Christopher Merrill

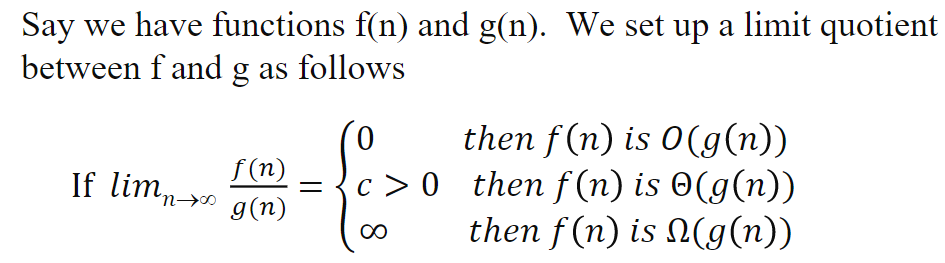
10/1/2017

CS325

Homework #1

1) To create the desired algorithm, I would first execute a merge sort. Merge sort is T(n) = 2T(n/2) + ϴ(n). Being ϴ(n), after applying the Master Theorem it would be ϴ(n log n), which is what we want in this case. From there we could apply binary searches to find the desired pairs that sum to x. The run time of the binary searches would then be irrelevant in the overall run time (for large n), since binary search requires ϴ(log n) and the ϴ(n log n) from merge sort would dominate. Therefore the overall run time using this method would be ϴ(n log n).

2) The following solutions were determined using the limit method presented in the lecture videos:



a. , by the limit method f(n) is O(g(n))

b. , by the limit method f(n) is ϴ (g(n))

c. , log n grows slower than as n approaches

infinity, so by the limit method f(n) is O(g(n))

d. , by the limit method f(n) is Ω(g(n))

e. , by the limit method f(n) is ϴ (g(n))

f. , 2n grows slower than n! as n approaches

infinity, so by the limit method f(n) is O(g(n))

3)

a.

To prove the following statement:

If f1(n) = O(g(n)) and f2(n) = O(g(n)) then f1(n) + f2(n) = O(g(n))

We can say the following:

If f1(n) = O(g(n)) then there exits constants c1 and n1 such that

I. 0 < f1(n) ≤ c1g(n) for all n ≥ n1

and since f2(n) = O(g(n)) there exists constants c2 and n2 such that

II. 0 < f2(n) ≤ c2g(n) for all n ≥ n2

Therefore

III. f1(n) ≤ g(n)

And

IV. f2(n) ≤ g(n)

For n ≥ (n1 + n2)

Now by adding III and IV we get:

0 < f1(n) + f2(n) ≤ g(n) + g(n) for n ≥ (n1 + n2)

* 0 < f1(n) + f2(n) ≤ 2 g(n) for n ≥ (n1 + n2)

So when c = 2 we get:

0 < f1(n) + f2(n) ≤ c g(n) for n ≥ (n1 + n2)

Which means

f1(n) + f2(n) = O (g(n))

b.

In order for the following statement to be true:

If f(n) = O(g1(n)) and f(n) = O(g2(n)) then g1(n)) = ϴ(g2(n))

The following has to be true:

By definition, f(n) = O(g1(n)) implies that there exists positive constants c1 and n1 such that

I. 0 ≤ f(n) ≤ c1g1(n) for all n ≥ n1

And, by definition, f(n) = O(g2(n)) implies that there exists positive constants c2 and n2 such that

II. 0 ≤ f(n) ≤ c2g2(n) for all n ≥ n2

Now in order for g1(n)) = ϴ(g2(n)) to be true, there must exit positive constants c3, c­4, and n3 such that

III. 0 ≤ c3g2(n) ≤ g1(n)  ­≤ c4g2(n) for all n ≥ n3

However it can be noted that if f(n) = n, g1(n) = n2and g2(n) = n3 (which satisfies I and II for c1 = 1, c2 = 1, and n1 = n2 = 2), there can be many cases of c3, c­4, and n3 such that III doesn’t hold. Such as c3 = 1, c4 = 1, and n3 = 2. Therefore the original statement is not always true.

4) Programs submitted to TEACH.

5)

a.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Insertion Sort Runtime Analysis Code:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <vector>

#include <time.h>

#include <stdlib.h>

#include <ctime>

#include <cstdio>

#include <iostream>

using namespace std;

int main()

{

vector<int>

int i, j, key, row = 0;

int count = 0;

int n = 1000;

clock\_t start;

double runtime;

//Seed the random number generator

srand(time(NULL));

//Perform 10 rounds of insertion sort, doubling n each time

while (count < 10) {

//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 Insertion Sort on the vector

for (i = 1; i < n; i++) {

key = values[i];

for(j = i - 1; (j >= 0) && (values[j] > key); j--) {

values[j+1] = values[j];

}

values[j+1] = key;

}

//Calculate the runtime

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

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

//Increment count and double n-value

count++;

n \*= 2;

}

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Merge Sort Runtime Analysis Code

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <vector>

#include <time.h>

#include <stdlib.h>

#include <ctime>

#include <cstdio>

#include <cstdlib>

#include <iostream>

using namespace std;

//Function declarations

void merge(vector<int>& left, vector<int>& right, vector<int>& vals);

void mergeSort(vector<int>& vsls);

int main()

{

vector<int> values;

int i, j, key, row = 0;

int count = 0;

int n = 1000;

clock\_t start;

double runtime;

//Seed the random number generator

srand(time(NULL));

//Perform 10 rounds of insertion sort, doubling n each time

while (count < 10) {

//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 Merge Sort on the vector

mergeSort(values);

//Calculate the runtime

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

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

//Increment count and double n-value

count++;

n \*= 2;

}

return 0;

}

void merge(vector<int>& left, vector<int>& right, vector<int>& vals) {

int leftLength = left.size();

int rightLength = right.size();

int i = 0, j = 0, k = 0;

//While there are still values in the left and right vectors

while (j < leftLength && k < rightLength)

{

//If the left vectors value is less than the right vectors, add it to

//the main vector and increment the left vector

if (left[j] < right[k]) {

vals[i] = left[j];

j++;

}

//Otherwise add the value from the right vector to the main vector

else {

vals[i] = right[k];

k++;

}

//Increment to the next value in the main vector

i++;

}

//Add any leftover values from the left vector

while (j < leftLength) {

vals[i] = left[j];

j++;

i++;

}

//Add any leftover values from the right vector

while (k < rightLength) {

vals[i] = right[k];

k++;

i++;

}

}

void mergeSort(vector<int>& vals) {

int mid = vals.size() / 2;

vector<int> left;

vector<int> right;

//Only perform mergeSort if the vector has 2 or more values

if (vals.size() > 1){

//Split the vector into left and right vectors

for (int j = 0; j < mid; j++) {

left.push\_back(vals[j]);

}

for (int j = 0; j < (vals.size()) - mid; j++) {

right.push\_back(vals[mid + j]);

}

//Recursive step

mergeSort(left);

mergeSort(right);

//Merge the sorted vectors

merge(left, right, vals);

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

b. I used 10 values of n for each sorting algorithm, starting at 1000 and doubling each time.

c.

As far as which graphs represent the data best, I would say that the insertion sort graph really shows how the runtime exponentially increases with n. The graph that includes both insertion and merge sort data is also a great representation of just how much faster merge sort is for large n values. The merge sort graph was a bit of a disappointment as it appears linear. I think if we could increase n much higher, we would start to see the logarithmic nature of the algorithm.

d. The regression equations/curves can be seen on the plots below:

e. The experimental data points for insertion sort are spot on. The regression curve is almost exactly x2 which is what we expect. Merge sort however appears linear, which disagrees with the theoretical runtime of log n. If we were to use only worst-case scenario data and look at much larger n values, I believe the logarithmic nature would be more evident.