Algorithm Analysis

Memory space and processor cycles are both finite resources. How can we measure the performance of an algorithm as the problem size increase? We use big O notation which describes the limiting behavior of an algorithm as an argument tends toward infinity. Here is an example of **O(n)**

```
for (int i=0; i<20; i++)
   {
   // perform command
   }
```

Where processor time increase linearly as 20 increases in size. If i=20 time units then at i=100 we use 100 time units for a linear increase in time. Sequential steps are normally ignored because they have a much smaller effect on run time than the multiplier steps.

O(n^2) represents the behavior of a doubly nested loop.

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4    {
 5    int d=10;
 6    int r=50;
 7
 8    for (int i=0; i<d; i++)
 9       {
10       for (int j=0; j<r; j++)
11          {
12          printf("array %d,%d = %d\n", i, j, i*j);
13          }
14       }
15    }
```

For example: a simple 15 line program with one loop inside of the other. Let's go line by line. Lines 1, 3, 5, and 6 are only performed once during the run of the program taking 4 time units. The inner loop, lines 11, 12, and 13, is performed ***r*** times. The outer loop, lines 9 through 14, is performed ***d*** times. This means line 12 is performed ***r * d*** time units. Put the parts together for ***r * d*** units + 4 units, **O(r*d + 4)**, or **O(n^2)** in big O notation. Those 4 units can be ignored, because as ***r * d*** grows larger, those 4 units remain the same. When ***r * d*** = 500, 4 units is only 4/500 or 0.8%. As ***r * d*** grows even larger the error of not adding those extra 4 units, is even smaller. Ignoring the linear steps in big O notation is normal, since you are looking for an upper bound on time use. A search, or sort algorithm, with **O(n^2)** is not so good, one of **O(n lg n)** is better. Two is the most often used logarithm base with algorithm analysis. We use lg instead of log or ln, which denote decimal or natural logarithms respectively.

```
Insertion sort (A)
                                           cost  time
1   for j <- 2 to length[A]                 c1   n
2      do key <- A[j]                       c2   n-1
3         i <- j - 1                        c3   n-1
4         while i > 0 and A[i] > key        c4   sum from j=2 to n of t_j
5            do A[i + j] <- A[i]            c5   sum from j=2 to n of t_j - 1
6               i <- i - 1                  c6   sum from j=2 to n of t_j - 1
7         A[i + 1] <- key                   c7   n-1
```

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \Sigma^n_{j=2} t_j + c_5 \Sigma^n_{j=2} (t_j-1) + c_6 \Sigma^n_{j=2} (t_j-1) + c_7 (n-1)$$

best case is when it is already sorted

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 (n-1) + c_7 (n-1)$$

$$= (c_1 + c_2 + c_3 + c_4 + c_7) n - (c_2 + c_3 + c_4 + c_7)$$

$= a n + b$ a linear function of n.

worst case is when it is in reverse sorted order

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 (n(n+1)/2 - 1) + c_5 (n(n-1)/2) + c_6 (n(n-1)/2) + c_7 (n-1)$$

$$= (c_4/2 + c_5/2 + c_6/2) n^2 + (c_1 + c_2 + c_3 + c_4/2 – c_5/2 - c_6/2 + c_7) n - (c_2 + c_3 + c_4 + c_7$$

$= a n^2 + b n + c$ a quadratic function of n

[from Introduction to Algorithms by Cormen, Leiserson, Rivest, & Stein]

```
BubbleSort(list)
                                       cost   time
for all elements of list                 c1   n-1
   if list[i] > list[i+1]                c2   n * (n-1)
      swap(list[i], list[i+1])           c3   n * (n-1)
return list


void BubbleSort(int arr[], int n)
   {
   int i, j;
                                       cost   time
   for (i = 0; i < n-1; i++)             c1   n
   // Last i elements already in place
      for (j = 0; j < n-i-1; j++)        c2   sum (n-i) from j=1 to n-i
         if (arr[j] > arr[j+1])          c3   sum (n-i) from j=1 to n-i
            swap(&arr[j], &arr[j+1]);    c4   ???
   }



if n = 10     c1 * n + c2 * 1+2+...+10 = 45
if n = 1000   c1 * 1000 + c2 * 499500 = 500500
```

To review :

- O(n) is a program which grows linearly with an increase in n.
- O(n^2) shows quadratic growth
- O(lg n) shows logarithmic growth

A divide and conquer program, such as a binary sort, has a logarithmic growth pattern. If one breaks a problem into quarters or even into eighths the pace of the logarithmic growth slows dramatically. These divisions are common in 2D and 3D problem spaces.

Using big O notation allows us to pick the optimal algorithm or data structure to solve our problem. If you know **n** will never grow very large, you can use an **O(n^2)** algorithm without any time penalty. With fast, modern processors you can perform brute force work on your problem and not get dinged too badly. But, it is best to optimize once you have solved the problem using brute force techniques. Usually a brute force answer will show you how, and where, to optimize your program for faster results. To me, a faster algorithm means I can increase the size of n. Finding the answer to a smaller problem using brute force is OK, but finding the answer to a much larger program, using a little programming elegance, is more fulfilling.

Description of the word find algorithm and analysis of its runtime.

```
1) Scan input word for non alpha characters, ignore them if found.
2) Convert all letters to lower case
3) Count the characters in input word into an alphabet array inWord.count[26]
      and total count to inWord.total

4) Input each word in the dictionary
5)    Scan input word for non alpha characters, ignore them if found.
6)    Convert all letters to lower case
7) Count the characters in the dictionary word into its alphabet array
      dictionary.count[26] and total count to dictionary.total

8)    for i = 0 to 25   // from a to z
9) if ( (dictionary.count[i] > inWord.count[i]) AND
        (dictionary.total >= Minimum number of letters) AND
           (dictionary.total <= inWord.total) )
10)              don't keep this word; break out of inner loop
11) else
12)   keep this word
13)   print words that pass test and save them to a disk file
```

```
Line 1 runs in  O(number of characters in input word) time
Line 2 runs in  O(number of characters in input word) time
Line 3 runs in  O(number of characters in input word * 26) time

Line 5 runs in  O(number of words in dictionary * number of characters in word) time
Line 6 runs in  O(number of words in dictionary * number of characters in word) time
Line 7 runs in  O(number of words in dictionary * number of characters in word * 26) time

Line 9 runs in  O(number of words in dictionary * 26) time
Line 10 runs in O(number of words in dictionary * 26 * 1 - the fraction of success) time

Line 12 runs in O(number of words in dictionary * 26 * the fraction of success) time
Line 13 runs in O(number of words in dictionary * the fraction of success) time
```

So, if **n** represents the number of words in the dictionary and **k** represents the number of characters in the longest word in the dictionary, the algorithm runs in **O(nk)** time. This is from line 7, the most time consuming part of the algorithm.

```
void buildQuadTree(struct quad *here, struct quad *node)
   {
   if (node->x <= here->x && node->y <= here->y)
      {
      if (here->sw != NULL)  //  -,- quadrant
         buildQuadTree(here->sw, node);
      else
         {
         here->sw = node;
         return;
         }
      }

   if (node->x <= here->x && node->y > here->y)
      {
      if (here->nw != NULL)   //  -,+ quadrant
         buildQuadTree(here->nw, node);
      else
         {
         here->nw = node;
         return;
         }
      }

   if (node->x > here->x && node->y <= here->y)
      {
      if (here->se != NULL)   //  +,- quadrant
         buildQuadTree(here->se, node);
      else
         {
         here->se = node;
         return;
         }
      }

   if (node->x > here->x && node->y > here->y)
      {
      if (here->ne != NULL)   //  +,+ quadrant
         buildQuadTree(here->ne, node);
      else
         {
         here->ne = node;
         return;
         }
      }
   }
```

Building a quadTree works most efficiently when the nodes fill the two dimensional space evenly. At each level of the recursion, you process one quarter the number nodes as the previous level. By dividing the work by ¼ at each level, the algorithm runs in **$O(\log_4 n)$** time. If **n** is equal to 1000 then $\log_4(1000)$ equals 4.98289… Testing buildQuadTree() with 1000 nodes shows it requires 5 to 6 levels to hold all of the nodes. Time and space complexity of the algorithm is **$O(\log_4 n)$**.

```
void searchQuadTree(struct quad *here, struct quad *node)
   {
   int x1, x2, y1, y2, d;
   int dist = RAD * RAD;       // is node within RAD of here?
   static int q;               // how many close neighbors

   if (here->sw != NULL)       //  -,- quadrant
      searchQuadTree(here->sw, node);
   if (here->nw != NULL)       //  -,+ quadrant
      searchQuadTree(here->nw, node);

   x1 = here->x; y1 = here->y;
   x2 = node->x; y2 = node->y;
   d = ((x2-x1)*(x2-x1))+((y2-y1)*(y2-y1));

   if (d <= dist)              // difference of squares of the distance
      {                        // eliminates the need for sqrt()
      qNode[q].x = here->x;
      qNode[q].y = here->y;    // add node to list if close enough
      qNode[q].dist = d;
      q++;
      }

   if (here->se != NULL)       //  +,- quadrant
      searchQuadTree(here->se, node);
   if (here->ne != NULL)       //  +,+ quadrant
      searchQuadTree(here->ne, node);
   }
```

This search algorithm also divides **n** by four at each level of the search. searchQuadTree() runs in **O($\log_4$ n)** also.

https://en.wikipedia.org/wiki/Analysis_of_algorithms

http://aofa.cs.princeton.edu/10analysis/

http://cs.lmu.edu/~ray/notes/alganalysis/

http://personal.us.es/almar/cg/09quadtrees.pdf