Enlai Yii
2064210

Homework_2

Exercise 1

$$T(n) = 2T(\frac{n}{2}) + n^2$$

Assume that $T(n) \le c$ for all $n < n_0$ for some constants c and $n_0 > 0$.

(a)

Solve for $T(\frac{n}{2})$.

$T(\frac{n}{2}) = 2T(\frac{n}{4}) + (\frac{n}{2})^2$

Solve for $T(n)$.

$T(n) = 2(2T(\frac{n}{4}) + (\frac{n}{2})^2) + n^2 = 4T(\frac{n}{4}) + 2\frac{n^2}{4} + n^2 = 4T(\frac{n}{4}) + \frac{n^2}{2} + n^2$

$= 4T(\frac{n}{4}) + \frac{n^2 + 2n^2}{2} = 4T(\frac{n}{4}) + n^2(\frac{3}{2}) = 2^2T(\frac{n}{2^2}) + n^2(\frac{2^2 - 1}{2^{2-1}})$

Solve for $T(\frac{n}{4})$.

$T(\frac{n}{4}) = 2T(\frac{n}{8}) + (\frac{n}{4})^2$

Solve for T(n) again.

$T(n) = 4T(\frac{n}{4}) + \frac{n^2}{2} + n^2 = 4(2T(\frac{n}{8}) + (\frac{n}{4})^2) + \frac{n^2}{2} + n^2$

$= 8T(\frac{n}{8}) + 4\frac{n^2}{16} + \frac{n^2}{2} + n^2 = 8T(\frac{n}{8}) + \frac{n^2}{4} + \frac{n^2}{2} + n^2$

$= 8T(\frac{n}{8}) + n^2\frac{7}{4} = 2^3T(\frac{n}{2^3}) + n^2(\frac{2^3 - 1}{2^{3-1}})$

The general pattern for any k is.

$T(n) = 2^kT(\frac{n}{2^k}) + n^2\frac{2^k - 1}{2^{k-1}}$

Base Case for T(n) is $T(n) = 2^kT(1) + n^2\frac{2^k - 1}{2^{k-1}}$

Since T(1) is a constant, we can assume the $c = 2^kT(1)$ and $k = log_2(n)$.

$$T(n) \le cn^2 \text{ for all } n \ge 1$$

Thus, by unwinding the recurrence, $T(n) = O(n^2)$

(b)

We want to prove that there exists a constant c and an integer K such that $T(n) \le cn^2$ for all $n \ge k$. We want to show by induction that $T(n) = O(n^2)$.

Base Case: T(1) = 1

Inductive Step:

Inductive Hypothesis: Assume that $T(m) \le cm^2$ for all m < n. We want to show that $T(n) \le cn^2$.

$T(n) = 2T(\frac{n}{2}) + n^2$

$T(k) \le cK^2$ for K < n

$T(n) = 2T(\frac{n}{2}) + n^2$

$\le 2T(c(\frac{n}{2})^2) + n^2$

$= 2c\frac{n^2}{4} + n^2$

$= \frac{cn^2}{2} + n^2 = n^2(\frac{c}{2} + 1)$

$\frac{c}{2} + 1 \le c$

$1 \le \frac{c}{2}$

$2 \le c$

Therefore, we have shown $T(n) \le cn^2$ for $c \ge 2$.

Thus, we have proven $T(n) = O(n^2)$. ⏹

(c)

By DC Recurrence Theorem, the recurrence relation is $T(n) = aT(\frac{n}{b}) + c\,f(n)$.

The given recurrence relation is $T(n) = 2T(\frac{n}{2}) + n^2$. By DC Recurrence Theorem, a = 2, b = 2, c = 1, and $f(n) = n^2$.

If c = 1, then $T(n) = \theta(f(n)\,log_b n)$. Thus, $T(n) = \theta(n^2\,log_2 n)$.

To determine which function grows faster, we can use a = 2.

$log_b a = log_2 2 = 1$, so comparing $n^{log_b a}$ and n^2. We have $n < n^2$.

Since n^2 grows faster than n, we have $T(n) = \theta(n^2)$. By definition 2.6, if $f(n) = \theta(g(n))$, then $f(n) = O(g(n))$. Therefore, by DC Recurrence Theorem, $T(n) = O(n^2)$.

Exercise 2

Find the heaviest and lightest stone by placing two stones on two different pans. Or find the largest and smallest element from a set of n elements.

1) An algorithm that takes 2n - 3 weights.
2) A divide and conquer algorithm that takes $\frac{3n}{2} - 2$ weights when n is a power of 2, $n = 2^k$

1) The total number of comparisons is (n - 1) + (n - 2) = 2n - 3 comparisons/weights.
2) A divide and conquer algorithm that divides the group into two subgroups and finds the heaviest and lightest stone from each subgroup. Keep finding the heaviest and lightest recursively.

A = set of n stones to compare
left = left subset
right = right subset

```
function FIND2STONES(A, left, right):
    # if there is only one element
    if left = right
        return A[left]

    # if there are two elements, then find the heaviest and lightest stone
    if left + 1 = right
        if A[left] > A[right]
            heaviest = A[left]
            lightest = A[right]
        else
            heaviest = A[right]
            lightest = A[left]
        return (heaviest, lightest)

    # divide the input, and find the heaviest and lightest stone for each subset
    mid = floor( (left + right)/2 )
    (HV1, LT1) = FIND2STONES(A, left, mid)
    (HV2, LT2) = FIND2STONES(A, mid + 1, right)
```

```
     # find the heaviest stone from the subset
14   if HV1 > HV2
15         heaviest = HV1
16   else
17         heaviest = HV2

     # find the lightest stone from the subset
18   if LT1 > LT2
19         lightest = LT2
20   else
21         lightest = LT1

     # return the heaviest and lightest element
22   return (heaviest, lightest)
```

From line 11 through 13, the set is divided into 2 subsets. Then 2 comparisons are made to find the heaviest and lightest stone from the 2 subsets. So, we have the recurrence relation of $T(n) \ = \ 2T(\frac{n}{2}) \ + \ 2$, where T(n) is the number of comparisons and assume that n is a power of 2.

By induction, we want to show that $T(n) \ = \ \frac{3n}{2} \ - \ 2$ weights.

Base Case: $T(2) \ = \ 3\frac{2}{2} \ - \ 2 \ = \ 1$

Induction Step:

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + 2 \\ &= 2(3\frac{n/2}{2} - 2) + 2 \\ &= \frac{3n}{2} - 2 \text{ for } n > 2. \end{aligned}$$

Exercise 3

S = [20, 11, 3, 7, 5, 10, 2, 13]

Simple Sort

Find the first (i) and second (j) smallest elements, then swap positions. The color green means that position is already sorted.

20 11 3 7 5 10 2 13	→	11 20 3 7 5 10 2 13	(20 > 11, so swap)	C = 1
11 20 3 7 5 10 2 13	→	3 20 11 7 5 10 2 13	(11 > 3, so swap)	C = 1
3 20 11 7 5 10 2 13	→	2 20 11 7 5 10 3 13	(3 > 2, so swap)	C = 1
2 20 11 7 5 10 3 13	→	2 11 20 7 5 10 3 13	(20 > 11, so swap)	C = 1
2 11 20 7 5 10 3 13	→	2 7 20 11 5 10 3 13	(11 > 7, so swap)	C = 1
2 7 20 11 5 10 3 13	→	2 5 20 11 7 10 3 13	(7 > 5, so swap)	C = 1
2 5 20 11 7 10 3 13	→	2 3 20 11 7 10 5 13	(5 > 3, so swap)	C = 1
2 3 20 11 7 10 5 13	→	2 3 11 20 7 10 5 13	(20 > 11, so swap)	C = 1
2 3 11 20 7 10 5 13	→	2 3 7 20 11 10 5 13	(11 > 7, so swap)	C = 1
2 3 7 20 11 10 5 13	→	2 3 5 20 11 10 7 13	(7 > 5, so swap)	C = 1
2 3 5 20 11 10 7 13	→	2 3 5 11 20 10 7 13	(20 > 11, so swap)	C = 1
2 3 5 11 20 10 7 13	→	2 3 5 10 20 11 7 13	(11 > 10, so swap)	C = 1
2 3 5 10 20 11 7 13	→	2 3 5 7 20 11 10 13	(10 > 7, so swap)	C = 1
2 3 5 7 20 11 10 13	→	2 3 5 7 11 20 10 13	(20 > 11, so swap)	C = 1
2 3 5 7 11 20 10 13	→	2 3 5 7 10 20 11 13	(11 > 10, so swap)	C = 1
2 3 5 7 10 20 11 13	→	2 3 5 7 10 11 20 13	(20 > 11, so swap)	C = 1
2 3 5 7 10 11 20 13	→	2 3 5 7 10 11 13 20	(20 > 13, so swap)	C = 1

The total number of comparisons is 17.

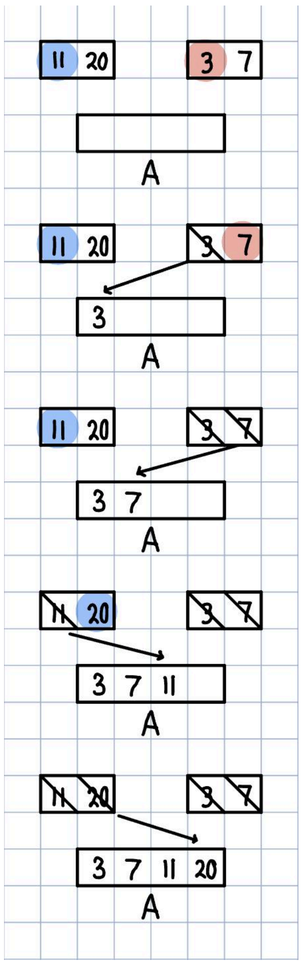
Merge Sort

Split the array into two equal halves: [20 11 3 7] [5 10 2 13]

Sort the first half:

[20 11 3 7] → [20 11] [3 7] → [20] [11] [3] [7]

[20] [11] [3] [7] → [11 20] [3 7] (11 < 20 and 3 < 7) C = 2



Compare 3 and 11, 3 < 11, so 3 is inserted to the array C = 1

Compare 7 and 11, 7 < 11, so 7 is inserted to the array C = 1

Compare 11 and 20, 11 < 20, so 11 is inserted to the array C = 1

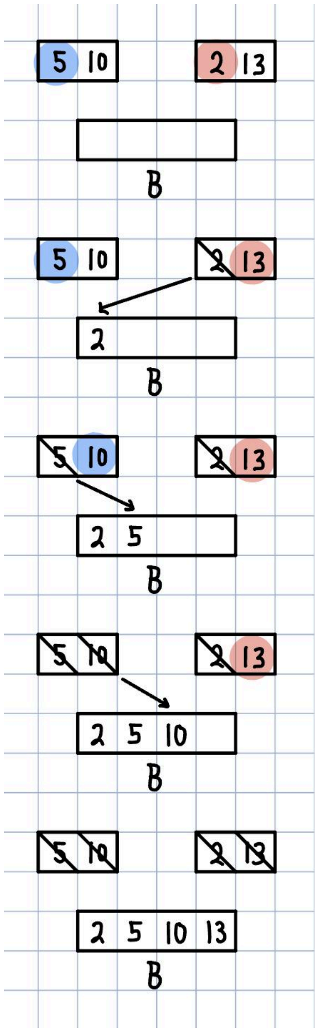
20 is inserted to the array

The first half has a total number of 5 comparisons.

Sort the second half:

[5 10 2 13] → [5 10] [2 13] → [5] [10] [2] [13]

[5] [10] [2] [13] → [5 10] [2 13] (5 < 10 and 2 < 13) C = 2



Compare 5 and 2, 2 < 5, so insert 2 into the array C = 1

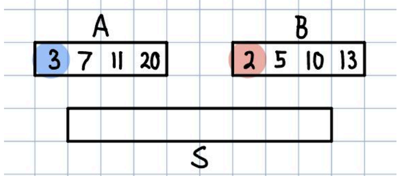
Compare 5 and 13, 5 < 13, so insert 5 into the array C = 1

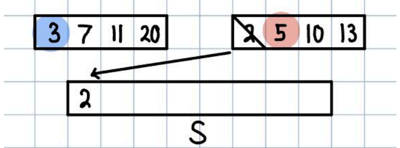
Compare 10 and 13, 10 < 13, so insert 10 into the array C = 1

Insert 13 into the array

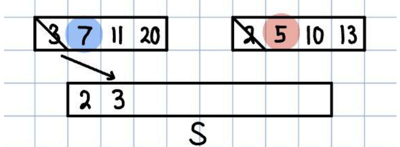
The second half has a total number of 5 comparisons.

Merge both arrays together

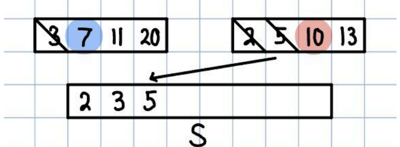




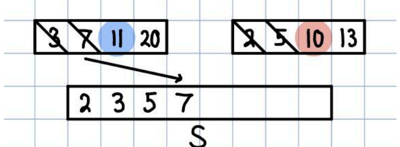
Compare 3 and 2, 2 < 3, so insert 2 into the array
C = 1



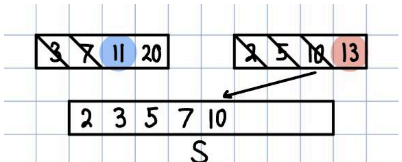
Compare 3 and 5, 3 < 5, so insert 3 into the array
C = 1



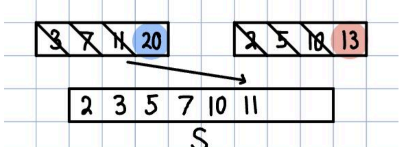
Compare 7 and 5, 5 < 7, so insert 5 into the array
C = 1



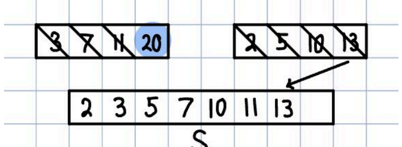
Compare 7 and 10, 7 < 10, so insert 7 into the array
C = 1



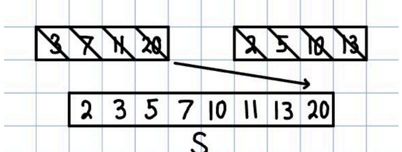
Compare 11 and 10, 10 < 11, so insert 10 into the array
C = 1



Compare 11 and 13, 11 < 13, so insert 11 into the array
C = 1



Compare 20 and 13, 13 < 20, so insert 13 into the array
C = 1



Insert 20 into the array

The total number of comparisons for merging both arrays into one sorted array is 7 comparisons.
The total number of comparisons for the whole algorithm is 5 + 5 + 7 = 17 comparisons.

Quick Sort

The 1st element is the pivot point. And the color green means the element is already sorted.

20 11 3 7 5 10 2 13	→	13 11 3 7 5 10 2 20	(20 is the greatest element in the array)	C = 7
13 11 3 7 5 10 2 20	→	2 11 3 7 5 10 13 20	(13 is greater than [11 3 7 5 10 2])	C = 6
2 11 3 7 5 10 13 20	→	2 11 3 7 5 10 13 20	(2 is less than [11 3 7 5 10])	C = 5
2 11 3 7 5 10 13 20	→	2 10 3 7 5 11 13 20	(11 is greater than [3 7 5 10])	C = 4
2 10 3 7 5 11 13 20	→	2 5 3 7 10 11 13 20	(10 is greater than [3 7 5])	C = 3
2 5 3 7 10 11 13 20	→	2 3 5 7 10 11 13 20	(5 as pivot: 3 < 5 and 5 < 7)	C = 2

The total number of comparisons is 27.

Exercise 4

Show the correctness of QuickSort by induction.

Base Case: When the size of S is 1, then the algorithm is correct for a single element.

Inductive Step:

Inductive Hypothesis: Assume that QuickSort is correct for all sets of size less than n.

We want to show that the algorithm is correct for an input of n + 1 elements.

By the induction hypothesis, the pivot point can be p = 0, 1, …, n - 1. The pivot partitions the array S into 2 parts, S1 of size p and S2 of size n - p - 1. Since S1 is not greater than p and S2 is not less than p, then by induction hypothesis, both subsets (S1 and S2) are sorted recursively with (n + 1) - 1 comparisons. After the recursive calls, we concatenate all of the arrays, S1 + [p] + S2.

Thus, by induction, QuickSort correctly sorts all of the elements in the array. ◻

Exercise 5

Given a set of n files, determine whether AT LEAST n / 2 are identical.

Given 2 files, are they the same or not?

1) A divide and conquer algorithm that takes $O(n\ log\ n)$ queries.

F = set of n files

left = left subset

right = right subset

```
countFiles = 0

function INFRINGEMENT(F, left, right, countFiles):
        # if there is one element
1       if left = right:
2             return F[left]

        # if given two files are the same, then increment count by 1
3       if F[left] = F[right]
4             return countFiles++

        # divide the array into 2 subarrays
5       mid = floor( (left + right) / 2 )
6       C1 = INFRINGEMENT(F, left, mid, countFiles)
7       C2 = INFRINGEMENT(F, mid + 1, right, countFiles)
8       countFiles = C1 + C2

        # check whether if at least n / 2 of the set are identical
        # if count is more than n / 2, then return true
9       if countFiles >= |F| / 2
10            return TRUE
11      else
12            return FALSE
```

The algorithm will recursively divide the set of n files into two subsets until size of 1 is reached (i.e. n/2, n/4, n/8, …, 1). This will take O(log n) comparisons. On line 3, the comparison will compare all of the files, which will take O(n) comparisons.

Thus, the algorithm will take O(n log n) comparisons.