James Palmer

HW1

CS 325 Spring 17

1. When you solve 8n^2 = 64nlgn, you get n=43.5593. Since you can see by the graph that 8n^2 starts to get larger after this, the answer would be that insertion sort runs faster until n=43. After that, merge sort runs faster.
2. a. f(n) =O(g(n)), limit -> 0

b. f(n) is Ω(g(n)), f(n) is O(n) and g(n) is O(logn). O(n) > O(logn). limit -> inf

c. f(n) =Θ(g(n)), both sides are O(logn)

d. f(n) =Θ(g(n)), both sides are O(n^2)

e. f(n) is O(g(n)), f(n) is O(n\*logn) and g(n) is O(n\*sqrtn). O(n\*logn) < O(n\*sqrtn)

f. f(n) =Θ(g(n)), both sides are O(k^n)

g. f(n) =O(g(n)), limit -> 0 or  f(n)=Θ(g(n)), limit(n->inf) f(n)/g(n) = 1/2 > 0

h. f(n) =O(g(n)), limit -> 0

i. f(n) is O(g(n)), f(n) is O(2^n) and g(n) is O(n!). O(2^n) < O(n!)

j. f(n) is O(g(n)), f(n) is O(logn) and g(n) is O(sqrtn). O(logn) < O(sqrtn).

1. a. Set min and max to the first value in the list. Compare pairs of values [2i], [2i+1] from i = 0 to n/2. The greater of the pair values will then be compared to max and the lesser will be compared to min.

b. n/2 comparisons for the initial pair comparisons. 2\*n/2 for each comparison to the min/max.

c.  [9, 3, 5, 10, 1, 7, 12]. min = 9, max = 9

[9>3], [5<10], [1<7], [12] 5 comparisons

9 == max; 3 < min, min = 3

10 > max, max = 10; 5 > min

7 < max; 1 < min, min = 1

12 > max, max = 12; 12 > min

min = 1, max = 12

1. a) If f1(n) = O(g(n)) and  f2(n) = O(g(n)) then f1(n)= Θ (f2(n) )

By definition;

f1(n) <= c1g(n) = 1/c1\*f1(n) <= g(n)

f2(n) <= c2g(n) = 1/c2\*f2(n) <= g(n)

This does not provide sufficient information to prove that f1(n)= Θ (f2(n))

Counterexample: f1(n) = n; f2(n) = n^2, g(n) = n!

f1(n) = O(g(n)) and  f2(n) = O(g(n)) but f1(n)=/= Θ (f2(n) )

b)  If f1(n) = O(g1(n)) and  f2(n) = O(g2(n)) then  f1(n)+ f2(n)= O(max{g1(n), g2(n)} )

By definition;

f1(n) ≤ c1g1(n) and f2(n) ≤ c2g2(n) for large n. Thus,

f1(n) + f2(n) ≤ c1g1(n) + c2g2(n)

≤ c1 max(g1(n), g2(n)) + c2 max(g1(n), g2(n))

≤ (c1 + c2) max(g1(n), g2(n)).

A)

void mergeSort(int front, int mid, int rear){

    int n1 = mid - front + 1;

    int n2 = rear - mid;

    int \*L = new int[n1 + 1];

    int \*R = new int[n2 + 1];

    register unsigned i = 0;

    for(; i < n1; ++i)

        L[i] = A[front + i];

    register unsigned j = 0;

    for(; j < n2; ++j)

        R[j] = A[mid + 1 + j];

    L[n1] = INT\_MAX;

    R[n2] = INT\_MAX;

    i = j = 0;

    for(; k <= rear; ++k){

        if(L[i] <= R[j]){

            A[k] = L[i++];

            c += j;

        }

        else

            A[k] = R[j++];

    }

}

void insertionSort(int front, int rear) {

    register int i, j;

    int key;

    for (i = front + 1; i <= rear; ++i) {

        key = A[i];

        j = i - 1;

        while (j >= front && A[j] > key) {

            A[j + 1] = A[j];

            ++c;

            --j;

        }

        A[j + 1] = key;

    }

}

 //keeping merge and insertion calls separate

int main(){

    register unsigned int i, j, n, k;

clock\_t start\_t, end\_t, total\_t;

    unsigned key;

    while(scanf("%u", &n) == 1){

        for(i = c = 0; i < n; ++i)

            scanf("%u", &A[i]);

        mergeSort(0, n - 1);

        printf("Exchange operations in merge: %u\n", c);

    }

while(scanf("%u", &n) == 1){

        for(i = z = 0; i < n; ++i)

            scanf("%u", &A[i]);

        insertionSort(0, n - 1);

        printf("Exchange operations in insertion: %u\n", c);

    }

    return 0;

}

2)Merge Sort acts as a O(nlgn)

Insertion sort acts as O(n2)

I added a quick clock sort to the code above which can be implemented like:

double t1, t2;

    for (length = 1000; length <= max\_length; )

    {

cout << "\nLength\t: " << length << '\n';

        read();

        t1 = clock();

        bubbleSort();

        t2 = clock();

        cout << "Merge Sort\t: " << (t2 - t1)/CLK\_TCK << " sec\n";

        read();

        t1 = clock();

        insertionSort();

        t2 = clock();

        cout << "Insertion Sort\t: " << (t2 - t1)/CLK\_TCK << " sec\n";

switch (length)

        {

        case 1000 :

            length = 1000;

            break;

        case 2000 :

            length = 2000;

            break;

        case 5000 :

            length = 5000;

            break;

        case 10000 :

            length = 10000;

            break;

        case 20000 :

            length = 20000;

            break;

        case 30000 :

            length = 30000;

            break;

        case 40000 :

            length = 40000;

            break;

        }

c)

d) The best type of curve that fits each data set is as seen from part C. Both parties are slightly close to their respective “theoretical running times” of x2 for Insertion and O(nlgn) for Merge.

e) As seen above, using the “theoretical running times” given by this weeks’ lectures, the output is far different. The experimental running times from my graphs in part C showed a steady increase rather than the default looking graph from part d. In comparison, similar to what was stated in this week’s lectures, coding in C is slower than python. For this reason, why my graphs have more extreme curves than part d. Also, my data might be constricted in some way because this week’s homework asked for random integers. Granted, I kept my straight forward and constantly increasing rather than a jumble of numbers. Insertion axis was closer to its theoretical running time of x2 than its counterpart. However, it proved that, even if my data was in a similar fashion. It is impossible to exactly recreate the theoretical running time when running a program.