Tristan Santiago

CS325_400

Homework Assignment #2

October 2, 2018

Problem 1: Give the asymptotic bounds for T(n) in each of the following recurrences. Make your bounds as tight as possible and justify your answers.

A. T(n) = 3T(n − 1) + 1

Given: $T(n) = 3T(n-1) + 1$

In the equation above, if we replace n with n – 1, we get:

$T(n-1) = 3T((n-1) -1)+1$ (by replacing n with n-1). If we simplify, it becomes:

$T(n-1)=3T(n-2)+1$

Now, if we repeat the process and substitute n with n-2 in the original equation:

$T(n-2) = 3T(n-2)-1)+1$ (by replacing n with n-2). If we simplify, it becomes:

$T(n-2) = 3T(n-3)+1$

Now, if we use this new information to continue substituting:

$T(n) = 3\underline{T(n-1)}+1$ (here we substitute T(n-1) for the definition of T(n-1))

$= 3\underline{(3T(n-2)+1)}+1$ (by substituting the definition of T(n-1) for T(n-1))

Now, we combine like terms (and we don't simplify, so we can see the pattern):

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

Again, we repeat using the equation above. We substitute T(n-2) for it's definition:

$=3^2\underline{(3T(n-3)+1)}+3+1$ (by substitution). Now we combine the like terms:

$=3^3T(n-3)+3^2+3+1$

We can now see that a pattern is starting to develop, so we make an assumption of how long this will continue until we reach the base case. We will assume that after k steps:

$T(n) = 3^k * T(n-k)+3^{k-1}+3^{K-2}+...3+1$

Until k = n – 1, then substitute:

$T(n) = 3^{n-1} * T(1) + 3^{k-1}+3^{K-2}+...3+1 .. + 3^{n-2}$

$= 3 + 1 + ... 3^{n-1} = \sum_{i=0}^{n-1} 3^i$

$\therefore \Theta(3^n)$

B. $T(n) = 2T\left(\frac{n}{4}\right) + nlgn$

The master method applies to recurrences of the form:

$T(n) = a\ T(n/b) + f\ (n)$

Where $a \geq 1$, $b > 1$, and f is asymptotically positive.

By the master method, the given equation tells us:

1. Extract a, b, and f(n) from the given recurrence.
   $a = 2, b = 4, f(n) = n \log n$
2. Determine $n^{logb(a)}$.
   $n\ log^b a = n^{log4} 2 = n^{0.5}$
3. Compare f(n) and $n^{logb(a)}$ asymptotically.
   $F(n) = n \log(n)$ and $n^{\log b(a)} = n^{0.5}$, where $\log4(2) < 1$
   $F(n)\ n \log\ (n) > \log4(2)$
4. Determine the appropriate Master Method case and apply it.
   Thus, case 3, but we have to check the regularity condition.
   The following should be true:

   $af(\frac{n}{b}) \leq cf(n)$, where $c < 1$

   $2(\frac{n}{4}) \log (\frac{n}{4}) \leq c\ n \log n$

   This is true for when $c = \frac{1}{2}$

   So because $f(n) = \Omega(n^{logb(a)+\epsilon}) = \Omega(n^{log4(2)+\epsilon})$ where $\epsilon = 0.5$,

   $T(n) = \Theta(f(n)) = \Theta\ (n \log n)$

   $\therefore T(n) = \Theta\ (n \log n)$

Problem 2

The ternary search algorithm is a modification of the binary search algorithm that splits the input not into two sets of almost-equal sizes, but into three sets of sizes approximately one-third.

a. **Verbally describe and write pseudo-code for the ternary search algorithm.**

Similar to the binary search, the ternary search requires a sorted list (array) in order to function properly. The ternary search follows a divide and conquer strategy in order to locate its target value. However, unlike the binary search algorithm, it divides the array into three parts, instead of two. In order to divide the list into three parts, two middle values are calculated. During its execution, the following rules are followed: (let us call value to be located 'x')

- If the target value is equal to the first middle value, then that means the value has been located in the array and the first middle value is returned.
- If the target value is equal to the second middle value, then that means the value has been located in the array and the second middle value is returned.
- If the target value is less than the first middle value, then the first third of the array is searched.
- If the target value is greater than the second middle value, then the last third of the array is searched.
- Otherwise, the second third of the array is searched.

It is important to note that the second third (2/3) of the array is ignored with each iteration.

Pseudo Code

```
int ternarySearch(array, l, h, target)

    if (l <= h)

        middle1 = l + (h – l)/3;

        middle2 = h – (h – l)/3;

    if (target == array[middle1])

        return middle1;

    if (target == array[middle2]

        return middle2;

    /* If the target value is less than middle1, search the first third: */

    if (target < array[middle1])

        return ternarySearch(array, l, middle1-1, target)
```

```
        /* If the target is greater than middle2, search the last third: */
        if (target > array[middle2])
            return ternarySearch(array, middle2+1, h, target)
        /* Else, search the second third: */
        else
            return ternarySearch(array, middle1+1, middle2-1, target)
    /* If the element is not found in the array: */
    return -1;
```

b. Give the recurrence for the ternary search algorithm.

By applying the Master Method

$T(n) = aT(n/b) + f(n)$, with $a \geq 1$ and $b > 1$

Where

$T(n)$ = the total time for an algorithm on an input size of (n)

n = the size of the current problem

a = the number of sub-problems in the recursion

b = the factor by which the sub-problems size is reduced in each recursive call

n/b = the size of each sub-problem

$f(n)$ = the cost for the work that has to be done outside the recursive calls (cost of dividing/merging)

1. Extract a, b, and f(n) from the given recurrence.

   $T(n) = 1T(n/3) + f(n)$

   $a = 1, b = 3, f(n) = c$

2. Determine $n^{\log b(a)}$.

   $n^{\log b(a)} = n^{\log 3(1)} = n^0 = 1$

3. Compare $f(n)$ and $n^{\log b(a)}$ asymptotically.

$f(n) = c$

$n^0 = 1$

4. Determine the appropriate Master Method case and apply it.

   So because $f(n) = \Theta(n)$, then this is case 2, and

   $T(n) = \Theta(n^{\log_b(a)}\log(n))$

   $T(n) = \Theta(n^0\log(n))$

   $T(n) = \Theta(1\log(n))$ ($n^0 = 1$), so

   $T(n) = \Theta(\log_3(n))$

c. Solve the recurrence to determine the asymptotic running time of the algorithm. How does the running time of the ternary algorithm compare to that of the binary search algorithm?

I used this explanation to help me understand the answer to this question:

https://cs.stackexchange.com/questions/29755/why-is-binary-search-faster-than-ternary-search

Binary search has $\log_2(n) + O(1)$ many comparisons, while on the other hand ternary search has $2 \cdot \log_3(n) + O(1)$ because in each step, you must perform 2 comparisons in order to cut the list into three parts.

$$2 \cdot \log_3(n) + O(1) = 2 \cdot \frac{\log(2)}{\log(3)} \log_2(n) + O(1)$$

So since $2 \cdot \frac{\log(2)}{\log(3)} = 1.261 > 1$ we actually get more comparisons with ternary search.

Another explanation that makes sense to me:

https://stackoverflow.com/questions/32572355/binary-search-vs-ternary-search

4

Both have constant space, but big O time for Ternary Search is Log_3 N instead of Binary Search's Log_2 N **which both come out to log(N) since log_b(N) = log_x(N)/log_x(b)**.

In practice Ternary Search isn't used because you have to do an extra comparison at each step, which in the general case leads to more comparisons overall. 2 * Log_3(N) comparisons vs Log_2(N) comparisons.

share improve this answer

edited Sep 14 '15 at 19:57

answered Sep 14 '15 at 19:39

Louis Ricci

16.4k ● 4 ● 34 ● 57

Problem 3

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

a. Verbally describe and write pseudo code for the min_and_max algorithm.

Pseudo code

minMax(int array[], min, max)

/* If the array only has one element: */

if (end = start)

/* That element is both the min and max value. */

min = array[start];

max = array[start];

/* Else if the array has two elements: */

/* One of the elements is bigger than the other or they are both equal: */

else if end – start = 1

if array[start] ≤ array[end]

min = array[end];

max = array[end];

else min = array[end];

max = array[start];

/* Else if the array has more than two elements: */

else if end – start > 1

/* Divide the array recursively and determine the min and max in each part: */

minMax(array[start…⌊$(start + end/ 2)$⌋], min, max)

minMax(array[⌊$(start + end)/2$⌋ + 1 … $end, min2, max2$)

/* If the min from the second list is smaller than the min from the first: */

if (min2 < min)

/* Make it the new minimum: */

min = min2;

/* If the max from the second list is greater than the max from the first: */

if (max2 > max)

/* Make it the new maximum: */

max = max2;

b. Give the recurrence.

T(n) = aT(n/b) + f(n), where a ≥ 1 and b > 1

n = the size of the current problem

a = the number of sub-problems in the recursion

b = the factor by which the sub-problem's size is reduced in each recursive call

n/b = the size of each sub-problem

f(n) = the cost for the work that has to be done outside the recursive calls (cost of dividing/merging)

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

c. 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.

By applying the Master Method

$T(n) = aT(n/b) + f(n)$, with $a \geq 1$ and $b > 1$

Where

$T(n)$ = the total time for an algorithm on an input size of (n)

n = the size of the current problem

a = the number of sub-problems in the recursion

b = the factor by which the sub-problems size is reduced in each recursive call

n/b = the size of each sub-problem

f(n) = the cost for the work that has to be done outside the recursive calls (cost of dividing/merging)

1. Extract a, b, and f(n) from the given recurrence.

   $T(n) = 2T(n/2) + c$

   $a = 2, b = 2, f(n) = c$

2. Determine $n^{logb(a)}$.

   $n^{logb(a)} = n^{log2(2)} = n^1$

3. Compare f(n) and $n^{logb(a)}$ asymptotically.

   $f(n) = c$

   $n^{logb(a)} = n^{log2(2)} = n^1 = n$

   $F(n) = n$

4. Determine the appropriate Master Method case and apply it.

   Thus case 2, evenly distributed because:
   $F(n) = \Theta(n)$,
   $T(n) = \Theta(n^{logb(a)}\log(n))$
   $T(n) = \Theta(n^1\log(n))$
   $T(n) = \Theta(n \log(n))$

   The iterative method finds the minimum and maximum elements in $O(n)$ time. Compared to the iterative implementation, in the divide and

conquer implementation, the number of comparisons is less, because the min/max are found in O(n log(n)) time.

Problem 4

Consider the following pseudocode for a sorting algorithm.

```
StoogeSort(A[0 ... n – 1])
if n = 2 and A[0] > A[1]        // Step 1
        swap A[0] and A[1]      // Constant time
else if n > 2
        m = ceiling(2n/3)       // Constant time
        StoogeSort(A[0...m – 1])        // Step 2: Recursive call
        StoogeSort(A[n – m ... n – 1])          // Recursive call
        StoogeSort(A[0... m – 1])       // Recursive call
```

A. State a recurrence for the number of comparisons executed by STOOGESORT

T(n) = the total time for an algorithm on an input size of (n)

n = the size of the current problem

a = the number of sub-problems in the recursion

b = the factor by which the sub-problems size is reduced in each recursive call

n/b = the size of each sub-problem

f(n) = the cost for the work that has to be done outside the recursive calls (cost of dividing/merging)

$T(n) = aT(n/b) + f(n)$, with $a \geq 1$ and $b > 1$

$a = 3, b = 3/2, f(n) = c$

$T(n) = 3T(3n/2) + c$

B. Solve the recurrence to determine the asymptotic running time.

With the recurrence of $T(n) = 3T(3n/2) + c$

By applying the Master Method

1. Extract a, b, and f(n) from the given recurrence.

$a = 3, b = 3/2, n = 2n, f(n) = c$

2. Determine $n^{logb(a)}$.
   $n^{logb(a)} = n^{\log 3/2(3)} = n^{2.71}$
3. Compare f(n) and $n^{logb(a)}$asymptotically.
   $f(n) = c$
   $n^{logb(a)} = n^{\log 3/2(3)} = n^{2.71}$
   $n > 1$
   $f(n) < n^{logb(a)}$
   Thus, case 1.
4. Determine the appropriate Master Method case and apply it.
   $f(n) < n^{logb(a)}$
   Thus case 1, because:
   $f(n) = O(n^{logb(a)-\epsilon})$m
   $T(n) = \Theta(n^{logb(a)})$, for an $\epsilon > 0$
   $T(n) = \Theta(n^{2.71})$

Problem 5B

Table of collected data can be found in accompanying Excel Spreadsheet (exported to a PDF).

Include a "text" copy of the modified code in the written HW submitted in Canvas.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <vector>
#include <algorithm>
#include <chrono>
using namespace std;
using namespace std::chrono;

/* Function prototypes */
int stoogeSort(int array[], int start, int end);
void fillArray(int array[], int n, const int MAX_VALUE, const int MIN_VALUE);
void printArray(int array[], int n);

int main() {
```

```
	// Constants
	const int MIN_VALUE = 0;	// Minimum value
	const int MAX_VALUE = 10000;  // Maximum value
	int array[10];
	int n = sizeof(array) / sizeof(array[0]); // Size of array.
                                                                        //
Get the system time.
	unsigned seed = time(0);


	// Seed the random number generator.
	srand(seed);
	cout << "Number of elements generated: " << n << endl;	// Number
of elements in array.
	fillArray(array, n, MAX_VALUE, MIN_VALUE);


	// https://www.geeksforgeeks.org/measure-execution-time-function-cpp/
	// Get starting timepoint
	auto start = high_resolution_clock::now();


	// Call Merge Sort procedure.
	stoogeSort(array, 0, n - 1);


	// Print the sorted array.
	printArray(array, n - 1);


	// Get ending timepoint
	auto stop = high_resolution_clock::now();
	auto duration = duration_cast<microseconds>(stop - start);
	cout << "Time taken by function: " << duration.count() << "
microseconds" << endl;
	return 0;
}
```

```
int stoogeSort(int array[], int start, int end) {
	if (start >= end) {
		return 0;
	}
	int n = end - start + 1;
	if (n == 2 && array[start] > array[end]) {
		int temp = array[end];
		array[end] = array[start];
		array[start] = temp;
	}
	else if (n > 2) {
		//int m = floor(n / 3);
		//int m = ceil(n / 3);
		int m = (end - start + 1) / 3;
		stoogeSort(array, start, end - m);
		stoogeSort(array, start + m, end);
		stoogeSort(array, start, end - m);
	}
	return 0;
}

void printArray(int array[], int n) {
	for (int i = 0; i < n; i++) {
		cout << array[i] << " ";
	}
	cout << endl;
}

void fillArray(int array[], int n, const int MAX_VALUE, const int MIN_VALUE)
{
	for (int i = 0; i < n; i++) {
		array[i] = (rand() % (MAX_VALUE - MIN_VALUE + 1)) + MIN_VALUE;
```

```
        }
}
```

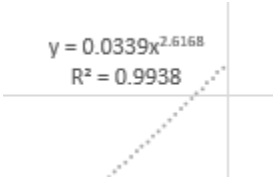
Problem 5C

Data plotted in accompanying Excel Spreadsheet (exported to a PDF).

Problem 5D

What type of curve best fits the StoogeSort data set? Give the equation of the curve that best "fits" the data and draw that curve on the graphs of created in part c). Compare your experimental running time to the theoretical running time of the algorithm?

It appears that the curve that best fits my collected data for StoogeSort is the Power curve, which appears as follows:



The other curves gave me much lower values for R^2, whereas the Power curve gave me an R^2 of 0.9938 which is really close to a perfect fit. The equation of the curve is $y = 0.0339x^{2.6168}$, which I think can also be expressed as $n^{2.6168}$. The theoretical running time I obtained from my earlier analysis of StoogeSort was $T(n) = \Theta(n^{2.71})$, and when compared to my experimental running time it isn't an exact match, but it is very, very close. Therefore, I would conclude that my implementation of StoogeSort is pretty much running as expected.