Xiaoying Li
CS 325 -Winter 2020
Homework #2

- **Problem 1**

a.

This problem can be solved using the iteration method.

$T(n) = bT(n-1) + 1$
$= b[bT(n-2) + 1] + 1$
$= b^2T(n-2) + b + 1$
$= b^2[bT(n-3) + 1] + b + 1$
$= b^3T(n-3) + b^2 + b + 1$
$= b^3[bT(n-4) + 1] + b^2 + b + 1$
$= b^4T(n-4) + b^3 + b^2 + b + 1$
$\vdots$
$= b^{n-1}T(1) + b^{n-2} + b^{n-3} + b^{n-4} + \cdots + b^2 + b + 1$
$= \frac{b^{n-1} - 1}{b-1} + b^{n-1}T(1)$
$= b^n \cdot \frac{1}{b^2 - b} - \frac{1}{b-1} + b^{n-1}T(1)$

Because b is a fixed positive integer greater than 1, and T(1) is a constant, then

$$T(n) = O(b^n)$$

b.

This problem can be solved using master method.

$$T(n) = 3 \cdot T\left(\frac{n}{9}\right) + n \cdot logn$$

Then $a = 3, b = 9, \log_b a = \log_9 3 = \frac{1}{2}$.

Compare $n^{\frac{1}{2}} = \sqrt{n}$ with $f(n) = nlogn$, because $\lim_{n\to\infty} \frac{\sqrt{n}}{nlogn} = 0$, so $f(n) = O(\sqrt{n})$.

Therefore, $T(n) = O(\sqrt{n})$.

- **Problem 2**

Design of the algorithm in C++:

```
void maximumSubarray(int A1[], int sizeA1, int A2[]) {
    int maxSumA1 = 0;
    int tempSumA1 = 0;
```

```
int rightA1 = 0;

// Using Kadane's Algorithm to find the max sum of contiguous subarray of A1.
for (int i = 0; i < sizeA1; i++) {
	tempSumA1 = tempSumA1 + A1[i];
	if (maxSumA1 < tempSumA1) {
		maxSumA1 = tempSumA1;
		rightA1 = i;
	}
	if (tempSumA1 < 0) {
		tempSumA1 = 0;
	}
}

// Using the max sum and the right index of the contiguous subarray to
// calculate the left index of the subarray.
int temp = maxSumA1;
int leftA1 = rightA1;

while (temp != 0) {
	temp = temp - A1[leftA1];
	leftA1--;
}

leftA1++;

// Print the result.
cout << "The maximum subarray of A1 is [" << leftA1 << ", " << rightA1 << "]."
	<< endl;
cout << "The maximun contiguous sum is " << maxSumA1 << "." << endl;

int currentSumA2 = 0;
int maxSumA2 = 0;
int sizeA2 = sizeA1 + 1;
int rightA2 = sizeA2-1;
int leftA2 = sizeA2-1;

// Calculate the sum of elements of A2 from the right end and find the max sum.
for (int i = 1; i <= sizeA2; i++) {
	currentSumA2 = currentSumA2 + A2[sizeA2-i];
	if (currentSumA2 > maxSumA2) {
		maxSumA2 = currentSumA2;
		leftA2 = sizeA2 - i;
	}
```

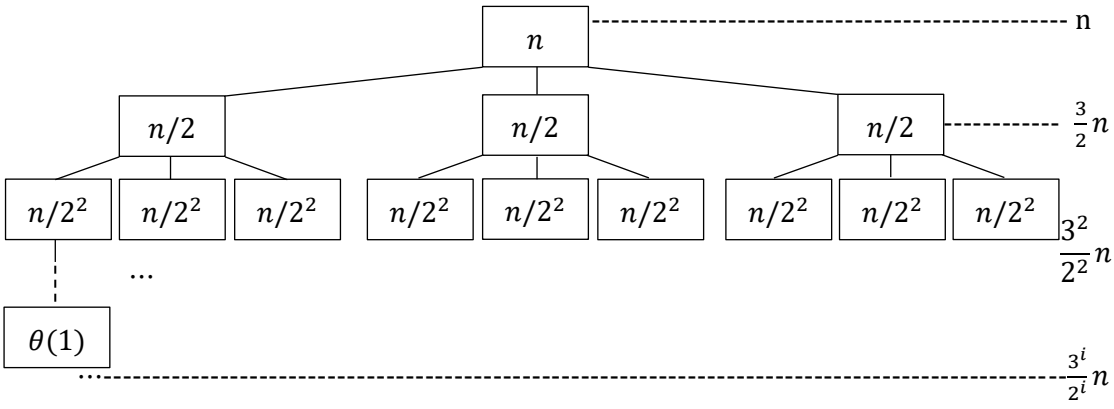
```
    }

    // Compare the max sum of A2 with the maximum sum of A1.
    // If the max sum of A2 is greater than the max sum of A1,
    // then the max sum of contiguous subarray of A2 is the max sum of A2.
    if (maxSumA1 < maxSumA2) {
        cout << "The maximum subarray of A2 is [" << leftA2 << ", " << rightA2
            << "]." << endl;
        cout << "The maximun contiguous sum is " << maxSumA2 << "." << endl;
    }

    // Else, the max sum of contiguous subarray of A2 is the maximum sum of A1.
    else {
        cout << "The maximum subarray of A2 is [" << leftA1 << ", " << rightA1
            << "]." << endl;
        cout << "The maximun contiguous sum is " << maxSumA1 << "." << endl;
    }
}
```

Prove it works in linear time:

In my function above, except for the 3 loops, all the other parts can be run in constant time. For the first "for (int i = 0; i < sizeA1; i++)" loop, it runs sizeA1 times, so its bigO is O(n). For the second "while (temp != 0)" loop, it runs at most sizeA1 times, so its bigO is O(n). And for the last "for (int i = 1; i <= sizeA2; i++)" loop, it runs sizeA2 times, and A2=A1+1, so its bigO is also O(n). Then the bigO for the whole function is O(1)+O(n)+O(n)+O(n)=O(n), which means the function works in linear time complexity.

- **Problem 3**

a.



$$T(n) = n + \frac{3}{2}n + \frac{3^2}{2^2}n + \cdots + \frac{3^i}{2^i}n$$

$= n + \frac{3}{2}n + \frac{3^2}{2^2}n + \cdots + \theta\left(n^{\log_2 3}\right)$

Therefore, $T(n) = O(n^{\log_2 3})$.

b.

In order to prove $T(n) = O(n^{\log_2 3})$, we must prove that $T(n) \le cn^{\log_2 3}$ for some positive constant c.

To prove it using mathematical induction, we must show that if $T(n) \le cn^{\log_2 3}$ is true, then $T(n+1) \le c((n+1)^{\log_2 3})$ is true, c is some positive constant number.

Firstly, we must show that $T(1) \le cn^{\log_2 3} = c$ is true. Because c can be any positive constant number, so let $c \ge T(1)$, $T(1) \le c$ is true.

Suppose $T(n) \le cn^{\log_2 3}$ is true for all positive numbers n, c is some positive constant number.

$$T(n+1) = 3 \cdot T\left(\frac{n+1}{2}\right) + n + 1$$

Because $n \ge 1$, so $\frac{n+1}{2} \le n$. According to our supposition, $T\left(\frac{n+1}{2}\right) \le c \cdot (\frac{n+1}{2})^{\log_2 3}$. So,

$$T(n+1) \le 3c \cdot (\frac{n+1}{2})^{\log_2 3} + n + 1$$

$$T(n+1) \le 3c \cdot (\frac{1}{2})^{\log_2 3}(n+1)^{\log_2 3} + n + 1$$

$$T(n+1) \le c(n+1)^{\log_2 3} + n + 1 = O((n+1)^{\log_2 3})$$

Thus, $T(n) = O(n^{\log_2 3})$ is true for some positive constant number c is proved.

- **Problem 4**

a.

The badSort algorithm is a recursive algorithm, so in order to sort the input array and not run into infinite loop, it must hit the base case. And one of the conditions to hit the base case (aka the first if statement) is n=2, or the array is A[0,1].

When n>2, the algorithm uses the divide and conquer approach to recursively call the badSort itself. So at least one of A[0…m-1], A[n-m…n-1] and A[0…m-1] equal to A[0, 1] at least once.

If A[0…m-1]=A[0, 1], then m-1=1, then m=2;

If A[n-m…n-1] =A[0, 1], then n-m=0 and n-1=1, then m=n=2, but n>2, so m=n=2 can never be true;

If A[0…m-1]=A[0, 1], then m-1=1, then m=2.

Thus, in order to hit the base case, m=2.

And $m = \lceil \alpha \cdot n \rceil$, if m=2, then $\lceil \alpha \cdot n \rceil = 2$, then $1 < \alpha n \le 2 \rightarrow \frac{1}{n} < \alpha \le \frac{2}{n}$. Because $n > 2 \rightarrow \frac{1}{n} < \frac{1}{2}$, so $\alpha > \frac{1}{2}$. So, in order to hit the base case, $\alpha > \frac{1}{2}$.

Therefore, if $\alpha \le \frac{1}{2}$, the divide and conquer approach of badSort would fail to sort the

input array.

b. No.

According to problem a, if the badSort work correctly, when n>2, $\frac{1}{2} < \alpha \leq \frac{2}{n}$. If $\alpha = \frac{3}{4}$, then $\frac{3}{4} \leq \frac{2}{n} \rightarrow n \leq \frac{8}{3}$, then $2 < n \leq 2.666\cdots$. But n is an integer, obviously there is no integer between 2 and 2.666….

In order to fix it, because $\alpha \leq \frac{2}{n} \rightarrow n \leq \frac{2}{\alpha}$, there must be at least one integer solution for $2 < n \leq \frac{2}{\alpha}$, which means $\frac{2}{\alpha} \geq 3$, so $\alpha \leq \frac{2}{3}$. Therefore, when $\alpha = \frac{3}{4}$, the badSort cannot work correctly. And in order to fix it, we can let $\frac{1}{2} < \alpha \leq \frac{2}{3}$.

But if we do want the badSort to work correctly when $\alpha = \frac{3}{4}$, we can fix the algorithm by changing $m = \lceil \alpha \cdot n \rceil$ to $m = round(\alpha \cdot n)$. Because according to problem a, in order to hit the base case, m=2; and when $m = round(\alpha \cdot n) = 2$, $\frac{3}{2} \leq \alpha n < \frac{5}{2}$. If $\alpha = \frac{3}{4}$, then $\frac{3}{2} \leq \frac{3}{4}n < \frac{5}{2} \rightarrow 2 \leq n < \frac{10}{3}$, and n>2, n is an integer, so n has an solution n=3, which means the badSort can work correctly under this condition.

Therefore, in order to fix it, we can let $\frac{1}{2} < \alpha \leq \frac{2}{3}$ or change $m = \lceil \alpha \cdot n \rceil$ to $m = round(\alpha \cdot n)$.

c. Recurrence:

$$T(n) = 3T(\alpha n) + O(1)$$

d.

When $\alpha = \frac{2}{3}$, the recurrence from problem c can be written as:

$$T(n) = 3T(\frac{2}{3}n) + O(1)$$

Use master method to solve the recurrence from problem c, $a = 3, b = \frac{3}{2}, \log_b a = \log_{\frac{3}{2}} 3$.

Compare $n^{\log_{\frac{3}{2}} 3} \approx n^{2.7}$ with $f(n) = O(1)$, $f(n) = O(n^{\log_{\frac{3}{2}} 3})$.

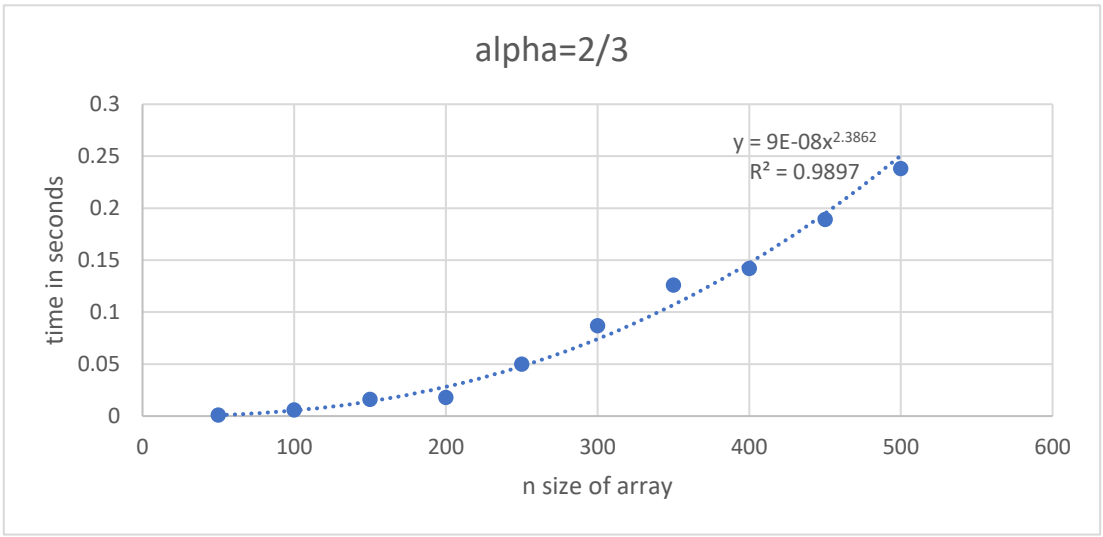
Therefore, $T(n) = O(n^{\log_{\frac{3}{2}} 3})$. When $\alpha = \frac{2}{3}$, the asymptotic time complexity of badSort is $O(n^{\log_{\frac{3}{2}} 3})$.

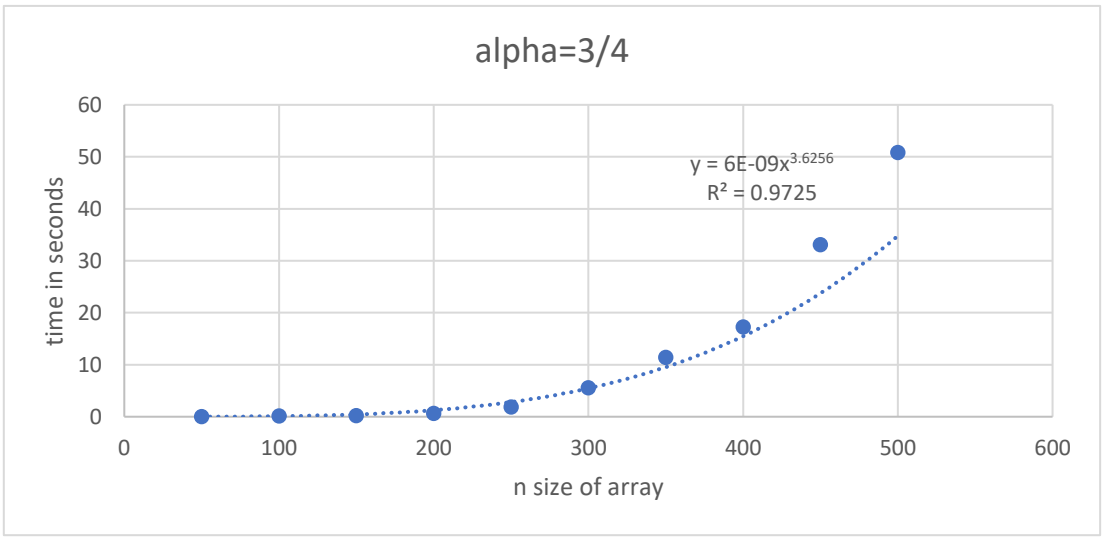
- **Problem 4**

b.

n size of array	time in seconds	
	alpha=2/3	alpha=3/4
50	0.001	0.014
100	0.006	0.132
150	0.016	0.216
200	0.018	0.632
250	0.05	1.878
300	0.087	5.584
350	0.126	11.42
400	0.142	17.278
450	0.189	33.054
500	0.238	50.83

c.





The experimental running times of badSort when alpha=2/3 and 3/4 can be described by the equations we concluded above. The theoretical running times can be described by their Big O.

The experimental running time equation is $y = 9 \cdot 10^{-8} x^{2.3862} = O(x^{2.4})$ for alpha=2/3 and $y = 6 \cdot 10^{-9} x^{3.6256} = O(x^{3.7})$ for alpha=3/4. And the theoretical Big O for alpha=2/3 is $O\left(n^{\log_{\frac{3}{2}} 3}\right) \approx O(n^{2.7})$, for alpha=3/4 is $O\left(n^{\log_{\frac{4}{3}} 3}\right) \approx O(n^{3.8})$. So, our experimental result is very close to the theoretical running times.

d.

Alpha=2/3 provides better performance.

Because for every same n value, alpha=3/4 uses more time than alpha=2/3. And the big O for alpha=3/4 is larger than alpha=2/3.