Christopher Merrill

CS325

Homework 2

**Problem 1**

**a)** **Recurrence:** T(n) = 2T(n - 2) + 1

**Muster Method:** T(n) = aT(n – b) + f(n)

a = 2, b = 2, f(n) = 1 so d = 0 because f(n) = ϴ(n0)

Therefore

T(n) = ϴ(ndan/b) = ϴ(n02n/2) = ϴ(2n/2)

T(n) = ϴ(2n/2)

**b)** **Recurrence:** T(n) = T(n - 1) + n3

**Muster Method:** T(n) = aT(n – b) + f(n)

a = 1, b = 1, f(n) = n3 so d = 3

Therefore

T(n) = ϴ(nd+1) = ϴ(n3+1) = ϴ(n4)

T(n) = ϴ(n4)

**c)** **Recurrence:** T(n) = 2T(n/6) + 2n2

**Master Method:** T(n) = aT(n/b) + f(n)

a = 2, b = 6, f(n) = 2n2

Using the limit method where f(n) = 2n2 and g(n) = n0.4:

Therefore

f(n) = Ω(n0.4 - ε)

**Case 3:**

**Regularity:** af(n/b) = 2(2(n/6)2) = 4n2/36 = n2/9

n2/9 ≤ cf(n) = c2n2 for c = 1/18

Therefore

T(n) = ϴ(n2)

**Problem 2**

**a)** The quaternary search algorithm takes a sorted array as input and then searches quadrants of the array for the target value. To accomplish this, it needs to know the starting index, the index at roughly one-quarter n, the index of the midpoint, the index at roughly three-quarters n, and the final index. It then compares each of these indices to the target value to determine which quadrant to sort next. Once the correct quadrant is found, it repeats the process until the target is found.

**Pseudocode:**

(note: “start” and “end” refer to the first and last indices to be searched)

QuaternarySearch(A[0…n-1], start, end, target)

n = start – end + 1

quarter = n/4

oneQuarterPoint = A[quarter]

midpoint = A[2\*quarter]

threeQuarterPoint = A[3\*quarter]

if n < 2 and A[start] != target

return false

else if A[start] = target

return true

else if oneQuarterPoint = target

return true

else if target < oneQuarterPoint

return QuaternarySearch(A, start, oneQuarterPoint-1, target)

else

if midpoint = target

return true

else if target < midpoint

return QuaternarySearch(A, oneQuarterPoint+1, midpoint-1, target)

else

if threeQuarterPoint = target

return true

else if target < threeQuarterPoint

return QuaternarySearch(A, midpoint+1, threeQuarterPoint -1, target)

else

if A[end] = target

return true

else

return QuaternarySearch(A, threeQuarterPoint+1, end, target)

return false

**b)** The recurrence of binary search is T(n) = T(n/2) + 1 which accounts for one comparison and a recursive call on the correct sub array. Therefore a quaternary, which has 3 comparisons and recursive calls on the corresponding quadrant, should have a recurrence of T(n) = T(n/4) + 3

**c) Recurrence:** T(n) = T(n/4) + 3

**Master Method:** T(n) = aT(n/b) + f(n)

a = 1, b = 4, f(n) = 3

Using the limit method where f(n) = 3and g(n) = 1:

Therefore

f(n) = ϴ(g(n)) = ϴ(1)

**Case 2:**

The runtime of the quaternary search is therefore equal to that of binary search, so there is no added efficiency by breaking an array into smaller chunks.

**Problem 3**

**a)** For a divide-and-conquer version of a min/max function, the idea would be to break the array into smaller and smaller halves. At each recursive call, the function would compare the max/min values of the subarray to the current max/min until the true maximum and minimum are found.

**Pseudocode:**

(note: “start” and “end” refer to the first and last indices to be searched)

MaxMin(A[0…n-1], start, end, max, min)

if start = end

max = min = A[start]

return (max, min)

else if start = end – 1

if A[start] < A[end]

max = A[end]

min = A[start]

else

max = A[start]

min = A[end]

else

midpoint = (start + end)/2

MaxMin (A, start, midpoint, max1, min1)

MaxMin(A, midpoint + 1, end, max2, min2)

if (max1 < max2)

max = max2

else

max = max1

if (min1 > min2)

min = min2

else

min = min1

return (max, min)

**b)** This algorithm performs two comparisons and has two recursive calls to half of the array, so the recurrence should be:

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

**c)** The asymptotic running time can be found using the master method

**Recurrence:** T(n) = 2T(n/2) + 2

**Master Method:** T(n) = aT(n/b) + f(n)

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

Using the limit method where f(n) = 2and g(n) = n:

Therefore

f(n) = O(n)

**Case 1:**

T(n) = ϴ() = ϴ(n)

The iterative method also has a run time of ϴ(n), so either method is equally efficient. However this method should perform less comparisons than the iterative approach.

**Problem 4**

**a)** StoogeSort works as followed: If there are 2 values in the array/subarray, and the first is larger than the second, swap them. If there are more than two elements, StoogeSort the initial 2/3 of the list, StoogeSort the final 2/3 of the list, then StoogeSort the initial 2/3 of the list again.

**b)** No, StoogeSort would not sort correctly if we replaced k = ceiling(2n/3) with k = floor(2n/3). Consider the case when n = 4:

k = floor((2\*4)/3) = floor(8/3) = 2

and

k = ceiling((2\*4)/3) = ceiling(8/3) = 3

Therefore we would miss an index if we used floor rather than ceiling, which would incorrectly set the start/end points of the next recursive call of StoogeSort.

**c)** The StoogeSort algorithm consists of 3 recursive calls to 2n/3 of the original array, and one comparison. Therefore the recurrence should be:

T(n) = 3T(2n/3) + 1

**d)** The asymptotic running time can be calculated using the Master Method

**Recurrence:** T(n) = 3T(2n/3) + 1 = 3T(n/(3/2)) + 1

**Master Method:** T(n) = aT(n/b) + f(n)

a = 1, b = 3/2 = 1.5, f(n) = 1

Using the limit method where f(n) = 1and g(n) = n2.7:

Therefore

f(n) = O(g(n)) = O(n2.7 - ε)

**Case 1:**

**Problem 5**

**a)** StoogeSort code submitted to TEACH

**b)** The code used to collect runtime data is included below. The following n-values were used: 100, 200, 400, 800, 1600, 3200, 6400.

void stoogeSort(vector<int>& vals, int start, int end) {

int temp;

int n = end - start + 1; //the total number of ints in the vector

int m = ceil(2\*n)/3; //m is equal to the ceiling of (2/3) \* n

if ((n == 2) && (vals[start] > vals[end])) {

//Swap vals[start] and vals[end]

temp = vals[start];

vals[start] = vals[end];

vals[end] = temp;

}

else if (n > 2){

temp = n/3; // Set temp eqaul to 1/3 of n

//Recursive Step

stoogeSort(vals, start, (end - temp)); //Sort the first 2/3 of the vector

stoogeSort(vals, (start+temp), end); //Sort the final 2/3 of the vector

stoogeSort(vals, start, (end - temp)); //Sort the first 2/3 of the vector again

}

}

int main()

{

vector<int> values; //A vector of vectors. The inner vectors are the lines in data.txt

int i, j, key, row = 0;

int count = 0;

int n = 100;

clock\_t start;

double runtime;

//Seed the random number generator

srand(time(NULL));

//Perform 5 rounds of insertion sort, doubling n each time

while (count < 8) {

//Add n values to the vector

for (i = 0; i < n; i++) {

//Generate a random value between 0 and 10,000

values.push\_back(rand() % 10001);

}

//Start the clock

start = clock();

//Perform Stooge Sort on the vector

stoogeSort(values, 0, (n-1));

//Calculate the runtime

runtime = (clock() - start) / (double) CLOCKS\_PER\_SEC;

cout << "STOOGE SORT RUNTIME FOR " << n << " VALUES: " << runtime << endl;

//Increment count and double n-value

count++;

n \*= 2;

}

return 0;

}

**c)**

**d)** An exponential curve best fits the StoogeSort data set (n2). The equation generated by Excel was: 1E-05n2 - 0.0208n + 6.8494. The run time calculated in Problem 4 was , so the experimental results aren’t far off from the theoretical value!