Algorithms

What is a good algorithm? Two often contradictory goals are:

1. An algorithm that is easy to understand, code, and debug
2. An algorithm that runs as fast as possible

If the program is to be used once or a few times, goal (1) is most important. (Cost of programmers time >> cost of running program)

If the program is to be used many times, then goal (2) is financially sound (provided the resulting program is significantly faster).

Often it is best to implement a simple algorithm and then determine the benefit to be had by writing a more complicated program (prototype).

The amount of work or running time of an algorithm depends on

1. The input to the algorithm
2. The quality of code generated by the compiler
3. The speed of the machine used to execute the algorithm
4. The time complexity of the algorithm (i.e., the method used by the algorithm to solve the problem)

How shall we measure the running time? Our measure should allow us to compare two algorithms so that we can determine whether one is more efficient. Possible measures are

- execution time, and
- number of instructions executed.

But both of these are dependent on the computer, language, and compiler used.

Role of abstraction: we identify abstract operations upon which the algorithm is based in order to separate the analysis from the implementation details (we are not going to count the number of microseconds it takes to execute instructions.)

Example: the basic operation in sorting algorithms is "comparison" and the number of comparisons in algorithm 1 is x and algorithm 2 is y.

As long as the total number of operations performed is roughly proportional to the number of basic operations, we have a good criterion for comparing several algorithms.

How can we describe the running time (number of basic operations, amount of work done, complexity) of an algorithm. Note that for the same algorithm, the actual number of operations performed depends on the size of the input (e.g. sorting 100 vs. 1000 names).

We will define the running time of an algorithm as a function t of the size of the input n.

problem	size of input
sort a list of numbers	# of entries in the list
solving a graph problem	# of nodes or edges or both

For many algorithms, the running time is a function of the particular input, and not just the input size. In that case, t(n) is defined to be the **worst case** running time (i.e. maximum over all inputs of size n). This gives us an upper bound on the running time. We are also interested in $t_{avg}(n)$ which is the average, over all inputs of size n, of the running time.

What is the average input? "average input" frequently has no meaning. Average-case bounds are usually hard to compute. Will the worst case ever appear in practice?

Given the worst case running time of an algorithm, how do we know whether or not we have an optimal algorithm? In other words, how much work is necessary and sufficient to solve the problem?

Two tasks to find a good algorithm

1. Devise an algorithm A and find a function *t* such that, for inputs of size n, A does at most t(n) steps in the worst case. (upper bound)
2. For some function *f*, prove that for any algorithm in the class under consideration, there is some input of size *n* for which the algorithm must perform at least f(n) steps. (lower bound)

(Class of algorithms refers to algorithms that use a specific operation to solve a problem.)

If function *t* and *f* are equal then algorithm A is optimal. If not there may be a better algorithm or a better lower bound.

Example: given a list of n>=1 entries, output the largest entry in the list.

Basic operation: comparison of two list entries.

Class of algorithms: algorithms that can compare and copy list entries, but do no other operations.

Upper bound: assume that we implement the list l using an array.

```
0. max = retrieve(first(l),l);
1. index = next(first(l),l);
2. while (index != end_list(l)) {
3.       if (max < retrieve(index,l))
4.         max = retrieve(index,l)
5.  }
```

where:
first(l) returns the first position in l.
next(x,l) returns the position following position x in list .
retrieve(x,l) returns the list entry at position x in list l.

Comparisons are done in line 3, which is executed exactly n-1 times. Thus, t(n)=n-1 is an upper bound on the number of comparisons.

Lower bound: assume that the entries in the list are distinct (worst-case). In a list with n distinct entries, n-1 entries are *not* the maximum. A particular entry is not the maximum only if it is smaller than at least one other entry in the list. Hence, n-1 entries must be "losers" in comparisons by the algorithm. Each comparison has only one loser, so f(n)=n-1 is a lower bound on the number of comparisons needed.

Conclusion: functions t and f are equal so the algorithm is optimal.

Big-oh notation

We will derive an approximation of the running time of an algorithm, called the asymptotic growth rate. Our approximation will ignore constant factors (e.g. overhead operations) and small inputs, and it will be able to separate algorithms that do drastically different amounts of work for large inputs.

Definition: T(N) is O(f(N)) if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$. (O(f(N)) are the functions that grow no faster than f(N).)

O(f(N)) is read as "big oh of f" or just "oh of f".

Example 1: suppose t(0) = 1, t(1) = 4, and in general $t(n) = (n+1)^2$. Then t(n) is $O(n^2)$, as we let $n_0 = 1$ and c = 4. That is for n>=1, we have $(n+1)^2 <= 4n^2$. (We cannot let $n_0 = 0$, because t(0) = 1 is not less than $c0^2 = 0$ for any constant c.)

Example 2: $t(n) = 3n^3 + 2n^2$ is $O(n^3)$. To see this, let $n_0 = 0$ and c = 5. Then, for n>=0, $3n^3 + 2n^2 <= 5n^3$.

Notice that in example 2, we could also say that this t(n) is $O(n^4)$ but this would be a weaker statement than saying it is $O(n^3)$. We want to find the best upper bound. Also, we do not consider small inputs (less than n_0 in size) and constant factors (c).

Definition: T(N) is Ω(g(N)) ("Omega") if there are positive constants c and n_0 such that $T(N) \geq cg(N)$ when $N \geq n_0$. (Ω(g(N)) are the functions that grow at least as fast as f(N).)

Definition: T(N) = Θ(h(N)) ("Theta") if and only if T(N) = O(h(N)) and T(N) = Ω(h(n)). (Θ(f) is the set of functions that grow at the same rate as f.)

Θ(f(N)) is read as "order f". (If T(N) is a polynomial of degree *k* then $T(N) = \Theta(N^k)$.

Example: our algorithm to find the maximum entry in a list is $\Theta(n)$.

Definition: T(N) is o(p(N)) if there are positive constants c and n_0 such that $T(N) < cf(N)$ when $N \geq n_0$. (T(N) is o(p(N)) if $T(N) = O(p(N))$ and $T(N) \neq \Theta(p(N))$.)

The tyranny of growth rates:

	growth rates		
N	nlgn	n^2	2n
10	30	100	1024
100	600	10,000	big!

N: *linear* growth, optimal for an algorithm that must process n inputs.

Nlgn: when n doubles, the running time more than doubles (but not much more).

N^2: when n doubles the running time increases fourfold. (*quadratic*)

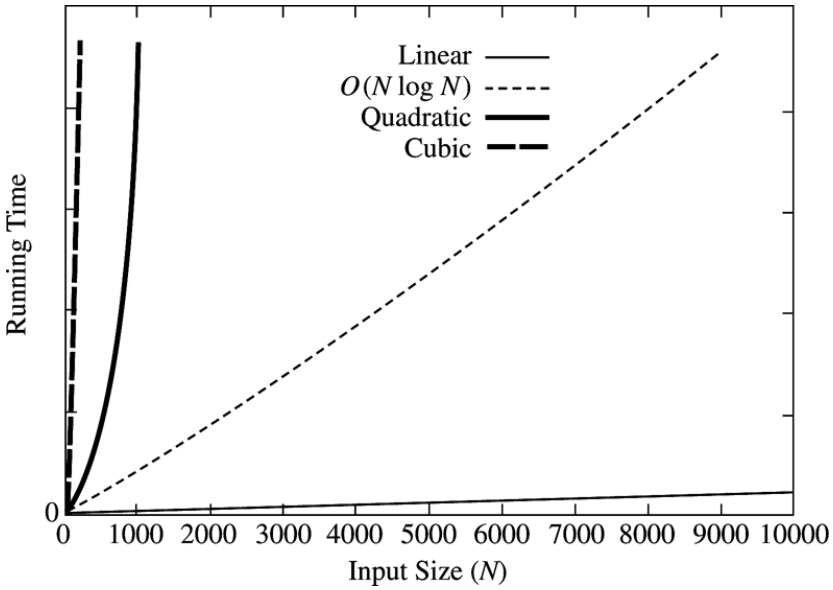
2^n: when n doubles, the running time squares. (*exponential*)

Example: suppose we can afford 1000 seconds (17 min) to solve a given problem. How large a problem can we solve?

Running time	max. problem size for 10^3 seconds	max problem size for 10^4 seconds	increase in size
100n	10	100	10.0
$5n^2$	14	45	3.2
$N^3/2$	12	27	2.3
2^n	10	13	1.3

Suppose we can buy a machine that runs ten times faster at no additional cost. Then we can spend 10^4 seconds on the problem where we spent 10^3 seconds before. What is the maximum size problem we can now solve?

As machines become faster, we desire to solve larger problems. Finding an algorithm with a low growth becomes more rather than less important!



Checking your results

N	CPU time (T)	T/N^2	T/N^3	$T/(N^2 \log N)$
100	022	.002200	.000022000	.0004777
200	056	.001400	.000007000	.0002642
300	118	.001311	.000004370	.0002299
400	207	.001294	.000003234	.0002159
500	318	.001272	.000002544	.0002047
600	466	.001294	.000002157	.0002024
700	644	.001314	.000001877	.0002006
800	846	.001322	.000001652	.0001977
900	1,086	.001341	.000001490	.0001971
1,000	1,362	.001362	.000001362	.0001972
1,500	3,240	.001440	.000000960	.0001969
2,000	5,949	.001482	.000000740	.0001947
4,000	25,720	.001608	.000000402	.0001938

Verify that a program is O(f(N)):

- compute values T(N)/f(N) for a range of values (usually spaced out by a factor of 2)

- if T(N) is a tight answer, computed values will converge to a positive constant

- if T(N) is overestimate (underestimate) values will converge to 0 (diverge).

A few grains of salt about Big-oh

The growth rate O(f(n)) of the worst case running time is not the sole, or necessarily the most important, criterion for evaluating an algorithm:

1. The statement that an algorithm is O(f(n)) does not imply that the algorithm ever takes that long. The actual running time might be lower.

2. The input that causes the worst case might not be encountered in practice.

3. If the program is to be run only on small inputs, the growth rate of the running time may be less important than the constant factor in the formula for running time. (Some algorithms with the best growth rate are never used in practice.)

4. A complicated but efficient algorithm may be hard to maintain.

5. An efficient algorithm may use too much space to be implemented without slow secondary storage, which may more than negate the efficiency.

Example: Sorting

Sorting problem: assume objects to be sorted are records and one of the fields, called the key, is of a type for which a linear- ordering relation <= is defined. The sorting problem is to arrange a sequence of records so that the values of their key fields form a non-decreasing sequence.

Internal sorting: takes place in main memory.

External sorting: number of objects too large to fit in main memory. (The bottleneck is the movement of data between main memory and secondary storage.)

Criteria to evaluate running time:

1) number of steps to sort n records.
2) number of comparisons between keys that must be made to sort n records. (Useful when comparison is expensive e.g. keys are long strings of characters.)
3) number of times the record must be moved (useful if size of records is large).
4) how much extra space the algorithm uses (in addition to the input). If the amount of space is constant with respect to the input size, the algorithm is said to work *in place*.

Bubble sort: on each of several passes, compare keys in adjacent positions and swap them if they are out of order.

```
void bubble (a: array[1..n] of integer) {
// procedure bubble sorts a into increasing order
      int i,j, temp;
(1)  for (int i = 1; i<n; i++)
(2)       for (int j = n; j > i; j--)
(3)           if (a[j-1] > a[j]) {
                  //swap (a[j-1] and a[j]);
(4)               temp = a[j-1];
(5)               a[j-1] = a[j];
(6)               a[j] = temp;
              }
}
```

A. Input size = n.

B. Assignment statements (4), (5), (6) each take O(1) time. Thus, for this group of statements O(max(1,1,1)) = O(1).

Working from the inside out:

C. Testing the condition of the if-statement is O(1). The worst case is that statements (4) - (6) will be executed, thus statements (3) - (6) take O(1) time.

D. Consider the for-loop in (2) - (6). We charge O(1) for each iteration to account for incrementing the index, for testing the limit, and for jumping back to the beginning of the loop. For lines (2) - (6), the body takes O(1) time for each iteration. The number of iterations is n-i, so the time spent in the loop of lines (2) - (6) is O((n-i) x 1) which is O((n-i)).

E. Consider the outer loop. Statement (1) is executed n-1 times, so the total running time of the program is bounded above by some constant times $\sum_{i=1}^{n-1} n - i = n(n-1)/2$, which is $O(n^2)$ and $\Omega(n^2)$. Since there are inputs for which n(n-1)/2 comparisons are done, it is $\Theta(n^2)$.

Recursion

```
public static long factorial(int n) {
    if (n <= 1)
        return 1
    else
        return n * factorial(n-1);
}
```

Just a loop in disguise: O(n)

```
public static long fib(int n) {
    if (n <= 1)
        return 1
    else
        return fib(n-1) + fib(n-2);
}
```

T(0) = T(1) = 1

T(N) = T(N-1) + T(N-2) + 2

For N > 4, fib(N) >= $(3/2)^N$ (Chapter 7 will show how to solve recurrence equaltions.)

Why is recursive Fibonacci so slow?

Binary search

```
1       /**
2        * Performs the standard binary search.
3        * @return index where item is found, or -1 if not found.
4        */
5       public static <AnyType extends Comparable<? super AnyType>>
6       int binarySearch( AnyType [ ] a, AnyType x )
7       {
8           int low = 0, high = a.length - 1;
9
10          while( low <= high )
11          {
12              int mid = ( low + high ) / 2;
13
14              if( a[ mid ].compareTo( x ) < 0 )
15                  low = mid + 1;
16              else if( a[ mid ].compareTo( x ) > 0 )
17                  high = mid - 1;
18              else
19                  return mid;   // Found
20          }
21          return NOT_FOUND;     // NOT_FOUND is defined as -1
22      }
```

Work inside the loop is O(1).

How many times does the loop execute?

Every time through the loop, the number of values left to be searched is at least halved. How many times can you halve the array?

The number of times the loop can iterate is at most $\lceil \log(N-1) \rceil + 2$.

$\Rightarrow$ running time is O(log N)

Find(item): O(logN)

Insert(item): O(n) // maintain sorted order