CS 325 – Analysis of algorithm

Name: Shou-Cheng Wu

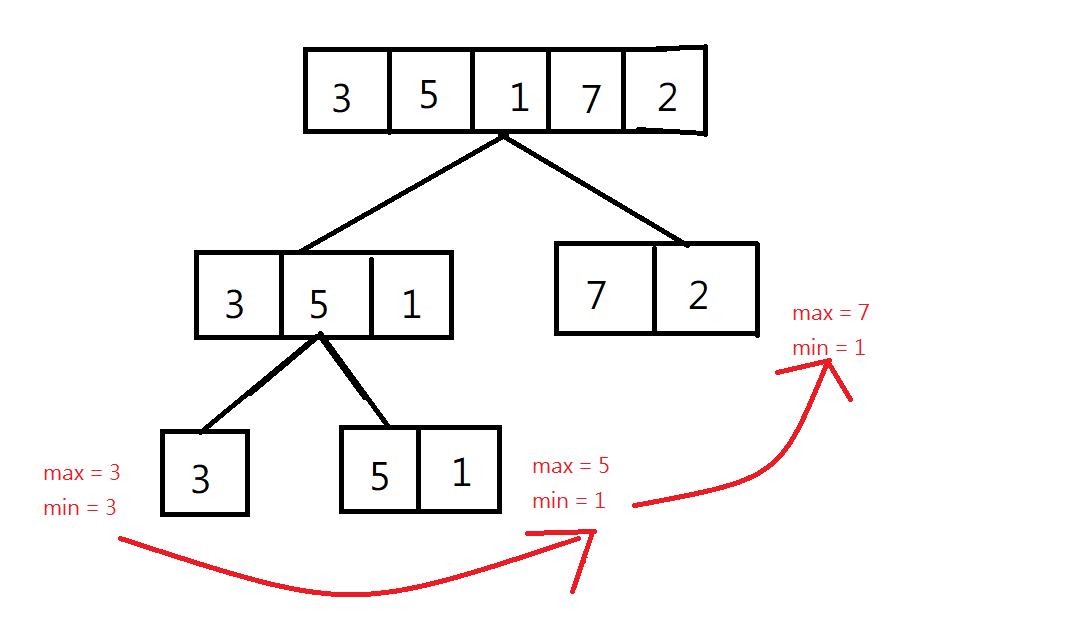
ID: 933920148

Email: wusho@oregonstate.edu

Problem 1: Design and analyze a **divide and conquer** algorithm that determines the minimum and maximum value in an unsorted list (array).

1. **Verbally describe and write pseudo-code for the min and max algorithm.**

Based on the divide and conquer algorithm, we are supposed to divide the problem into sub-problems. About this question, the first step is that we need a global maximum and a global minimum. Then, we are supposed to keep dividing arrays into smaller arrays until the size of it is either one or two. Then, we can compare and update the maximum and minimum with other subarrays.



The process of finding maximum and minimum

**pseudo-code:**

func MaxMinSolution (array, left\_index, right\_index, min, max)

if array\_size = 1 then

min = min (min, array[left\_index])

max = max (max, array[left\_index])

return min, max;

end

if array\_size ==2 then

min = min (min, array[left\_index],array[right\_index])

max = max (max, array[left\_index],array[right\_index])

return min, max

end

middle 🡨(left\_index + right\_index)/2

min, max 🡨 MaxMinSolution (array, left\_index, middle, min, max)

min, max 🡨 MaxMinSolution (array, middle+1, right\_index, min, max)

return min, max

end func

1. **Write a recurrence for the running time T(n) of your algorithm.**

T(n) = 2T(n/2)+O(1)

1. **Solve the recurrence to determine the asymptotic running time of the algorithm. How does the theoretical running time of the recursive min\_and\_max algorithm compare to that of an iterative algorithm for finding the minimum and maximum values of an array.**

**Master Method:**

= = n

f(n) = Θ () ,ε = 1

It is class 1.

So, the time complexity is Θ ( =Θ (n)

**Iterative algorithm:**

T(n) = 2T(n/2)+1

=2[2\*T(n/4) +1]+1 = 4 T(n/4) + 1 + 2

=4 [2\*T(n/8) +1] +1 +2 = 8 T(n/8) + 1 + 2 + 4

* , , , … = n
* )

**Problem 2: (5 pts) The Mergesort3 algorithm is a variation of Mergesort that instead of splitting the list into two halves, splits the list into three thirds. Mergesort3 recursively sorts each third and then merges the thirds together into a sorted list by calling a function named Merge3.**

1. **Write pseudo-code for Mergesort3 and Merge3**

func Merge3(a)

If (len(a)==2) then

a <- sorted(a)

end if

if (len(a)>=3) then

t 🡨 int (len(a)/3)

left 🡨 a[0:t]

mid 🡨 a[t:2t]

right🡨a[2t:end]

if len(left), len(mid),len(right) >= 3 then

Merge3(left), Merge3(mid), Merge3(right)

else

sort left, mid, right

end

leftindex = midindex = rightindex = totalindex = 0

While (leftindex<len(left),midindex<len(mid),rightindex<len(right))

a[totalindex] <- min(left,mid,right)

totalindex +=1

if the minimum number choose from left then

leftindex += 1

else if the minimum number choose from mid then

midindex += 1

else

rightindex +=1

end

end while

return a

end func

1. **Let T(n) denote the running time of Mergesort3 on an array of size n. Write a recurrence relation for T(n).**

T(n) = 3 T(n/3) + O(n)

1. **Solve the recurrence relation to obtain the asymptotic running time.**

f(n) = n

() = = n

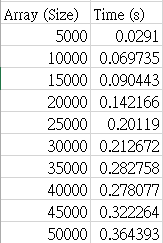
(n) =() It is case 2, so the time complexity is (n)

Problem3:

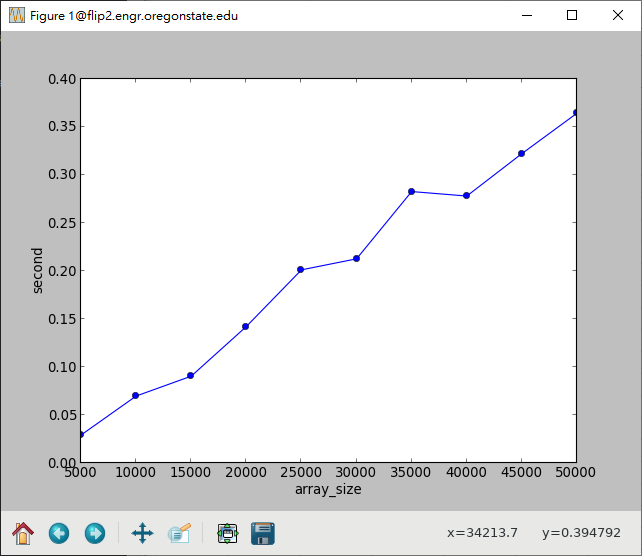
Submit it through teach system

Problem4:

1. Submit it through teach system
2. **Collect running times**

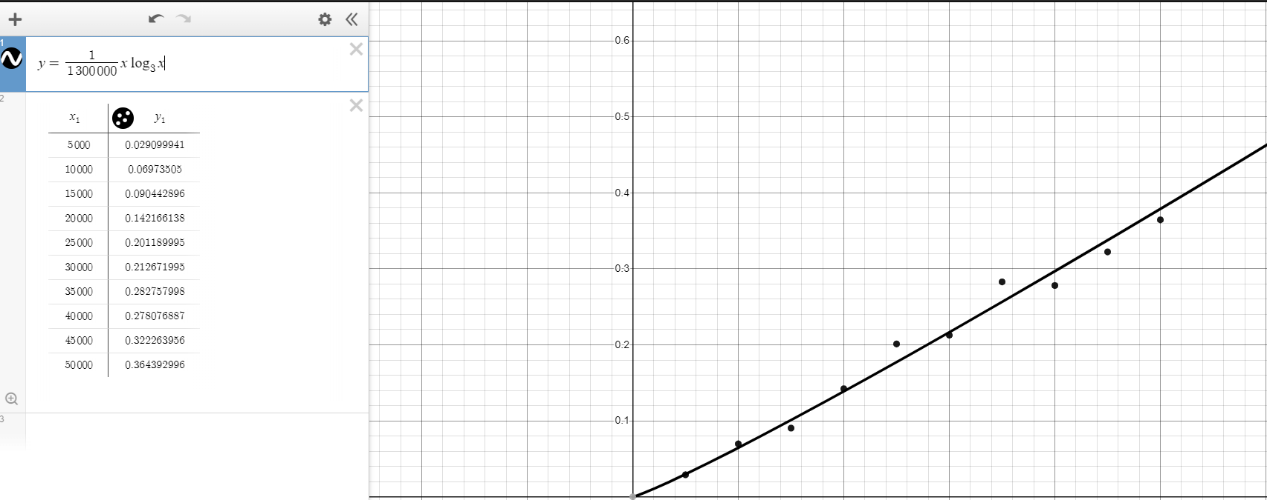


Running time of Mergersort3



Running time of Mergersort3

1. Plot data and fit a curve

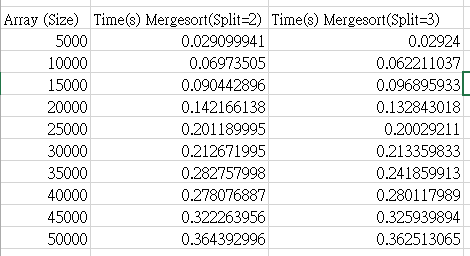


Running time of Mergersort3

y = x x

1. Compare

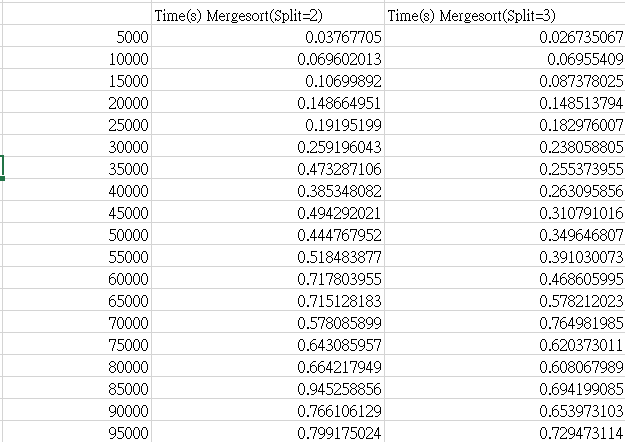
The two graphs below show that the two merge sorts’ running are pretty similar because when we do a 2-way merge sort, the time complexity is O(nn). And the time complexity of 3 – way merge sort is O(nn). These two algorithm’s growth rates are the same.



Time comparison between Mergesort(split =2) and Mergesort(split =3) n = 5000 ~ 50000

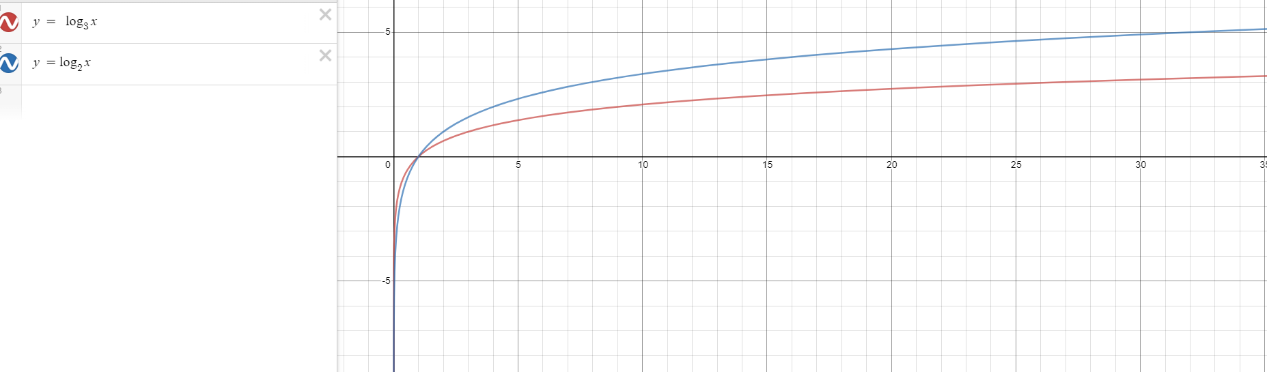
Time comparison between Mergesort(split =2) and Mergesort(split =3) n = 5000 ~ 50000

Generally, we consider that O(nn) and O(nn) are the same because we say that they are both O(nlogn). However, if n is large enough, we still can know that 3-way merge sort is a little faster than 2-way merge sort. I make the largest n be 95000 and do the experiment and plot the data below. This makes sense because n is lower than n



Time comparison between Mergesort(split =2) and Mergesort(split =3) n = 5000 ~ 95000

Time comparison between Mergesort(split =2) and Mergesort(split =3) n = 5000 ~ 95000



n vs n