Python Intermediate Programming

Unit 4: Algorithmic Study

CHAPTER 10: ANALYSIS OF ALGORITHMS AND COMPLEXITY

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Analysis of Algorithms is a mathematical area of Computer Science in which we analyze the resources (mostly time; but sometimes space; and at the chip level, power) used by algorithms to solve problems.
- An algorithm is a precise procedure for solving a problem, written in any notation that humans understand (and thus can carry-out the algorithm): if we write an algorithm as code in some programming language, then a computer can execute it too.

Objectives

- The main tool that we use to analyze algorithms is big-O notation: it means "growth (in resources, based on the problem size) on the order of". We use big-O notation to characterize the performance of an algorithm by placing it in a complexity class (most often based on its WORST-CASE behavior -but sometimes on its AVERAGE-CASE behavior) when solving a problem of size N: we will learn how to characterize the size of problem, which is most often as simple as N is the number of values in a list/set/dictionary being processed.
- Once we know the complexity class of an algorithm, we have a good handle on understanding its actual performance (within certain limits). Thus, in AA we don't necessarily compute the exact resources needed, but typically an approximate upper bound on the resources, based on the problem size.

Getting to Big-O Notation

LECTURE 1

Getting to Big-O Notation: Throwing away Irrelevant Details

- Here is one simple Python function for computing the maximum of a list (or returning None if there are no values in the list).

```
def maximum(alist):
    answer = None if alist == [] else alist[0]
    for i in range(1,len(alist)):
        if alist[i] > answer
            answer = alist[i]
    return answer
```

- Often, the problem size is the number of values processed: e.g., the number of values in a list or lines in a file. But we can use other metrics as well: it can be the count of number of digits in an integer value, when looking at the complexity of multiplication based on the size of the numbers. Thus, there is no single measure of size that fits all problems: instead, for each problem we try to choose a measure that makes sense and is natural for that problem.

Getting to Big-O Notation:

- Python translates functions like maximum into a sequence of instructions that the computer executes (the subject of one of the last week's lectures). To solve a problem, the computer always executes an integral number of instructions. For simplicity, we will assume that all instructions take the same amount of time to execute. So, to compute the amount of time it takes to solve a problem is equivalent to computing how many instructions the computer must execute to solve it (which we can divide by the number of instructions/second a machine executes, to compute the actual amount of time taken on that machine).

Getting to Big-O Notation:

- Again, we typically look at the worst-case behavior of algorithms. For maximum the worst case occurs if the list is in increasing order. In this case, each new value examined in the list will be bigger than the previous maximum, so the if statement's condition will always be True, which always requires updating answer. If any value was lower, it wouldn't have to update answer and thus take fewer instructions/less time to execute.
- It turns out that for a list of N values, the computer executes 14N + 9 instructions in the worst case for this function. You need to know more CS than you do at this time to determine this formula, but you will get there by ICS 51 (and I expect to cover the basics during the last week of the quarter, when I cover the Python Virtual Machine: its equivalent of machine code).

Getting to Big-O Notation:

- A simple way to think about this formula is that there are 14 computer instructions that are executed each time Python executes the body of the for loop and 9 instructions that are executed only once: they deal with starting and terminating the loop and the entire function. We can write I(N) = 14N + 9 for the worst case of the maximum function, where I(N) is the number of instructions the computer executes when solving a problem on a list with N values. Or to be more specific we would write Imaximum(N) = 14N + 9.
- I would like to argue now that if simplify this function to just I(N) = 14N we have lost some information, but not much. Specifically, as N gets bigger (i.e., we are dealing with very big problems - the kinds computers are used to solve), 14N and 14N+9 are relatively close. Let's look at the result of this function vs. the original as N gets for values of N increasing by a factor of 10.

N	14N+9	14N	error: (14N+9 - 14N)/(14N+9) as a % of N
1	23	14	61% or 39% accurate
10	149	140	6% 94% accurate
100	1409	1400	.6% 99.4% accurate
1000	14009	14000	.06% 99.94% accurate
…			
1,000,000	14,000,009	14,000,000	.00006% 99.99994% accurate
…			

Getting to Big-O Notation:

- Each line shows the % error of computing 14N when the true answer is 14N + 9. So, by the time we are processing a list of 1,000 values, using the formula 14N instead of 14N+9 is 99.94% accurate. For computers solving real problems, a list of 1,000 values is small: a list of millions is more normal. For 1,000,000 values 14N is off by just 9 parts in 14 million. So the 9 doesn't affect the value of the formula much.
- Analysis of Algorithms really should be referred to as ASYMPTOTIC Analysis of Algorithms, as it is mostly concerned with the performance of algorithms as the problem size gets very big (N -> infinity). We see here that as N->Infinity 14N is a better and better approximation to 14N+9: dropping the extra 9 becomes less and less important.

Getting to Big-O Notation:

- Here is a simple Python function for sorting a list of values. This is much simpler than the actual sort method in Python (and the simplicity of code results in this function taking much more time, but it is a good starting point for understanding sorting now; ICS-46 spends a week studying sorting). If you are interested in how this function accomplishes sorting, hand simulate it working on a list of 5-10 values (try increasing values, decreasing values, and values in random order): basically, each execution of the outer loops mutates the list so that the next value in the list is in the correct position.

Getting to Big-O Notation:

```
def sort(alist):
    for base in range(len(alist)):
        for check in range(base+1,len(alist)):
            if alist[base] > alist[check]:
                alist[base],alist[check]= alist[check],alist[base]
    return None  # list is mutated
```

- It turns out that for a list of N values, the computer executes 8N**2 + 12N + 6 instructions in the worst case for this function. The outer loop executes N times (N is len(alist)) and inner loop on average executes N/2 times, so the if statement in the inner loop is executed a quadratic number of times. We can say I(N) = 8N**2 + 12N + 6 for the worst case of the sort function, where I(N) is again the number of instructions the computer executes. Or to be more specific we would write Isort(N) = 8N**2 + 12N + 6.

- I would like to argue in the same way that if simplify this function to just I(N) = 8N**2, we have not lost much information. Let's look at the result of this this function vs. the original as N gets bigger and bigger

N	8N**2+12N+6	8N**2	error: (12N+6)/(8N**2+12N+6) as a % of N
1	26	8	70% or 30% accurate
10	926	800	14% 86% accurate
100	81,206	80,000	1.5% 98.5% accurate
1000	8,012,006	8,000,000	.15% 99.85% accurate

- So, by the time we are processing a list of 1,000 values, using the formula 8N**2 instead of 8N**2 + 12N + 6 is 99.85% accurate. For 1,000,000 values (10**6), 8N**2 is 8*10^12 while 8N**2 + 12N + 6 is 8*10^12 +12*10^6 + 6; the simpler formula is 99.999985% accurate.

Getting to Big-O Notation:

- **CONCLUSION (though not proven):** If the real formula I(N) is a sum of a bunch of terms, we can drop any term that doesn't grow as quickly as the most quickly growing term. In computing the maximum, the linear term 14N grows more quickly than the next term, the constant 9, which doesn't grow at all (as N grows) so we drop the 9 term. In sorting, the quadratic term 8N**2 grows more quickly than the next two terms, the linear term 12N and the constant 6, so we drop the 12N and 6 terms.
- In fact, we can prove that the Limit as N-> of 12N/8N**2 = 3/(2N) -> 0, which means we can discard the 12N term as growing more slowly than the 8N**2 term.

Getting to Big-O Notation:

- The result is a simple function that is still an accurate approximation of the number of computer instructions executed for lists of various LARGE sizes. It consists of a constant multiplied by some function of N (here N and N**2, but many other functions are possible too).
- We now will explain another rationale for dropping the constant in front of N and N**2, and classifying these algorithms as O(N) growing at a linear rate and O(N**2) growing at a quadratic rate. Again, O means "grows on the order of", so O(N) means grows on the order of N and O(N**2) means grows on the order of N**2.

Getting to Big-O Notation:

1. If we assume that every instruction in the computer takes the same amount of time to execute. Then the time taken for maximum is about 14N/speed and time for sort is about 8N**2/speed. We should really think about these formulas as (14/speed)N and (8/speed)N**2. We know the 14 and 8 came from the number of instructions inside loops that Python needed to execute: but a different Python interpreter (or a different language) might generate a different number of instructions and therefore a different constant. Thus, this number is based on TECHNOLOGY, and we want our analysis to be independent of technology. And, of course, "speed" changes based on technology too, and is part of that constant.

Getting to Big-O Notation:

- Since we are trying to come up with a "science" of algorithms, we don't want our results to depend on technology, so we are also going to drop the constant in front of the biggest term as well. For the reason explained above (relating to instructions generated by Python and the speed of the machine), this number is based solely on technology.
- Here is another justification for not being concerned with the constant in front of the biggest term.

Getting to Big-O Notation:

2. A fundamental question that we want answered about any algorithm is, "how much MORE resources does it need when solving a problem TWICE AS BIG". In maximum, when N is big (so we can drop the +9 without losing much accuracy) the ratio of time to solve a problem of size 2N to the time to solve a problem of size N is easily computed:

$$\frac{Imaximum(2N)}{Imaximum(N)} \sim \frac{14(2N)}{14N} \sim 2$$

The ratio is a simple number (no matter how many instructions are in the loop, since the constant 14 appears as a multiplicative factor in both the numerator and denominator, so in division it cancels itself out). So, we know for this code, if we double the size of the list, we double the number of instructions that maximum executes to solve the problem, and thus double the amount of time (for whatever the speed of the computer is).

Thus, the constant 14 is irrelevant when asking this "doubling" question.

Getting to Big-O Notation:

- Likewise, for sorting we can write

$$\frac{Isort(2N)}{Isort(N)} \sim \frac{8(2N)^{**}2}{8\ N^{**}2} \sim 4$$

- Again, the ratio is is a simple number, with the constant (no matter what it is, disappearing). So, we know for this code that if we double the size of the list, we increase by a factor of 4 the number of instructions that are executed, and thus increase by a factor of 4 the amount of time (for whatever the speed of the computer is).
- Thus, the constant 8 is irrelevant when asking this "doubling" question.

Getting to Big-O Notation:

- Note if we didn't simplify, we'd have

$$\frac{I(2N)}{I(N)} = \frac{8(2N)**2 + 12(2N) + 6}{8N**2 + 12N + 6} = \frac{32N**2 + 24N + 6}{8N**2 + 12N + 6}$$

- which doesn't simplify easiy; although, as N->inifinty, this ratio gets closer and closer to 4 (and is very close even for small-sized problems).
- As with air-resistance and friction in physics, typically ignoring the contribution of these negligible factors (for big, slow-moving objects) allows us to quickly solve an approximately correct problem.

Getting to Big-O Notation:

- Using big-O notation, we say that the complexity class of the code to find the maximum is O(N). The big-O means "grows on the order of" N, which means a linear growth (double the input size, double the time). For the sorting code, its complexity class is O(N**2), which means grows on the order of N**2, which means a quadratic growth rate (double the input size, quadruple the time).

Getting to Big-O Notation:

IMPORTANT: A Quick way to compute the complexity class of an algorithm

To analyze a Python function's code and compute its complexity class, we approximate the number of times the most frequently executed statement is executed, dropping all the lower (more slowly growing) terms and dropping the constant in front of the most frequently executed statement (the fastest growing term). We will show how to do this much more rigorously in the next lecture.

The maximum code executes the if statement N times, so the code is O(N). The sorting code executes the if statement N(N-1)/2 times (we will justify this number below), which is N**2/2 - N/2, so dropping the lower term and the constant 1/2, yields a complexity class of O(N**2) for this code.

Comparing Algorithms by their complexity classes

LECTURE 1

Comparing Algorithms by their complexity classes

- Primarily from this definition we know that if two algorithms, a and b, both solve some problem, and a is in a lower complexity class than b, then for all BIG ENOUGH N, Ta(N) < Tb(N): here Ta(N) means the Time it takes for algorithm a to solve the problem. Note that nothing here is said about small N; which algorithms uses fewer resources on small problems depends on the actual constants we dropped (and even on the terms that we dropped), so complexity classes have little to say for small problem sizes.

Comparing Algorithms by their complexity classes

- For example, if algorithm a is O(N) with a constant of 100, and algorithm b is O(N**2) with a constant of 1, then for values of N in the range [1,100],

```
Tb(N) = 1N**2 <= 100N = Ta(N)
```

- but for all values bigger than 100,

```
Ta(N) = 100N <= 1N**2 = Tb(N)
```

- As a second example, if algorithm a is O(N) with a constant of 1, and algorithm b is O(N**2) with a constant of 10, then for all values of N

```
Ta(N) = 1N <= 10N**2 = Tb(N)
```

Comparing Algorithms by their complexity classes

- So, in some cases a lower complexity class can be worse for small N, and in some cases it can be better. We need to go beyond complexity classes to find out. But it is guaranteed that FOR ALL SIZES BEYOND SOME VALUE OF N, the algorithm in the lower complexity class will run faster.
- Again, we use the term "asymptotic" analysis of algoritms to indicate that we are mostly concerned with the time taken by the code when N gets very large (going towards infinity). In both cases, because of their complexity classes, algorithm a will be better.
- What about the constants? Are they likely to be very different in practice? It is often the case that the constants of different algorithms are close. (They are often just the number of instructions in the main loop of the code). So, the complexity classes are a good indication of faster vs slower algorithms for all but the smallest problem sizes.

Comparing Algorithms by their complexity classes

- Although all possible mathematical functions might represent complexity classes (and many strange ones do), we will mostly restrict our attention to the following complexity classes. Note that complexity classes can interchangeably represent computing time, the # of machine operations executed, and such more nebulous terms as "effort" or "work" or "resources".
- As we saw before, a fundamental question about any algorithm is, "What is the time needed to solve a problem twice as big". We will call this the DOUBLING SIGNATURE of the complexity class (knowing this value empirically often allows us to know the complexity class as well).

Class	Algorithm Example	Doubling Signature
O(1)	pass argument->parameters/copying a reference	T(2N) = T(N)
O(LogN)	binary searching of a sorted list	T(2N) = c + T(N)
O(N)	linear searching a list (the in operator)	T(2N) = 2T(N)
O(NLogN)	Fast sorting	T(2N) = cN + 2T(N)

Fast algorithms come before here; NLogN grows a bit more slowly than linearly (because logarithms grow so slowly compared to N) and nowhere near as fast as O(N**2)

O(N**2)	Slow sorting; scanning N times list of size N	T(2N) = 4T(N)
O(N**3)	Slow matrix multiplication	T(2N) = 8T(N)
O(N**m)	for some fixed m: 4, 5, ...	T(2N) = 2**mT(N)

Tractable algorithms come before here; their work is polynomial in N. In the complexity calss below N is in an exponent.

O(2**N)	Finding boolean values that satisfy a formula	T(2N)=2**N T(N)

Comparing Algorithms by their complexity classes

- For example, for an O(N**2) algorithm, doubling the size of the problem quadruples the time required: If T(N) ~ cN^2 then T(2N) ~ c(2N)**2 = c4N**2 = 4cN**2 = 4T(N).
- Note that in Computer Science, logarithms are mostly taken to base 2. (Remember that algorithms and logarithms are very different terms). All logarithms are implicitly to base 2 (e.g., Log N = Log2 N). You should memorize and be able to use the following facts to approximate logarithms without a calculator.

```
Log 1000 ~ 10
```

- Actually, 2**10 = 1,024, 2**10 is approximately 1,000 with < a 3% error.

  Log a**b = b Log a, or more usefully, Log 1000**N = N Log 1000; so ...

  Log 1,000,000 = 20 : 1,000,000 = 1,000**2; Log 1000**2 = 2*Log 1000

  Log 1,000,000,000 = 30 : 1,000,000,000 = 1,000**3; Log 1000**3 = 3*Log 1000

Comparing Algorithms by their complexity classes

- So, note that Log is a very slowly growing function. When we increase from Log 1,000 to Log 1,000,000,000 (the argument grows by a factor of 1 million) the Log only grows by from 10 to 30 (by a factor or 3).
- In fact, we can compute these logarithms on any calculator that computes Log in any base. The following shows how to compute Log (base b) in terms of Log (base a).

```
Log (base b) X = Log (base a) X / Log (base a) b
```

Comparing Algorithms by their complexity classes

- So, Log (base b) X is just a constant (1/Log (base a) b) times Log (base a) X, so logarithms to any base are really all the same complexity class (regardless of the base) because they differ only by a multiplicative constant (which we ignore in complexity classes). For example,

```
Log(base 10) X = Log(base 2) X  /  Log(base 2) 10

~ .3 Log(base 2) X
```

- So, O(Log(base 10) X) = O(.3 Log(base 2) X) which simplifies to O(Log(base 2) X).

Comparing Algorithms by their complexity classes

IMPORTANT: Determining the Complexity Class Empirically - from a Doubling Signature

- If we can demonstrate that doubling the size of the input approximately quadruples the time of an algorithm, then the algorithm is O(N**2). We can use the doubling signatures shown above for other complexity classes as well. Thus, even if we cannot mathematically analyze the complexity class of an algorithm based on inspecting its code (something we will highlight in the next lecture), if we can measure it running on various sized problems (doubling the size over and over again), we can use the signature information to approximate its complexity class. In this approach we don't have to understand (even look at) the code.

Computing Running Times from Complexity Classes

LECTURE 1

Computing Running Times from Complexity Classes

- We can use knowledge of the complexity class of an algorithm to predict its actually running time on a computer as a function of N easily. For example, if we know the complexity class of algorithm a is O(N**2), then we know that Ta(N) ~ cN**2 for some constant c. The constant c represents the "technology“ used: the language, interpreter, machine speed, etc.; the N**2 (from O(N**2)) represents the "science/math" part: the complexity class. Now, given this information, we can time the algorithm for some large value N. Let's say for N = 10,000 (which is actually a pretty small N these days) we find that Ta(10,000) is 4 seconds.

Computing Running Times from Complexity Classes

- First, if I asked you to estimate Ta(20,000) you'd immediately know it is about 16 seconds (doubling the input of an O(N**2) algorithm approximately increases the running time by a factor of 4). Second, if we solve for c we have

```
Ta(N) ~ cN**2, substituting 10,000 for N and 4 for Ta(N) we have
Ta(10,000) = 4 ~ c 10,000**2 (from the formula), so solving for the the
technology constant c ~ 4x10**-8.
```

- So, by measuring the run-time of this code, we can calculate the constant "c", which involves all the technology (language, interpreter, computer speed, etc). Roughly, we can think of c as being the amount of time it takes to do one loop (# of instructions per loop/speed of executing instructions) where the algorithm requires N**2 iterations through the loops to do all its work.

Computing Running Times from Complexity Classes

- Therefore, Ta(N) ~ 4x10**(-8) x N**2. So, if asked to estimate the time to process 1,000,000 (10**6) values (100 times more than 10,000), we'd have

```
Ta(10**6) ~ 4x10**(-8) x (10**6)**2
Ta(10**6) ~ 4x10**(-8) x 10**12
Ta(10**6) ~ 4x10**4, or about 40,000 seconds (about 1/2 a day)
```

- Notice that solving a problem 100 times as big take 10,000 (which is 100**2) times as long, which is based on the signature for an O(N**2) algorithm when we increase the problem size by a factor of 100. If we go back to our sorting example,

$$\frac{I(100N)}{I(N)} \sim \frac{8(100N)^{**}2}{8\ N^{**}2} \sim 10{,}000$$

Computing Running Times from Complexity Classes

- In fact, while we often analyze code to determine its complexity class, if we don't have the code (or find it too complicated to analyze) we can double the input sizes a few times and see whether we can "fit the resulting times" to any of the standard signatures to estimate the complexity class of the algorithm. We should do this for some N that is as large as reasonable (taking some number of seconds to solve on the computer).
- Note for an O(2**N) algorithms, if we double the size of the problem from 100 to 200 values the amount of time needed goes up by a factor of 2**100, which is ~ 1.3x10**30. Notice that adding one more value to process doubles the time:
- this "exponential" time is the inverse function of logarithmic time, in terms of its growth rate: it grows incredibly quickly while logarithms grow incredible slowly.

Odds and Ends:

LECTURE 1

Odds and Ends:

- Note too that it is important to be able to analyze the following code. Notice that the upper bound of the inner loop (i) is changed by the outer loop.

```
for i in range(N):
    for j in range(i):
        body
```

- How many times does the "body" of the loop get executed? When the outer loop index i is 0, "body" gets executed 0 times; when the outer loop index i is 1, "body" gets executed 1 time; when the outer loop index i is 2, "body" gets executed 2 times; .... when the outer loop index i is N-1 (as big as i gets), "body" gets executed N-1 times. So, in totality, "body" gets executed 0 + 1 + 2 + 3 + ... + N-1 times, or dropping the 0, just 1 + 2 + 3 + ... + N-1 times. Is there a simpler way to write this sum?

Odds and Ends:

There is a simple, general closed form solution of adding up consecutive integers. Here is the proof that 1 + 2 + 3 + ... + N =N*(N+1)/2. This is a direct proof, but this relationship can also be proved by induction.

Let

```
S = 1 + 2 + 3 + ... + N-1 + N.
```

Since the order of the numbers makes no difference in the sum, we also have

```
S = N + N-1 + ... + 3 + 2 + 1.
```

If we add the left and right side (column by column) we have

```
S   =   1   +    2   +   ...   +   N-1   +  N
S   =   N   +  N-1   +   ...   +   2     +  1
-----------------------------------------------
2S  = (N+1) + (N+1) +   ... +   (N+1) + (N+1)
```

Odds and Ends:

- That is, each pair in the column sums to N+1, and there are N pairs to sum. Since there are N pairs, each summing to N+1, the right hand side can be simplified to just N*(N+1). so
- 2S = N*(N+1), therefore S = N(N+1)/2 = N**2/2 + N/2
- Thus, S is O(N**2): with a constant of 1/2 and a term of N/2 that is dropped (because its order is lower than that N**2). Note that either N or N+1 is an even number, so dividing their product by 2 is always a integer as it must be for the sum of integers: 6*7/2 = 21.
- So, looking back at the example of the code above, the total number of times the body gets executed is 0 + 1 + 2 + ... + N-1 which is the same as 1 + 2 + ... + N-1 so plugging N-1 in for N we have (N-1)(N-1+1)/2 = N**2/2 - N/2 which is still O(N**2) for the same reason.

Odds and Ends:

- We can apply this formula for putting N values at the end of a linked list that is initially empty (and has no cache reference to the last node). To put in the 1st value requires skipping 0 nodes; to put in the 2nd value requires skipping 1 nodes; to put in the 3rd value requires skipping 2 nodes; ... to put in the Nth value requires skipping N-1 nodes. So, the number of nodes skipped is 0 + 1 + ... + N-1, which by the formula is (N-1)N/2: so, building a linked list in this way is in the O(N**2) complexity class.

Fast Searching and Sorting

- There are obvious algorithms for searching a list in complexity O(N) and sorting a list in complexity O(N**2). But, there are surprisingly faster algorithms for these tasks: searching a list can be O(Log N) if the list is sorted (binary search); and sorting values in a list can be in O(N Log N).
- In lecture, I will briefly discuss the binary searching algorithm, for searching a sorted list: its complexity class is O(Log N). In fact the constant is 1, which means that when searching a list of 1,000,000 values, we must access the list at most about 20 times to either (a) find the index of the value in the list or (b) determine the value is not in the list. This is potentially 50,000 times faster than a loop checking one index after another (which we must do if the list is not sorted)! On large problems, algorithms in a lower complexity class typically execute much faster. Also note that when increasing the size of the list by 1,000 (to 1 billion values) the maximum number of accesses goes from 20 to 30.

Fast Searching and Sorting

- You should know that Python's sorting method on lists is O(N Log N), but we will not discuss the algorithm. As with binary searching, we will discuss the details in ICS-46.
- Note that we CANNOT perform binary searching efficiently on linked lists, because we cannot quickly find the middle of a linked list (for lists we just compute the middle index and access the list there). In fact, we have seen another self-referential data structure, binary search trees, that we can use to perform efficient searches (assuming the tree is bushy.

Fast Searching and Sorting

- Sorting is one of the most common tasks performed on computers. There are hundreds of different sorting algorithms that have been invented and studied. Many small and easy to write sorting algorithms are in the O(N**2) complexity class (see the sorting code above, for example). Complicated but efficient algorithms are often in the O(N Log N) complexity class. We will study sorting in more detail in ICS-46, including a few of these efficient algorithms.
- For now, memorize that fast (binary) searching is in the O(Log N) complexity class and fast sorting algorithms are in the O(N Log N) complexity class. If you are ever asked to analyze the complexity class of a task that requires sorting data as part of the task, assume the sorting function/method is in the O(N Log N) complexity class.

Summary

- To close for now, finding a new algorithm to solve a problem in a lower complexity class is a big accomplishment; a more minor (but still useful) accomplishment is decreasing the constant for an algorithm in the same complexity class (certainly useful, but often based more on technology than science). By knowing the complexity class of an algorithm we know a lot about the performance of the algorithm (especially if we measure the time it takes to solve certain sized problems: using this information we can accurately predict the time to solve other size problems).
- Finally, We can also reverse the process, and use a few measurements, doubling the size of the problem each time, to approximate the complexity class of an algorithm.