Algorithmic Cost and Complexity

There are *two aspects* of algorithmic performance:

• Time
- Instructions take time.
- How fast does the algorithm perform?
- What affects its runtime?

• Space
- Data structures take space
- What kind of data structures can be used?
- How does choice of data structure affect the runtime?

Measuring Performance

For example: A simple calculator :

Perform the four basic arithmetic functions:
- Addition
- Subtraction
- Multiplication
- Division

Prompt the user for:
- Operand 1
- Operand 2
- Operator

Algorithm Calculator

```
double  op1          // 1st operand
        op2          // 2nd operand
        answer ;     // result
char operator ;      // operator

// obtain operands and operator from user
printf( “Enter the first operand: ” );
scanf(“%lf”, &op1 );
printf( “Enter the second operand: ” );
scanf(“%lf”, &op2 );
printf( “Enter the operator: ” );
scanf(“%c”, &operator );

// perform the calculation
if ( operator == ‘+’ )
    answer = op1 + op2;

if ( operator == ‘-’ )
    answer = op1 - op2;

if ( operator == ‘*’ )
    answer = op1 * op2;

if ( operator == ‘/’ )
    answer = op1 / op2;

printf( “The answer is %f \n”, answer );

// end algorithm Calculator
```


Analyzing Work Done

How many operations does Calculator do?
- read/write pairs (to obtain data)
- Testing conditionals
- Branching
- Performing operation
- Assigning variables

Note:
We will ignore the *read/write* instructions. They deal with the world “outside the algorithm” and involve factors beyond what we care about here.

Measures of Work
(ignoring read/write pairs)

- What’s the ***best case***?
  Addition - four tests (@ 2 each)
  - one add
  - one assignment
  - total: 10
- What’s the ***worst case***?
  Division - four tests (@ 2 each)
  - one divide
  - one assignment
  - total: 10
- What’s the ***average (expected) case***?
  - 10

A Better Way?

```
// Perform the calculation
if ( operator == '+' )
    answer = op1 + op2;
else if ( operator == '-' )
    answer = op1 - op2;
else if ( operator == '*' )
    answer = op1 * op2;
else if ( operator == '/' )
    answer = op1 / op1;

printf( "The answer is %f\n", answer );
// end of algorithm
```

Measures of Work

(ignoring read/write pairs)

- What's the ***best case***?
  Addition - one test (@ 2 each)
  - one add
  - one assignment
  - total: 4
- What's the ***worst case***?
  Division - four tests (@ 2 each)
  - one divide
  - one assignment
  - total: 10
- What's the ***average (expected) case***?
  - (4+6+8+10)/4 = 7

The Dangers of "Average" Work

In many circumstances, the assumption of random distribution of input values is a faulty one.

What about a cash register?
• Addition operators most frequent (ring up an item)
• Subtraction less frequent (use a coupon)
• Multiplication rare (buy many of same item)
• Division very rare (???)

The average work in this situation would migrate somewhat towards 4 from the mean of 7 suggested by the assumption of random data.

Don't assume random distribution without reason.

Algorithm Analysis: Loops

Consider the following nested loops (LOOP1 and LOOP2) intended to sum each of the rows in an NxN two dimensional array, storing the row sums in a one-dimensional array `rows` and the overall total in `grandTotal`.

LOOP 1:

```
grandTotal = 0;
for (k=0; k<n-1; ++k)
    rows[k] = 0;
    for (j = 0; j <n-1; ++j){
        rows[k] = rows[k] + matrix[k][j];
        grandTotal = grandTotal + matrix[k][j];
    }
}
```

LOOP 2:

```
grandTotal =0;
for (k=0; k<n-1; ++k)
    rows[k] = 0;
    for (j = 0; j <n-1; ++j)
        rows[k] = rows[k] + matrix[k][j];
    grandTotal = grandTotal + rows[k];
}
```

- What is the number of addition operations? $2N^2$ versus $N^2 + N$
- Assuming we're working with a hypothetical computer that requires 1 microsecond to perform an addition, for N = 1000, loop 1 would take 2 sec., loop 2 would require just over 1 second. (For N= 100,000 time would be approx. 6 hrs and 3 hours respectively)

Big-O Notation

- It is a method of algorithm classification.

Definition: Suppose there exists a function *f(n)* defined on nonnegative integers such that the number of operations required by an algorithm for an input size *n* is less than or equal to some constant *c* times *f(n) (i.e. c*f(n))* for all but finitely many *n*.

That is, the number of operations is at worst proportional to *f(n)* for all large values of *n*.

Such an algorithm is said to be an O[*f(n)*] algorithm.

- Loop 1 and Loop 2 are both in the same big-O category: $O(N^2)$

Example 1:

Use big-O notation to analyze the time efficiency of the following fragment of C code:

```
for(k = 1; k <= n/2; k++)
{
    .
    .
    for (j = 1; j <= n*n; j++)
    {
        .
        .
    }
}
```

Since these loops are nested, the efficiency is $n^3/2$, or $O(n^3)$ in big-O terms.

Thus, for two loops with $O[f_1(n)]$ and $O[f_2(n)]$ efficiencies, the efficiency of the nesting of these two loops is $O[f_1(n) * f_2(n)]$.

Example 2:

Use big-O notation to analyze the time efficiency of the following fragment of C code:

```
for (k=1; k<=n/2; k++)
{
    .
    .
}
for (j = 1; j <= n*n; j++)
{
    .
    .
}
```

The number of operations executed by these loops is the sum of the individual loop efficiencies. Hence, the efficiency is $n/2+n^2$, or $O(n^2)$ in big-O terms.

Thus, for two loops with $O[f_1(n)]$ and $O[f_2(n)]$ efficiencies, the efficiency of the sequencing of these two loops is $O[f_D(n)]$ where $f_D(n)$ is the dominant of the functions $f_1(n)$ and $f_2(n)$.

Example 3:

Use big-O notation to analyze the time efficiency of the following fragment of C code:

```
k = n;
while (k > 1)
{
    .
    .
    k = k/2;
}
```

Since the loop variable is cut in half each time through the loop, the number of times the statements inside the loop will be executed is $\log_2 n$.

Thus, an algorithm that halves the data remaining to be processed on each iteration of a loop will be an $O(\log_2 n)$ algorithm.

Classification of Algorithms

Algorithms whose efficiency is dominated by a $\log_a n$ term are often called *logarithmic algorithms*. Because $\log_a n$ will increase much more slowly than n itself, logarithmic algorithms are generally very efficient.

Algorithms whose efficiency can be expressed in terms of a polynomial of the form

$$a_m n^m + a_{m-1} n^{m-1} + \ldots + a_2 n^2 + a_1 n + a_0$$

are called *polynomial algorithms*. Such algorithms are $O(n^m)$. For m=1, 2, or 3, they are called *linear*, *quadratic* or *cubic* algorithms, respectively.

Algorithms with efficiency dominated by a term of the form a^n are called *exponential algorithms*. They are of more theoretical rather than practical interest because they cannot reasonably run on typical computers for moderate values of n.

Complexity of Linear Search

In measuring performance, we are generally concerned with how the amount of work varies with the data. Consider, for example, the task of searching a list to see if it contains a particular value.

• A useful search algorithm should be *general*.

• Work done varies with the size of the list

• What can we say about the work done for list of *any* length?

```
i = 0;

while (i < MAX && this_array[i] != target)
     i = i + 1;

if (i <MAX)
    printf ( “Yes, target is there \n” );
else
    printf( “No, target isn’t there \n” );
```


Order Notation

How much work to find the target in a list containing N elements?
Note: we care here only about the *growth rate* of work. Thus, we *toss out all constant values*.

Best Case - It’s the first value
    “order 1,” O(1)
Worst Case - It’s the last value, N
    “order N,” O(N)
Average - N/2 (if value is present)
    “order N,” O(N)

- Best Case work is *constant*; it does not grow with the size of the list.
- Worst and Average Cases work is *proportional* to the size of the list, N.

Order Notation

O(1) or "Order One":

- does *not* mean that it takes only one operation
- *does* mean that the work *doesn't change* as N changes
- is a notation for *"constant work"*

O(N) or "Order N":

- does *not* mean that it takes N operations
- *does* mean that the work changes in a way that is *proportional* to N
- is a notation for *"work grows at a linear rate"*

Improving on Linear Search

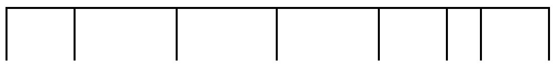
Can we do better?
Array of the Social Security Numbers of all students in this class.
Index *is* the Social Security Number.

000 00 0001
000 00 0002
000 00 0003
. . .
999 99 9998
999 99 9999

Results is O(1), but wastes HUGE space

Getting Realistic - Binary Search

- Assume a sorted list of 16 SSNs
- Search for one via *binary search*
- How much work is done now?



- **Worst case:**

16 / 2	Comparison #1
8 / 2	Comparison #2
4 / 2	Comparison #3
2 / 2	Comparison #4

- For 16 items, it takes 4 comparisons

- In general, it takes ($\log_2 N$) searches
  ($\log_2 16 = 4$ because $2^4 = 16$)

- Binary search is an *O(logN) algorithm*
 Since, it repeatedly cuts its remaining work in half, binary search involves work that grows at a rate proportional to the log of N.

How much better is *O(logN)*?

N	***O(logN)***
16	4
64	6
256	8
1024 (1Kilo)	10
16,384	14
131,072	17
262,144	18
524,288	19
1,048,576 (1Meg)	20
1,073,741,824 (1Gig)	30

- As N gets large, the difference becomes great.

Data Structures and Complexity

- Can we assume that data are:
  - sorted and
  - stored in an appropriate sized array?

```
#define MAX 30
int array[MAX];
```

- Still... we need to know what N is in advance to declare an Array.

- Binary Search Tree (BST) can be very valuable, if N is not predictable. A BST allows O(log N) search performance if certain conditions are met: The tree must be full and balanced.

Data Structures and Complexity

	Traverse	Search	Insert
Linked List (unsorted)	N	N	1
Linked List (sorted)	N	N	N
Array (unsorted)	N	N	1
Array(sorted)	N	log N	N
Binary Tree	N	N	log N
BST	N	log N	log N

Insertion = cost to find location + cost of insertion.

Bubblesort Revisited

Bubblesort works by comparing and swapping values in a list

23	78	45	8	32	56
23	78	45	8	32	56
23	78	45	8	32	56
23	78	8	45	32	56
23	8	78	45	32	56
8	23	78	45	32	56

Complexity of Bubblesort

How many comparisons will the inner loop do?
(N-1) + (N-2) + (N-3) + ... + 1

Average: N/2 for each "pass"

How many "passes" (outer loop) are there?
N – 1

Tossing constants:
- Each loop involves O(N) work
- Inner will be executed for each iteration of outer

So what is the complexity?

$$O(N) * O(N) = O(N^2)$$

```
void bubbleSort(int list[], int last)
{
   int current;

   for (current = 0; current < last; current ++)
      bubbleUp(list, current, last);
   return;
}

/* Move the lowest element in unsorted portion
   to the current element in the unsorted portion.
   Pre list must contain at least one element
       current: beginning of unsorted portion
       last: identifies end of the unsorted data
   Post array segment has been rearranged so that
       lowest element now at beginning of unsorted
        portion
*/
void bubbleUp(int list[],
              int current,
              int last)
{
   int walker;
   int temp;

   for (walker=last; walker > current; walker--)
      if(list[walker] < list[walker - 1]){
         temp = list[walker];
         list[walker] = list[walker – 1];
         list[walker-1] = temp;
      }

   return;
}
```


Comparison of *N*, *logN* and N^2

N	O(LogN)	O(N^2)
16	4	256
64	6	4K
256	8	64K
1,024	10	1M
16,384	14	256M
131,072	17	16G
262,144	18	6.87E+10
524,288	19	2.74E+11
1,048,576	20	1.09E+12
1,073,741,824	30	1.15E+18

Complexity of MergeSort

Merge sort requires $O(N \log_2 N)$ comparisons.

<u>The reasoning:</u>
All the merge operations across any given level of the trace diagram will require $O(N)$ comparisons. There are $\log_2 N$ levels. Hence, the overall efficiency is $O(N \log_2 N)$.

In level 1: There is one merge operation. We're merging 2 lists with size N/2.
In level 2: There are two merge operations. We're merging 2 pairs of lists with size N/4.
.
.
In the last level (i.e. level $\log_2 N$): There are N/2 merge operations. We're merging N/2 pairs of lists with size 1.

How much work is involved in each level?

- Each of the N numerical values is compared or copied during each level
- Therefore, the work for each level is *O (N)*

Thus the total for MergeSort is:

$$O(\log N) * O(N) = O(N\log N)$$

Example Problems

1. Algorithm A runs in $O(N^2)$ time, and for an input size of 4, the algorithm runs in 10 milliseconds, how long can you expect it to take to run on an input size of 16?
2. Algorithm A runs in $O(\log_2 N)$ time, and for an input size of 16, the algorithm runs in 28 milliseconds, how long can you expect it to take to run on an input size of 64?
3. Algorithm A runs in $O(N^3)$ time. For an input size of 10, the algorithm runs in 7 milliseconds. For another input size, the algorithm takes 189 milliseconds. What was that input size?
4. For an $O(N^k)$ algorithm, where k is a positive rational number, a friend tells you that instance of size M took 16 seconds to run. You run an instance of size 4M and find that it takes 256 seconds to run. What is the value of k?
5. Algorithm A runs in $O(N^3)$ time and Algorithm B solves the same problem in $O(N^2)$ time. If algorithm A takes 5 milliseconds to complete for an input size of 10, and algorithm B takes 20 milliseconds for an input size of 10, what is the input size that you expect the two algorithms to perform about the same?
6. For an $O(N^3)$ algorithm, an instance with N = 512 takes 56 milliseconds. If you used a different-sized data instance and it took 7 milliseconds how large must that instance be?

Answers

1. Algorithm A runs in $O(N^2)$ time, and for an input size of 4, the algorithm runs in 10 milliseconds, how long can you expect it to take to run on an input size of 16?

$$\frac{4^2}{10ms} = \frac{16^2}{x} \Rightarrow x = 160ms$$

2. Algorithm A runs in $O(\log_2 N)$ time, and for an input size of 16, the algorithm runs in 28 milliseconds, how long can you expect it to take to run on an input size of 64?

$$\frac{\log 16}{28\ ms} = \frac{\log 64}{x} \Rightarrow x = 42\ ms$$

3. Algorithm A runs in $O(N^3)$ time. For an input size of 10, the algorithm runs in 7 milliseconds. For another input size, the algorithm takes 189 milliseconds. What was that input size?

$$\frac{10^3}{7\ ms} = \frac{N^3}{189} \Rightarrow N = 30$$

4. For an $O(N^k)$ algorithm, where k is a positive rational number, a friend tells you that instance of size M took 16 seconds to run. You run an instance of size 4M and find that it takes 256 seconds to run. What is the value of k?

$$\frac{M^k}{16\ ms} = \frac{(4M)^k}{256} \Rightarrow 4^k = \frac{256}{16} \Rightarrow k = 2$$

5. Algorithm A runs in $O(N^3)$ time and Algorithm B solves the same problem in $O(N^2)$ time. If algorithm A takes 5 milliseconds to complete for an input size of 10, and algorithm B takes 20 milliseconds for an input size of 10, what is the input size that you expect the two algorithms to perform about the same?

For algorithm A: $O(N^3)$ means
execution time <= $c1 * N^3$
for N = 10 execution time is 5ms = $c1 * 10^3$
so , $c1 = 5 / 10^3$

For algorithm B: $O(N^2)$ means
execution time <= $c2 * N^3$
for N = 10 execution time is 20 ms = $c2 * 10^2$
so , $c2 = 20 / 10^2$

what is N for which
$c1 * N^3 = c2 * N^2$

Substitute for c1 and c2:
$5 / 10^3 * N^3 = 20 / 10^2 * N^2$
=> N = 40

6. For an $O(N^3)$ algorithm, an instance with N = 512 takes 56 milliseconds. If you used a different-sized data instance and it took 7 milliseconds how large must that instance be?

$$\frac{512^3}{56ms} = \frac{N^3}{7} \Rightarrow N = 256ms$$