CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

LEC02
PROGRAM ANALYSIS (40-51)

Lecture slide courtesy of Prof. Steven Skiena

EXERCISE 2-4

2-4. *[8]* What value is returned by the following function? Express your answer as a function of *n*. Give the worst-case running time using Big Oh notation.

```
function conundrum(n)
  r := 0
  for i := 1 to n do
    for j := i + 1 to n do
      for k := i + j − 1 to n do
        r := r + 1
  return(r)
```

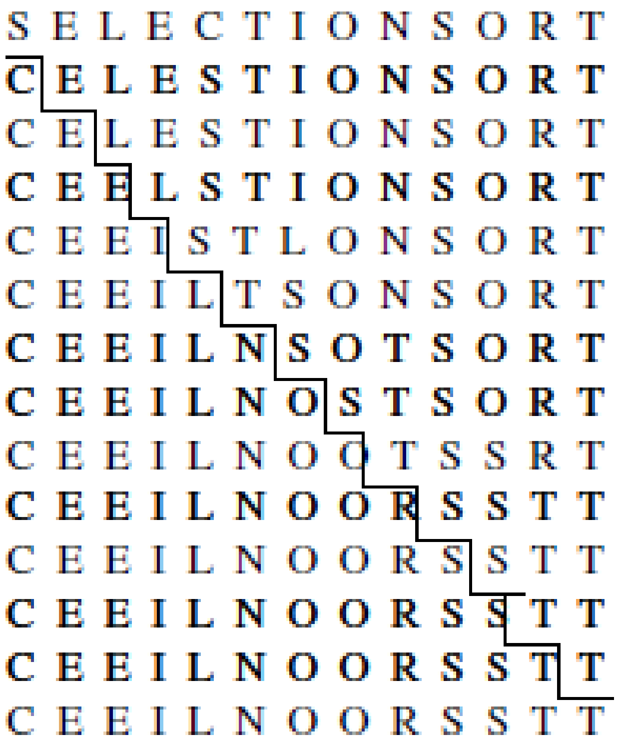
> conundrum(1) [1] 3
> conundrum(2) [1] 6
> conundrum(3) [1] 12
> conundrum(4) [1] 21
> conundrum(5) [1] 35
> conundrum(6) [1] 54
> conundrum(7) [1] 80
> conundrum(8) [1] 113
> conundrum(9) [1] 155
> conundrum(10) [1] 206

$$\sum_{i=1}^{n} 1 = n \qquad \sum_{i=1}^{n} c = nc \qquad \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad \sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$$

- r<-1:10
- r^3
- 1 8 27 64 125 216 343 512 729 1000

REASONING ABOUT EFFICIENCY: SELECTION SORT




```
selection_sort(int s[], int n)
{
        int i,j;                /* counters */
        int min;                /* index of minimum */

        for (i=0; i<n; i++) {
                min=i;
                for (j=i+1; j<n; j++)
                        if (s[j] < s[min]) min=j;
                swap(&s[i],&s[min]);
        }
}
```

```
void selectionSort(int[] array, int startIndex)
{
    if ( startIndex >= array.length - 1 )
        return;
    int minIndex = startIndex;
    for ( int index = startIndex + 1; index < array.length; index++ )
    {
        if (array[index] < array[minIndex] )
            minIndex = index;
    }
    int temp = array[startIndex];
    array[startIndex] = array[minIndex];
    array[minIndex] = temp;
    selectionSort(array, startIndex + 1);
}
```

SELECTION SORT WORST CASE ANALYSIS

- The outer loop goes around n times.
- The inner loop goes around at most n times for each it eration of the outer loop
- Thus selection sort takes at most n*n -> $O(n^2)$ time in the worst case.

```
selection_sort(int s[], int n)
{
        int i,j;                /* counters */
        int min;                /* index of minimum */

        for (i=0; i<n; i++) {
                min=i;
                for (j=i+1; j<n; j++)
                        if (s[j] < s[min]) min=j;
                swap(&s[i],&s[min]);
        }
}
```

MORE CAREFUL ANALYSIS

```
selection_sort(int s[], int n)
{
        int i,j;                /* counters */
        int min;                /* index of minimum */

        for (i=0; i<n; i++) {
                min=i;
                for (j=i+1; j<n; j++)
                        if (s[j] < s[min]) min=j;
                swap(&s[i],&s[min]);
        }
}
```

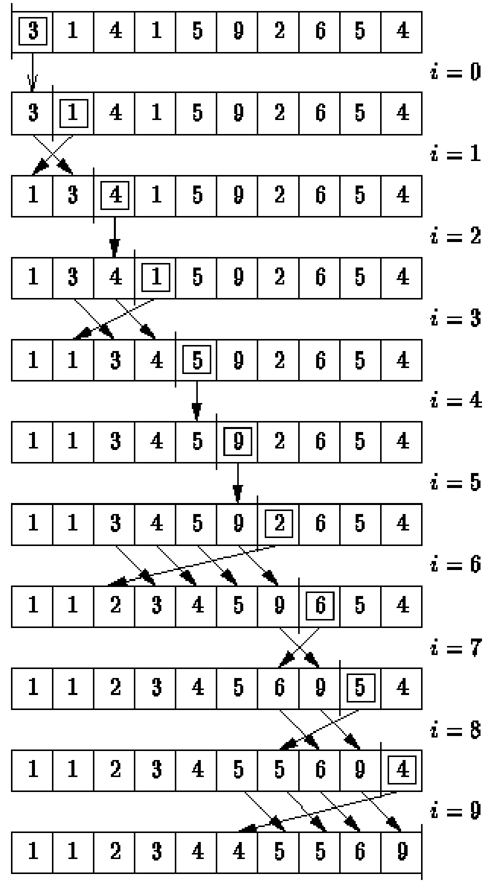
- An exact count of the number of times the *if* statement is executed is given by:

$$S(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} n - i - 1$$

$$S(n) = (n-2) + (n-3) + \ldots + 2 + 1 + 0 = (n-1)(n-2)/2$$

- Thus the worst case running time is $\Theta(n^2)$.
- Can we say $O(n^2)$ or $\Omega(n^2)$?

REASONING ABOUT EFFICIENCY: INSERTION SORT




```
for (i=1; i<n; i++) {
        j=i;
        while ((j>0) && (s[j] < s[j-1])) {
                swap(&s[j],&s[j-1]);
                j = j-1;
        }
}
```

```
public static int[] RecursiveInsertionSort(int[] array, int r
{
    int i;
    if (n > 1)
       RecursiveInsertionSort(array, n - 1);
    else {
       int k = array[n];
       i = n - 1;
       while (i >= 0  & &  array[i] > k) {
           array[i + 1] = array[i];
           i = i - 1;
       }
       array[i + 1] = k;
    }
    return array;
}
```

INSERTION SORT WORST CASE ANALYSIS

- How often does the inner *while* loop iterate?
  - two different stopping conditions:
  - One to prevent us from running off the bounds of the array ($j > 0$)
  - One to mark when the element finds its proper place in sorted order ($s[j] < s[j-1]$).
- Since **worst-case analysis** seeks an upper bound on the running time, we ignore the early termination and assume that this loop *always* goes around i times.
- insertion sort must be a quadratic-time algorithm, i.e. $O(n^2)$.

REASONING ABOUT EFFICIENCY: STRING PATTERN MATCHING

Problem: Substring Pattern Matching
Input: A text string *t* and a pattern string *p*.
Output: Does *t* contain the pattern *p* as a substring, and if so where?

```
a b
  a b b
    a
      a b b a
-------------
a a b a b b a
```

Searching for the substring *abba* in the text *aababba*.

```
int findmatch(char *p, char *t)
{
        int i,j;                        /* counters */
        int m, n;                       /* string lengths */

        m = strlen(p);
        n = strlen(t);

        for (i=0; i<=(n-m); i=i+1) {
                j=0;
                while ((j<m) && (t[i+j]==p[j]))
                        j = j+1;
                if (j == m) return(i);
        }

        return(-1);
}
```

WORST CASE ANALYSIS

What is the worst-case running time of these two nested loops?

```
int findmatch(char *p, char *t)
{
        int i,j;                        /* counters */
        int m, n;                       /* string lengths */

        m = strlen(p);
        n = strlen(t);

        for (i=0; i<=(n-m); i=i+1) {
                j=0;
                while ((j<m) && (t[i+j]==p[j]))
                        j = j+1;
                if (j == m) return(i);
        }

        return(-1);
}
```

$\Theta(n)$

$\Theta(m)$

$\Theta(n-m)$

$O(m+2)$

$O(n+m+(n-m)(m+2))$?

$n \geq m, n \leq nm,$

$\boxed{O(nm)}$

MATRIX MULTIPLICATION

Problem: Matrix Multiplication

Input: Two matrices, A (of dim. $x \times y$) and B (dim. $y \times z$).

Output: An $x \times z$ matrix C where $C[i][j]$ is the dot product of the ith row of A and the jth column of B.

```
for (i=1; i<=x; i++)
        for (j=1; j<=y; j++) {
                C[i][j] = 0;
                for (k=1; k<=z; k++)
                        C[i][j] += A[i][k] * B[k][j];
        }
```

WORST CASE ANALYSIS

The number of multiplications $M(x, y, z)$ is given by the following summation:

$$M(x,y,z) = \sum_{i=1}^{x} \sum_{j=1}^{y} \sum_{k=1}^{z} 1$$

Sums get evaluated from the right inward. The sum of z ones is z, so

$$M(x,y,z) = \sum_{i=1}^{x} \sum_{j=1}^{y} z$$

The sum of y zs is just as simple, yz, so

$$M(x,y,z) = \sum_{i=1}^{x} yz$$

Finally, the sum of x yzs is xyz.

Thus the running of this matrix multiplication algorithm is $O(xyz)$.

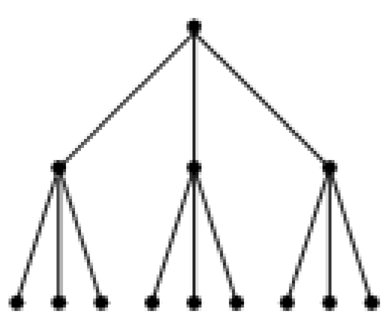
If we consider the common case where all three dimensions are the same, this becomes $O(n^3)$—i.e. , a cubic algorithm.

LOGARITHMS

- It is important to understand deep in your bones what logarithms are and where they come from.
- A logarithm is simply an inverse exponential function. Saying $b^x = y$ is equivalent to saying that $x = \log_b y$.
- Logarithms reflect how many times we can double something until we get to n, or halve something until we get to 1.
- Logarithms arise in any process where things are repe atedly halved.

BINARY SEARCH

- In binary search we throw away half the possible number of keys after each comparison. Thus twenty comparisons suffice to find any name in the million-name Manhattan phone book!
- How many time can we halve n before getting to 1?
- Answer: ceiling(lgn)



A height h tree with d children per node as d^h leaves.

Here $h = 2$ and $d = 3$

LOGARITHMS AND TREES

- How tall a binary tree do we need until we have n leaves? The number of potential leaves doubles with each level.
- How many times can we double 1 until we get to n?
- Answer: ceiling(lgn)

MORE ABOUT TREES

A binary tree of height 1 can have up to 2 leaf nodes

What is the height *h* of a rooted binary tree with *n* leaf nodes?

- Note that the number of leaves doubles every time we increase the height by one.
- To account for *n* leaves, $n = 2^h$ which implies that $h = \log_2 n$.

What if we generalize to trees that have *d* children, where $d = 2$ for the case

of binary trees?

- A tree of height 1 can have up to *d* leaf nodes, while one of height

two can have up to d^2 leaves.

The number of possible leaves multiplies by *d* every time we increase the height by one, so to account for *n* leaves, $n = d^h$ which implies that $h = \log_d n$,

LOGARITHMS AND BITS

- How many bits do you need to represent the numbers from 0 to 2^i - 1?
- Each bit you add doubles the possible number of bit patterns,
- so the number of bits equals $lg(2^i) = i$

LOGARITHMS AND MULTIPLICATION

- Recall that

$$\log_a(xy) = \log_a(x) + \log_a(y)$$

- This is how people used to multiply before calculators, and remains useful for analysis.
- What if $x = a$?

$$\log_a n^b = b \cdot \log_a n$$

THE BASE IS NOT ASYMPTOTICALLY IMPORTANT

Recall the definition, $c^{\log c\, x} = x$ and that

$$\log_b a = \frac{\log_c a}{\log_c b}$$

Thus $\log_2 n = \frac{1}{\log_{100} 2} * \log_{100} n$. Since $\frac{1}{\log_{100}\ 2}$ = 6.643 is just a constant, it does not matter in the Asymptotic notations .

FEDERAL SENTENCING GUIDELINES

- 2F1.1. Fraud and Deceit; Forgery; Offenses Involving Altered or Counterfeit Instruments other than Counterfeit Bearer Obligations of the United States.
  - (a) Base offense Level: 6
  - (b) Specific offense Characteristics
- (1) If the loss exceeded $2,000, increase the offense level as follows:

Loss(Apply the Greatest)	Increase in Level
(A) $2,000 or less	no increase
(B) More than $2,000	add 1
(C) More than $5,000	add 2
(D) More than $10,000	add 3
(E) More than $20,000	add 4
(F) More than $40,000	add 5
(G) More than $70,000	add 6
(H) More than $120,000	add 7
(I) More than $200,000	add 8
(J) More than $350,000	add 9
(K) More than $500,000	add 10
(L) More than $800,000	add 11
(M) More than $1,500,000	add 12
(N) More than $2,500,000	add 13
(O) More than $5,000,000	add 14
(P) More than $10,000,000	add 15
(Q) More than $20,000,000	add 16
(R) More than $40,000,000	add 17
(Q) More than $80,000,000	add 18

MAKE THE CRIME WORTH THE TIME

- The increase in punishment level grows *logarithmically* in the amount of money stolen.
- Thus it pays to commit one big crime rather than many small crimes totaling the same amount.