\log q + c(n-q-1)\log(n-q-1)$$

$$f(q) = \frac{cq\ln q + c(n-q-1)\ln(n-q-1)}{\ln 2}$$

$$f'(q) = \frac{c\ln q + c - c\ln(n-q-1) - c}{\ln 2}$$

$$f'(q) = \frac{c\left[\ln q - \ln(n-q-1)\right]}{\ln 2}$$

$f'(q) = 0$ occurs at $q = \frac{n-1}{2}$

$$f''(q) = \frac{c}{\ln 2}\left(\frac{1}{q} + \frac{1}{n-q-1}\right)$$

Clearly,

$f''(q) > 0$ at $q = \frac{n-1}{2}$

$\Rightarrow q = \frac{n-1}{2}$ is a minimum of $f(q)$

Description

But case $\Longleftrightarrow$ ~~min~~ minimum $T(n) \Longleftrightarrow$ at $q = \frac{n-1}{2}$

ie But case occurs at even distribution among L and R

$$f\left(\frac{n-1}{2}\right) = c\left[\frac{n-1}{2}\log\left(\frac{n-1}{2}\right) + \left(n - \frac{(n-1)}{2} - 1\right) \log\left(n - \frac{(n-1)}{2} - 1\right)\right]$$

$$\boxed{f\left(\frac{n-1}{2}\right) = c(n-1)\log\left(\frac{n-1}{2}\right)} \rightarrow \text{min of } f(q)$$

$\rightarrow$ Substituting the above value in equation (1)

$$\Rightarrow T(n) \geq c(n-1)\log\left(\frac{n-1}{2}\right) + \Theta(n)$$

$$= c(n-1)\log(n-1) - c(n-1)\log_2 2 + \Theta(n)$$

$$= c(n-1)\log(n-1) - c(n-1) + \Theta(n)$$

$$= cn\log(n-1) - c\log(n-1) - c(n-1) + \Theta(n)$$

$$\geq cn\log\left(\frac{n}{2}\right) - c\log(n-1) - c(n-1) + \Theta(n)$$

(as $n \geq 2$)

$$= cn\log n - (2cn + c\log(n-1) - c) + \Theta(n)$$

$$\boxed{T(n) \geq cn\log n}$$

(We can pick $c =$ small constants so that $\Theta(n)$ dominates $2cn - c\log(n-1) - c$)

$\Rightarrow$ quicksort's best case running time is

$\Omega(n \log n)$