Maggie Johnson

Handout # 14
CS109

# Analysis of Algorithms I

*Key topics:*

* Introduction
* Generalizing Running Time
* Doing a Timing Analysis
* Big-Oh Notation
* Big-Oh Operations
* Analyzing Some Simple Programs - no subprogram calls
* Worst-Case and Average Case Analysis
* Analyzing Programs with Non-Recursive Subprogram Calls
* Classes of Problems

---

There may be several different ways to solve a particular problem. For example, there are several methods for doing sorts. How can you decide which method is the best in a certain situation? How would you define "best" - is it the fastest method or the one that takes up the least amount of memory space?

Understanding the relative efficiencies of algorithms designed to do the same task is very important in every area of computing. In the 1950's and 60's, many mathematicians and computer scientists developed the field of algorithm analysis. One researcher in particular, Donald Knuth, wrote a three-volume text called The Art of Computer Programming which provided a foundation for the subject.

As mentioned earlier, an algorithm can be analysed in terms of time efficiency or space utilization. We will consider only the former right now. The running time of an algorithm is influenced by several factors:

1) Speed of the machine running the program

2) Language: assembly runs much faster than C which runs faster than Pascal

3) Efficiency of the compiler that created the program

4) The size of the input: processing 1000 records will take more time than processing 10 records.

5) Nature of the input: if the item we are searching for is at the top of the list, it will take less time to find it than if it is at the bottom.

Because of the first three items in the list, we can't use an exact measurement of running time. To say that a particular algorithm written in Visual C++ and running on a Pentium takes 0.0000025 seconds to run tells us nothing about the general time efficiency of the algorithm, because the measurement is specific to a given environment. We need a general metric for the time efficiency of an algorithm; one that is independent of processor or language speeds, or compiler efficiency.

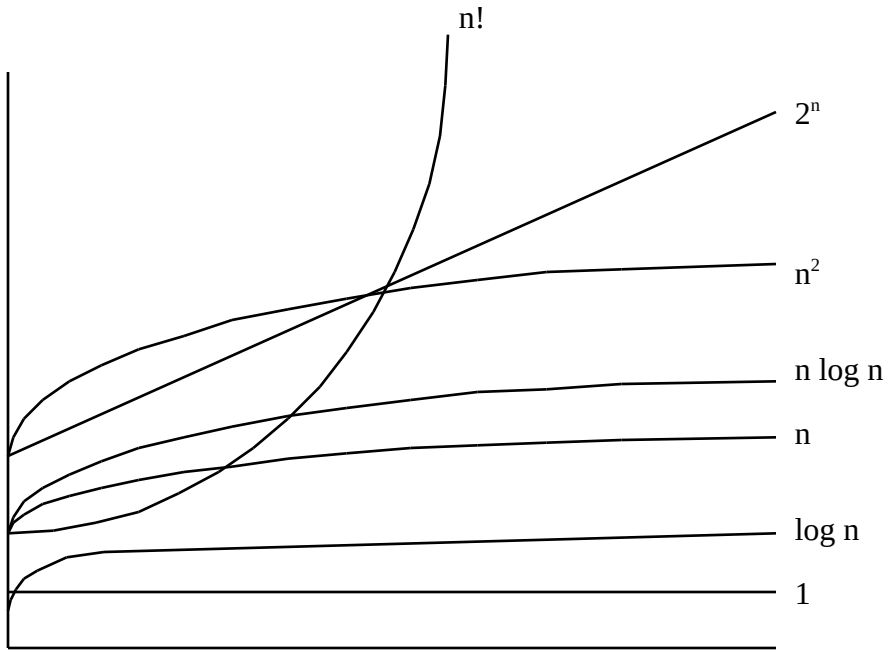
The other complication in doing timing analysis is item 4: the size of the input. As a result of this problem, we usually express the running time of an algorithm as a function of the size of the input.

Thus, if the input size is n, we express the running time as T(n). Finally, the last item in the list requires us to express timing analyses in terms of "worst case", "average case" or "best case".

## Generalizing Running Time

The problem of generalizing a timing analysis is handled by not dealing with exact numbers but instead with order of magnitude or rate of growth. In other words, we want to know how the execution time of an algorithm grows as the input size grows. Do they grow together? Does one grow more quickly than the other - how much more? The ideal situation is one where the running time grows very slowly as you add more input. So, rather than deal with exact values, we keep it general by comparing the growth of the running time as the input grows, to the growth of known functions. The following functions are the ones typically used:

| input size n | (1) | log n | n | n log n | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|---|---|
| 5 | 1 | 3 | 5 | 15 | 25 | 125 | 32 |
| 10 | 1 | 4 | 10 | 33 | 100 | $10^3$ | $10^3$ |
| 100 | 1 | 7 | 100 | 664 | $10^4$ | $10^6$ | $10^{30}$ |
| 1000 | 1 | 10 | 1000 | $10^4$ | $10^6$ | $10^9$ | $10^{300}$ |
| 10000 | 1 | 13 | 10000 | $10^5$ | $10^8$ | $10^{12}$ | $10^{3000}$ |




For small inputs, the difference in running times would hardly be noticeable on a fast computer. For inputs of 100,000, however, (if we assume one microsecond per instruction) an n log n algorithm will take about 1.7 CPU seconds, an $n^2$ algorithm will take about 2.8 CPU hours which is unacceptable, and an $n^3$ algorithm would take 31.7 CPU *years*. There is no way we would ever use such an algorithm to deal with large inputs.

## Doing a Timing Analysis

T(n) or the running time of a particular algorithm on input of size n is taken to be the number of times the instructions in the algorithm are executed. As an illustration, consider a pseudocode algorithm which calculates the mean of a set of n numbers:

```
1. Read n
2. Sum = 0
3. i = 1
4. while i <= n
5.       read number
6.       add number to sum
7.       increment i
8. mean = sum / n
```

Statements 1, 2, & 3 are each executed once. Statements 5, 6, 7 are each executed n times. Statement 4 which controls the loop is executed n + 1 times (one additional check is required), and statement 8 is executed once. This is summarized below:

| Statement | Number of times executed |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 3 | 1 |
| 4 | n+1 |
| 5 | n |
| 6 | n |
| 7 | n |
| 8 | 1 |

Thus, the computing time for this algorithm in terms of input size n is: $T(n) = 4n + 5$. We can see intuitively, that as n grows, the value of this expression grows linearly. We say T(n) has an "order of magnitude (or rate of growth) of n". We usually denote this using big-oh notation: $T(n) = O(n)$, and we say that the algorithm has a **complexity** of O(n). But, intuition is not enough for us skeptical computer science types - we must be able to show mathematically which of the standard functions given in the table above indicates the correct rate of growth.

## Big-Oh Notation

**Definition 1**: Let f(n) and g(n) be two functions. We write:

$$f(n) = O(g(n)) \quad \text{or} \quad f = O(g)$$

(read "f of n is big oh of g of n" or "f is big oh of g") if there is a positive integer C such that $f(n) <= C * g(n)$ for all positive integers n.

The basic idea of big-Oh notation is this: Suppose f and g are both real-valued functions of a real variable x. If, for large values of x, the graph of f lies closer to the horizantal axis than the graph of some multiple of g, then f is of order g, i.e., $f(x) = O(g(x))$. So, g(x) represents an upper bound on f(x).

*Example 1*

Suppose f(n) = 5n and g(n) = n. To show that f = O(g), we have to show the existence of a constant C as given in Definition 1. Clearly 5 is such a constant so f(n) = 5 * g(n). We could choose a larger C such as 6, because the definition states that f(n) must be less than or equal to C * g(n). Therefore, a constant C exists (we only need one) and f = O(g).

*Example 2*

In the previous timing analysis, we ended up with T(n) = 4n+5, and we concluded intuitively that T(n) = O(n) because the running time grows linearly as n grows. Now, however, we can prove it mathematically:

To show that f(n) = 4n + 5 = O(n), we need to produce a constant C such that f(n) <= C * n for all n. If we try C = 4, this doesn't work because 4n + 5 is not less than 4n. We need C to be at least 9 to cover *all* n. If n = 1, C has to be 9, but C can be smaller for greater values of n (if n = 100, C can be 5 - `why is this?`). Since the chosen C must work for all n, we look at the largest required value for C:

$$4n + 5 <= 4n + 5n = 9n$$

Since we have produced a constant C that works for all n we can conclude T(4n+5) = O(n).

*Example 3*

$f(n) = n^2$: We will prove that f(n) <> O(n). To do this, we must show that there cannot exist a constant C that satisfies the big-oh definition. We will prove this by contradiction. Suppose there is a constant C that works; then, by the definition: $n^2 <= C * n$ for **all** n. Suppose n is any positive real number greater than C, then: $n * n > C * n$ or $n^2 > C * n$. So there exists a real number n such that $n^2 > C * n$. This contradicts the supposition, so the supposition is false. There is no C that can work for all n: f(n) <> O(n) when $f(n) = n^2$

*Example 4*

Suppose $f(n) = n^2 + 3n - 1$. We want to show the $f(n) = O(n^2)$.

$$f(n) = n^2 + 3n - 1$$

$$< n^2 + 3n \quad \text{(subtraction makes things smaller so drop it)}$$

$$<= n^2 + 3n^2 \quad \text{(since } n <= n^2 \text{ for integer n)}$$

$$= 4n^2$$

Therefore, if C = 4, we have shown that $f(n) = O(n^2)$. Notice that all we are doing is finding a simple function that is an upper bound on the original function. Because of this, we could also say that $f(n) = O(n^3)$ since $(n^3)$ is an upper bound on $n^2$. This would be a much weaker description, but it is still valid.

*Example 5*

Show $f(n) = 2n^7 - 6n^5 + 10n^2 - 5 = O(n^7)$

$f(n) < 2n^7 + 6n^5 + 10n^2$

$<= 2n^7 + 6n^7 + 10n^7$

$= 18n^7$ thus C = 18 and we have shown the $f(n) = O(n^7)$

You are probably noticing a pattern here. Any polynomial is big-oh of its term of highest degree. We are also ignoring constants. A proof of this is very straightforward. Any polynomial (including a general one) can be manipulated to satisfy the big-oh definition by doing what we did in the last example: take the absolute value of each coefficient (this can only increase the function); Then since

$$n^j <= n^d \quad \text{if } j <= d$$

we can change the exponents of all the terms to the highest degree (the original function must be less than this too). Finally, we add these terms together to get the largest constant C we need to find a function that is an upper bound on the original one.

Big-Oh, therefore is a useful method for characterizing an approximation of the upper bound running time of an algorithm. By ignoring constants and lower degree terms, we end up with the approximate upper bound. In many cases, this is sufficient. Sometimes, however, it is insufficient for comparing the running times of two algorithms. For example, if one algorithm runs in O(n) time and another runs in $O(n^2)$ time, you cannot be sure which one is fastest for large n. Presumably the first is, but perhaps the second algorithm was not analysed very carefully. It is important to remember that Big-Oh notation gives no information on how good an algorithm is, it just gives an upper bound on how bad it can be.

There is another notation that is sometimes useful in the analysis of algorithms. To specify a lower bound on the growth rate of T(n), we can use Big-Omega notation. If the lower bound running time for an algorithm is g(n) then we say $T(n) = \Omega(n)$. As an example, any algorithm with m inputs and n outputs would require at least $\Omega(m+n)$ work.

One last variation on this theme is Big-Theta notation. Big-$\Theta$ bounds a function from both above and below, so two constants must be provided rather than one, in formal proofs of the bounds of a given function.

There is one adjustment that we must make to our definition of Big-Oh. You may have noticed that we have been avoiding logarithms in the discussion so far. We cannot avoid them for long, however, because many algorithms have a rate of growth that matches logarithmic functions (binary search, mergesort, quicksort). As a brief review, recall that $\log_2 n$ is the number of times we have to divide n by 2 to get 1; or alternatively, the number of 2's we must multiply together to get n:

$$n = 2^k \quad <-> \quad \log_2 n = k$$

Many "Divide and Conquer" algorithms solve a problem by dividing it into 2 smaller problems, then into 2 even smaller problems; you keep dividing until you get to the point where solving the problem is trivial. This division by 2 suggests a logarithmic running time.

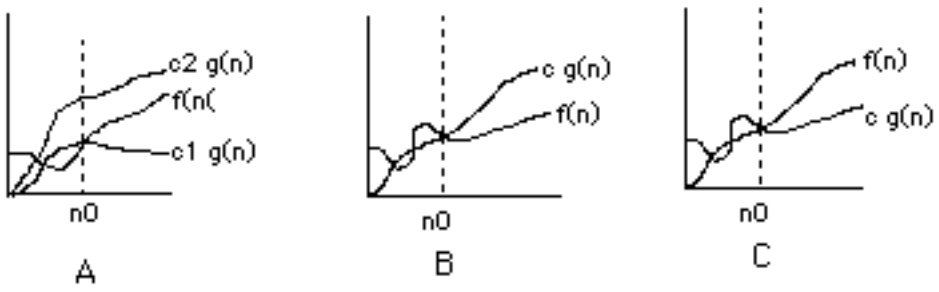
Thus, since log(1) = 0, there is no constant C such that 1 <= C * log(n) for all n. For n >= 2, it is the case that 1 <= log(n) and so the constant C = 1 works for sufficiently large n (larger than 1). This suggests that we need a stronger definition of Big-Oh.

**Definition 2**: Let f(n) and g(n) be two functions. We write:

f(n) = O(g(n)) or f = O(g)

if there is a positive integer C and N such that f(n) <= C * g(n) for all integers n >= N.

With this more general definition, we can say that 1 = O(log(n)) since C = 1 and N = 2 will work. Also, with this definition, we can show very clearly the difference between the three types of notation:




In all three, n0 is the minimal possible value, but any greater value would work. (A) shows Big-Θ which bounds a function between two constant factors. We can write f(n) = Θ(g(n)) if there exist positive constants n0, c1 and c2 such that to the right of n0, the value of f(n) always lies between c1 * g(n) and c2 * g(n) (inclusive). (B) shows Big-Oh which gives an upper bound for a function to within a constant factor. We can write f(n) = O(g(n)) if there exist positive constants n0 and c such that to the right of n0, f(n) always lies on or below c * g(n). Finally, (C) shows Big-Ω which gives a lower bound for a function to within a constant factor. We can write f(n) = Ω (g(n)) if there exist positive constants n0 and c such that to the right of n0, f(n) always lies on or above c * g(n).

### Big-Oh Operations

**Summation Rule**

Suppose T1(n) = O(f1(n)) and T2(n) = O(f2(n)). Further, suppose that f2 grows no faster than f1, i.e., f2(n) = O(f1(n)). Then, we conclude that T1(n) + T2(n) = O(f1(n)). In other words, O(f1(n) + f2(n)) = O(max(f1(n), f2(n))).

Proof: Suppose that C and C' are constants such that T1(n) <= C * f1(n) and T2(n) <= C' * f2(n). Let D = the larger of C and C'. Then,

T1(n) + T2(n) <= C * f1(n) + C' * f2(n)

<= D * f1(n) + D * f2(n)
<= D * (f1(n) + f2(n))
<= O(f1(n) + f2(n))

Further, we have shown that the polynomial of highest degree (of f1(n) and f2(n)) will determine the value of big-oh.

**Product Rule**

Suppose T1(n) = O(f1(n)) and T2(n) = O(f2(n)). Then, we conclude that T1(n) * T2(n) = O(f1(n) * Of2(n)).

This can be proven using a similar strategy as the Summation Rule proof.

### Analyzing Some Simple Programs - no subprogram calls

General Rules:

1) All basic statements (assignments, reads, writes, conditional testing) run in constant time: O(1).

2) The time to execute a loop is the sum, over all times around the loop, of the time to execute all the statements in the loop, plus the time to evaluate the condition for termination which is O(1).

3) The complexity of an algorithm is determined by the complexity of the most frequently executed statements. If one set of statements have a running time of $O(n^3)$ and the rest O(n), then the complexity of the algorithm is $O(n^3)$. This is a result of the Summation Rule.

*Example 6*

Compute the number of instructions executed for the following code segment:

```
for (i = 2; i < n; i ++)
        sum += i;
```

The number of iterations of a for-loop is equal to the top index of the loop minus the bottom index, plus one more to account for the conditional test. (Note: if the for-loop terminating condition is i <=n then the number of iterations is (top index+1 minus bottom index) + 1). In this case, we have n - 2 + 1 = n-1. The assignment in the loop is executed n-2 times. So, we have (n-1) + (n-2) = (2n-3) instructions executed = O(n).

Complexity problems may ask for "number of instructions executed" which means you need to provide an equation in terms of n. Or, we may just ask for the complexity in which case you need only provide a big-oh expression.

*Example 7*

Consider the inner loop of selection sort. Find the number of instructions executed and the complexity of this algorithm?

```
1)      for (i = 1; i < n; i ++) {
2)              SmallPos = i;
3)              Smallest = Array[SmallPos];
4)              for (j = i+1; j <=n; j++)
5)                      if (Array[j] < Smallest) {
6)                              SmallPos = j;
7)                              Smallest = Array[SmallPos]
                        }
8)              Array[SmallPos] = Array[i];
9)              Array[i] = Smallest;    }
```

Statement 1 is executed n times (n -1+1); statements 2, 3, 8, 9 (each representing O(1) time) are executed n - 1 times, once on each pass through the outer loop. On the first pass through this loop with i = 1, statement 4 is executed n times; statement 5 is executed n-1 times, and assuming a worst case where the elements of the array are in descending order, statements 6 and 7 (each O(1) time) are executed n-1 times.

On the second pass with i = 2, statement 4 is executed n-1 times and 5, 6, 7 are executed n-2 times, etc. Thus, statement 4 is executed (n) + (n-1) +... + 2 times and statements 5, 6, 7 are executed (n-1) + (n-2) + ... + 2 + 1 times. The first sum = n(n+1)/2, and the second is equal to n(n-1)/2.

Thus, the total computing time is:

$$
\begin{aligned}
T(n) &= (n) + 4(n-1) + n(n+1)/2 + 3[n(n-1) / 2] \\
&= n + 4n - 4 + (n^2 + n)/2 + (3n^2 - 3n) / 2 \\
&= 5n - 4 + (4n^2 - 2n) / 2 \\
&= 5n - 4 + 2n^2 - n \\
&= 2n^2 + 4n - 4 = O(n^2)
\end{aligned}
$$

*Example 8*

The following program segment makes array A an n x n identity matrix (A x M = M x A = M for any n x n matrix M). What is the complexity of this code?

```
1)      scanf("%d ", &n);
2)      for (i = 1; i <= n; i ++)
3)              for (j = 1; j <= n; j ++)
4)                      A[i,j] = 0;
5)      for (i = 1; i <= n; i ++)
6)              A[i,i] = 1;
```

A program can be analyzed in parts, and then we can use the summation rule to find a total running time for the entire program. Line 1 takes O(1) time, We go around the loop of lines 5 & 6 n times giving us a total of O(n) time spent in that loop. We go around the loop of lines 3 & 4 n times. We go around the outer loop of line 2 n times for a total of $O(n^2)$.

Thus the running time of the segment is $O(1) + O(n^2) + O(n)$. We apply the summation rule to conclude that the running time of the segment is $O(n^2)$.

## Worst-Case and Average Case Analysis

Thus far, we have been analyzing programs considering an input of size n, i.e., the maximum number of input items. This is worst-case analysis; we are looking at the largest input size possible and determine the running time from that. There may be situations where you want to know how long the algorithm runs with an average number of inputs. Average-case analysis is much more involved than worst-case.

What you have to do is determine the probability of encountering each input size for a given problem. How often will the input size be 10 or 1000? You also have to determine the distributions of the various data values. How often will the item we are looking for be in the first place we look? Using these probabilities, you can compute an average-case analysis, but it is totally dependent on the assumptions that you have made to set the probabilities. That is always the hardest part of average case analysis. Because of these complications, worst-case analysis is far more common.

Just to give you a sense of how average-case analysis works, we will look at an example.

*Example 9*

Here is a simple linear search algorithm which returns the index location of a value in an array.

```
/* a is the array of size n we are searching through */
i = 0;
while (i < n and x != a_i)
        i = i + 1;
if (i <= n)
        location = i;
else
        location = 0;
```

We will do an average-case analysis with the assumption that an element x is in the array. There are n types of possible inputs when x is known to be in the array. (We'll consider just the execution of the while loop.) If x is the first element of the array, the while loop condition will execute and fail, so only one line of the while loop is executed. If x is the second element of the array, the while loop condition will execute as well as the increment statement. Then we have to check the while loop condition again for a total of 3 lines. In general, if x is the ith term of the array, 2i-1 lines are executed. So, the average number of lines equals:

$$1 + 3 + 5 + ... + (2n - 1) \ / \ n = 2 (1 + 2 + 3 + ... + n) - n \ / \ n$$

We know that $1 + 2 + 3 + ... + n = n ( n + 1) / 2$, so the average number of lines executed is:

$$2 [n ( n + 1) / 2] - n / n = n = O(n)$$

Notice we are also assuming that x is equally likely to be at any position in the array.

## Analyzing Programs with Non-Recursive Subprogram Calls

The first step is to analyze each individual procedure to determine its running time. We begin with the procedures that do not call any other procedures. Then, we evaluate the running times of the procedures that call previously evaluated procedures. Suppose we have determined the running time of procedure P to be O(f(n)). Therefore, any statement anywhere in the program that calls P also has a running time of O(f(n)). We just include this time in our analysis of any section of the program where P is called.

Functions are similar, but a little more complex because they can appear as values in statements and conditions. For an assignment or printf with function calls, we determine the running time by using the summation rule to add the running times of all the function calls. If a function call with running time O(f(n)) appears in a condition or for/while loop, its running time is accounted for as follows:

1) while/repeat: add f(n) to the running time for each iteration. We then multiply that time by the number of iterations. For a while loop, we must add one additional f(n) for the final loop test.
2) for loop: if the function call is in the initialization of a for loop, add f(n) to the total running time of the loop. If the function call is the termination condition of the for loop, add f(n) for each iteration except the first.
3) if statement: add f(n) to the running time of the statement.

To illustrate these points, look at the following program:

```
int a, n, x;

int foo(int x, int n) {
        int i;

1)              for (i = 1; i < n; i++)
2)                      x = x + bar(i, n);
3)              return x;
        }

int bar(int x, int n) {
        int i;

4)              for (i = 1; i < n; i++)
5)                      x = x + i;
6)              return x;
        }

void main(void) {
7)      GetInteger(n);
8)      a = 0;
9)      x = foo(a, n)
10)     printf("%d", bar(a, n))
}
```

We begin analyzing bar since it does not call any other subprograms. The for loop adds each of the integers 1 through n to x. Thus, the *value* of bar(x, n) = x + n(n+1)/2. The running time is analyzed as follows: Lines 3 & 4 iterate n times. Line 6 is executed once. Therefore, the running

time for bar is O(n) + O(1) = O(n). Next, we look at foo. Line 2 would normally take O(1) time but it has a call to bar, so the time for line 5 is O(1) + O(n) = O(n). The for loop of lines 1 and 2 iterates n times, so we multiply O(n) for the body of the loop times the number of iterations (n) to get $O(n^2)$ which is the running time for a call to function foo.

Finally, we look at main. Lines 7 and 8 each take O(1). The call to foo on line 9 takes $O(n^2)$. The printf of line 10 takes O(1) (it probably takes more than this since it is also a function call…) plus a call to bar which gives us O(1)+O(n). Thus, the total running time for main is O(1) + O(1) + $O(n^2)$ + O(n) = $O(n^2)$.

Here is the body of a function:

```
sum = 0;
for (i = 1; i <= f(n); i++)
      sum += i;
```

where f(n) is a function call. Give a big-oh upper bound on this function if the running time of f(n) is O(n), and the value of f(n) is n!:

## Classes of Problems

Imagine that you have decided to ski across the Sierras over winter break. Since there are no McDonalds between Yosemite and Lee Vining, you will need to bring all your food with you in your pack. On this trip, taste is a secondary consideration. What matters is how much your food weighs and how many calories it contains. After raiding your kitchen, you discover that you have roughly 200 different food items. Each item has a certain weight and contains a certain number of calories:

| | | | |
|---|---|---|---|
| 1. | Snickers Bar | 200 calories | 100 grams |
| 2. | Diet Coke | 1 calorie | 200 grams |
| … | | | |
| 200. | Dry Spaghetti | 500 calories | 450 grams |

Given this selection of food, your task is to find the best subset that maximizes the number of calories, but falls within the weight limit you can carry in your pack (say 15kg). One algorithm that definitely produces a solution is to simply try every possible combination. Unfortunately, there are $2^{200}$ such combinations, and the Sierras would erode away before that algorithm terminates.

Is there another algorithm that runs in reasonable (*i.e.*, polynomial) time? If there is, no one has found it yet. This might make you suspect that no polynomial time solution exists, yet no one has ever been able to prove that either. This problem, known as the Knapsack Problem, is an example of a general class of problems called the **NP-complete** problems. These are problems for which no polynomial time solution exists, but it is unproven whether the problems necessarily require exponential time.

*Problems, Problems, Problems...*

In studying algorithms, we have seen functions and big-O notation used to characterize the running time of algorithms. There are some important terms used to classify problems, based on the running time of the algorithms that solve those problems. First, we review the different categories of functions for big-O notation:

The ones that involve n (n! or $n^n$) as an exponent are called **exponential** functions. Functions whose growth is <= $n^c$ for some constant c are **polynomial** functions. **Linear** functions have growth proportional to n. **Sub-linear** functions have growth proportional to log n. A **constant time** function has growth independent of n.

These descriptions of running times are frequently used to characterize different sets of problems. The general categories are:

**Intractable:** These are problems that require exponential time. Moreover, it has been proven that a solution in less than exponential time is impossible. For example, consider the problem of listing all possible committees of size one or more that could be formed from a group of n people. There are $2^n$ - 1 such committees so any algorithm that solves this problem must have at least $2^n$ - 1 steps, and thus a running time proportional to $2^n$. In addition, this problem does not have a polynomial solution.

**Polynomial:** These are problems for which sub-linear, linear or polynomial solutions exist. Insertion sort, quicksort, mergesort and other sorting algorithms are good examples, as well as finding the smallest or largest values in a list. Any algorithm whose running time grows proportional to a polynomial function can solve reasonable-size problems if the constant in the exponent is small.

**NP-Complete:** No polynomial solution has been found, although exponential solutions exist. However, it has *not* been proven that a polynomial solution could not be found.

Some famous examples:

a) Generalized Instance Insanity: Given n cubes, where each face of the cube is painted in one of n colors, is it possible to stack the cubes so that each of the n colors appears exactly once on each side of the column?

b) Traveling Salesperson: Given a map of cities and a cost of travel for each pair of cities, is it possible to visit each city exactly once and return home for less than k dollars?

c) n-bookshelf packing: You are given k books to put on n shelves. Will they fit? (the books are all of different sizes). This is another way of stating the Knapsack problem.

This group of problems is very interesting for a number of reasons. First of all, the "NP" means "nondeterministic polynomial". Nondeterministic does not mean that we have not yet been able to determine if the problem is polynomial. It means that the algorithm that we might define to solve one of these problems is nondeterminisitic, i.e., there is an element of chance in the algorithm. For example, we might write a solution as follows for the Traveling Salesperson problem:

Choose one of the possible paths, and compute its total cost.
If this cost is no greater than the allowable cost,

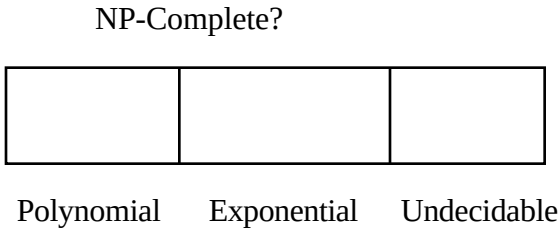
then declare a success
else declare nothing.

Someone must choose one of the possible paths (this is the nondeterministic part). If we choose the right one, the problem is solved quickly; if not, we learn nothing. We define the running time of an NP algorithm as the time required to execute the algorithm if we make the "right" choice, meaning the choice that would lead to a solution in the optimal amount of time. So we are trying to measure how good the algorithm is (not how good our choices are).

According to this definition, the Traveling Salesperson problem has a polynomial solution. If we make the right choice, we just have to compute the distance for n cities which is proportional to n.

Of course, the class of NP problems includes the truly polynomial problems. If we can design a deterministic (no choices) polynomial solution for a problem, then we surely can design a nondeterministic one. In addition, the Traveling Salesperson problem suggests that there might be NP problems that do not have polynomial solutions. In other words, there appear to be problems that can be solved in polynomial time by nondeterministic algorithms, but that might require exponential time by a deterministic algorithm. But no one has been able to prove this. In particular, no one has been able to show that there does not exist a polynomial solution to the Traveling Salesperson problem. This is still one of the important open questions in theoretical computer science.

So we have the following problem classification:




Let's look at some important NP-complete problems in more detail to see if we can better understand the common characteristics of these problems.

*Two Famous Problems*

1. Satisfiability: The problem is to determine whether any given logical formula is satisfiable, i.e., whether there exists a way of assigning TRUE and FALSE to its variables so that the result is TRUE.

Given a formula

- composed of variables a, b, c, ... and their logical complements, ~a, ~b, ~c...
- represented as a series of clauses, in which each clause is the logical OR (v) of variables and their logical complements
- expressed as the logical AND (^) of the clauses

Is there a way to assign values to the variables so that the value of the formula is TRUE? If there exists such an assignment, the formula is said to be satisfiable.

Is: (a) ^ (b v c) ^ (~c v ~a) satisfiable? _____
Is: (a) ^ (b v c) ^ (~c v ~a) ^ (~b) satisfiable? _____

2. Knapsack: The name of the problem refers to packing items into a knapsack. Is there a way to select some of the items to be packed such that their "sum" (the amount of space they take up) exactly equals the knapsack capacity? We can express the problem as a case of adding integers. Given a set of nonnegative integers and a target value, is there a subset of integers whose sum equals the target value?

Formally, given a set {a1, a2, ..., an} and a target sum T, where ai >= 0, is there a selection vector V = [v1, v2, ..., vn] each of whose elements is 0 or 1, such that

$$\sum_{i=1}^{n} (ai * vi) = T$$

For example, the set might be {4, 7, 1, 12, 10}. A solution for target sum T = 17 exists when V = [1, 0, 1, 1, 0] since 4 + 1 + 12 = 17. No solution is possible if T = 25.

*Characteristics of NP-Complete Problems*

The preceding examples are reasonable representatives of the class of NP-Complete problems. They have the following characteristics:

- Each problem is solvable, and a relatively simple approach solves it (although the approach may be very time-consuming). For each of them, we can simply enumerate all the possibilities, all ways of assigning the logical values of n variables, all subsets of the set S, all subsets of vertices in graph G. If there is a solution, it will appear in the enumeration; if not, we will discover this as well.
- There are $2^n$ cases to consider if we use the enumeration approach. Each possibility can be tested in a relatively small amount of time, so the time to test all possibilities and answer "yes" or "no" is proportional to $2^n$.
- The problems are apparently unrelated coming from logic, number theory, graph theory, etc.
- If it were possible to guess perfectly, we could solve each problem in a very small amount of time. The verification process could be done in polynomial time, given a good guess.

## Bibliography

A. Aho, J. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, MA: Addison-Wesley, 1974.

G. Brassard, P. Bratley, *Algorithmics: Theory and Practice*, Englewood Cliffs, NJ: Prentice Hall, 1988.

T. Cormen, C. Leiserson, R. Rivest, *Introduction to Algorithms*, New York: McGraw-Hill, 1991.

M. Garey, D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, San Francisco, CA: H. Freeman.

J. Hartmanis, R. Stearns, "On the Computational Complexity of Algorithms," *Transactions of the American Mathematical Society*, Vol. 117, 1965.

D. Knuth, *Fundamental Algorithms, volume 1 of The Art of Programming, 2nd ed.*, Menlo Park, CA: Addison-Wesley, 1973.

_____, *Seminumerical Algorithms volume 2 of The Art of Programming, 2nd ed.*, Menlo Park, CA: Addison-Wesley, 1981.

_____, *Sorting and Searching, volume 3 of The Art of Programming*, Menlo Park, CA: Addison-Wesley, 1973.

U. Manber, *Introduction to Algorithms: A Creative Approach*, Reading, MA: Addison-Wesley, 1989.

R. Sedgewick, *Algorithms, 2nd ed.*, Reading, MA: Addison-Wesley, 1989.

**Historical Notes**

The origin of Big-Oh notation is attributed to Paul Bachmann (1837-1920) in his number theory text written in 1892. The actual O symbol is sometimes called a Landau symbol after Edmund Landau (1877-1938) who used this notation throughout his work. The study of the running time of programs and the computational complexity of problems was pioneered by Hartmanis and Stearns (see above). Donald Knuth's series of books established the study of running time of algorithms as an essential ingredient of computer science. The class of polynomial problems was introduced in 1964 by Alan Cobham ("The Intrinsic Computational Difficulty of Functions" in *Proceedings of the 1964 Congress for Logic, Methodology, and the Philosophy of Science)*, and, independently by Jack Edmonds in 1965 ("Paths, Trees and Flowers" in *Canadian Journal of Mathematics*, 17), who also introduced the NP class and conjectured that $P \neq NP$.