Analysis of Algorithms (CS395) Class Notes

Robert B. Heckendorn
Computer Science Department, University of Idaho

May 6, 2013

Contents

Preface

This course is the study of algorithms and their analysis.

These are notes I lectured from and look like it.

This set of notes is provided **as is** as an aid to study, discussions in class, the book, and posted algorithms. It is only just beginning and as such it is not complete. But I hope it is helpful.

Chapter 1

Introduction

This course is about algorithms and their analysis.

1.1 Algorithms

Algorithms are core of Computer Science. The study of algorithms is the study of process.

An **algorithm** is instructions for doing something: a formal process, a recipe, laws, method for doing something

Algorithms come from ancient times:

- How to make butter and cheese
- Egyptian division
- Euclid's algorithm

Algorithms are executable math and as precise as mathematics.

What are necessary requirements for an Algorithm?

1. Well-defined
2. Transformation (function) of input to output
3. Guaranteed to work over a well specific domain
4. Terminates in a finite number of steps

Properties of algorithms:

- notation for the algorithm (written out in English? Chinese? Pseudocode? Python?)
- many algorithms for the solving the same problem (same transformation)

Computers give us the ability to implement an algorithm. An **implementation** of an algorithm is a specific description of the algorithm but not the algorithm itself.

In this course we want to understand about design, construction, analysis of algorithms with plenty of examples.

like mathematics:

- some algorithms are just beautiful
- beauty vs "it seems to work"
- beauty provides clarity/provability/supportability

An algorithm is a well defined description of how to compute a function defined over a **domain** in a way that must terminate. Some examples of function domain and results (**range**):

Defined over reals returning a real

$$f : \mathbb{R} \to \mathbb{R}$$

Defined over pairs of reals returning a real

$$f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

Let counting numbers $\mathbb{N} = 1, 2, 3, ...$ then

$$f : \mathbb{N} \to \mathbb{N}$$

Defined over 8 bit strings and return a counting number:

$$f : \mathbb{B}^8 \to \mathbb{N} \times \mathbb{N}$$

1.2 Analysis

The other half: Analysis:

Talk about casual word: **tractable**. Analyzes the consumption of resources. (limited resources or constraints (e.g. time))

Resource: time

HP sorted a table using n^2 sort
result: never sorted the table. Feature never used.

switched to a quick sort algorithm and then
result: sorting the table became the default

solvable means it can take an astronomical amount of resources. Advanced Encryption Standard: 256bits $2^{256} \approx 10^{77}$. What are resources important?

Resource: time
You are doing a research project one algorithm takes 1 min to run or 1 day to run? what is the effect?

Resource: space
What if one algorithm takes 10MB and the other takes 10GB?

Resource: how space is organized
How you access memory can be critical

Resource: money
Power ball is solvable it just costs you a dollar a ticket! It is **solvable** but **intractable**.

To understand the practical difficulty of problems and algorithms we need to see that we are not only interested in the classical difference between easy problems and hard problems but we must consider the subjective **intractable problems** as measured by terms of resources available and our tolerance for consumption of those resources such as time and money. Such as will it finish before we die, in time to analyze the data, before the entropy death of the universe, in human reaction time etc.

1.3 Euclid's Algorithm

Euclid's algorithm for finding the greatest common division is over 2000 years old!

what is $\gcd(m, n)$? Gcd is:

$$\gcd : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

the largest $g : g|m$ and $g|n$

Notation: $|$ means divides evenly. Like $2|6$, but it is not the case that $4|6$. That is the remainder of 6 divided by 4 is 2 or $6 \mod 4 \equiv 2$.

Throughout the class we will just use Python as the way to express our algorithms rather than using a random pseudocode. This has the advantage that the result will in most cases be executable. A side-effect is that if it is precise enough to execute then it is precise enough to specify the algorithm.

Here is one way to write Euclid's Algorithm:

```
EUCLID'S ALGORITHM

def euclid(m, n) :
    if n==0 : return m
    return euclid(n, m % n)
```

example:

```
gcd(55, 22)
same as gcd(22, 55%22) = gcd(22, 11)
same as gcd(11, 22%11)
answer is 11

gcd(55, 20)
same as gcd(20, 55%20) = gcd(20, 15)
same as gcd(15, 20%15) = gcd(15, 5)
same as gcd(5, 15%5)
answer is 5

gcd(55, 21)
same as gcd(21, 55%21) = gcd(21, 13)
same as gcd(13, 21%13) = gcd(13, 8)
same as gcd(8,  13%8)  = gcd(8,  5)
same as gcd(5,  8%5)   = gcd(5,  3)
same as gcd(3,  5%3)   = gcd(3,  1)
same as gcd(1,  3%1)
answer is 1
```

why does this work: $\gcd(m, n) = \gcd(n, m\%n)$

CASE 1: for any $g|m, n$
and $r = m\%n$
we know $m = k * n + r$
so $m = k * n + r$ and we know $g|m$ and $g|k * n$

therefore $g|r$!
and so $g|m\%n$

CASE 2: Also if $g|n, r$
similarly $g|m$.

BECAUSE OF CASE 1 and 2 the sets of divisors is the same $gcd(m, n) = gcd(n, m\%n)$

CASE 3: What about if $g|m, r$ does $g|n$? No.

$\gcd(m, n) \rightarrow \gcd(n, m\%n)$ works, but
$\gcd(m, n) \rightarrow \gcd(m, m\%n)$ does not work because $\gcd(60, 14)! = \gcd(60, 4) = \gcd(60, 60\%14)$

For Euclid's Algorithm to be an algorithm it must terminate. Can we prove that Euclid's Algorithm terminates?

$0 <= m\%n < n$

what happens when m%n==0?

$m = n + r$
where $r = (m - n)$

CASE 1: for any $g|m, n$
and $r = m - n$
then $g|r$

CASE 2: Also if $g|n, r$
similarly $g|m - n$ and $g|n$
then $g|m$

CASE 3: What about if $g|m, r$ does $g|n$?
$g|m - n$ and $g|m$ then $g|n$ so... Yes.

$gcd(m, n) \rightarrow gcd(n, m - n)$ works but
$gcd(m, n) \rightarrow gcd(m, m - n)$ fails because $m - n$ is necessarily less than n

```
euclid-minus.py 55 21 2>/dev/null | head -40
55 21
21 34
34 -13
-13 47
47 -60
-60 107
107 -167
-167 274
```

```
274 -441
-441 715
715 -1156
```

swap fixes this

How much resources does it take? PRINT x's in the algorithm for each "unit" of resource used.

Now look at generating primes. Now look at sieve of Erastatanes.

How do measure how hard it is? When have we made a fundamental change.

Chapter 2

Basic Analysis of Resource Use

2.1 Order of execution (counting x's)

We can't count seconds for time because machines very. But we can count things that are proportional to the amount of time used. We select something as a measure of the size of the input and measure how the number of x's changes as the size of the input changes.

Loosely speaking, the **order of execution** is count of **key operations** (x's) in terms of size of input. If T measures the time resource used then:

$$T(\text{input}) = \text{CostOfOperation} * Count(\text{input})$$

where $Count$ counts the number of units of resource used as a function of input size. For example if doing a matrix multiply algorithm $Count$ might measure the number of multiplies (a key operation tied to time) needed for a given size of matrices.

What is going to make a bigger effect on the use of resources: adjusting CostOfOperation or how $Count$ relates input size to number of operations performed? If $Count$ goes up as the square of the input and you can force it to instead go up as $n \log(n)$ then that is a game changer.

Some examples of counting:

Problem	Size of input	Key operation
Search for items in a list	Num items	Comparison
Multiply matrices	Matrix dims or num elem	Multiply 2 nums
Primality testing	Size of n or num dig of n	Div 2 nums
Graph problem	Num V or E	Visiting a V or E

The practiced software engineer understands how their algorithm performs in terms of input size. The engineer might ask: how does the program slow as input size is increased?

Assume a problem takes about order n^2 to do its task and I double the input size?

$$(2n)^2/(2n) = 4$$

what about for order n?

$$2n/n = 2$$

for $\sqrt{n}$?

$$\sqrt{2n}/\sqrt{n} = \sqrt{2}$$

for $\log_2(n)$?

$$\log(2n)/\log(n) = (\log(2) + \log(n))/\log(n) = \log(2)/\log(n) + 1$$

for 2^n?

$$2^{2n}/2^n = 2^n$$

2.2 Best, Worst, Average

Three important measures for problems of size n.

$$\begin{aligned} worst &= \max_{\text{all of size } n} C(n) \\ best &= \min_{\text{all of size } n} C(n) \\ avg &= \operatorname{avg}_{\text{all of size } n} C(n) \end{aligned}$$

Best case performance is finding an example input that uses the least resources. **Worst case performance** is finding an example input that uses the most resources. **Average case performance** is the expected performance over all inputs. This generally assumes some stated distribution of inputs of size n.

2.3 Example Analysis of Linear Search

```
LINEAR SEARCH

def isin(l, a) :
    for i in range(0, len(l)) :             # from 0 to len(l)-1
        if l[i] == a : return True
    else : return False
```

The program can be tested with this input

```
print(isin([3, 1, 4, 1, 5, 9, 2], 2))
print(isin([3, 1, 4, 1, 5, 9, 2], 3))
print(isin([3, 1, 4, 1, 5, 9, 2], 8))
```

worst case $= n$
best case $= 1$
average case (assume prob of being found is p prob of being found at position k is p/n)

The details for average case are:

$$\begin{aligned} p/n(n(n+1))/2 + n(1-p) &= p(n+1)/2 + n(1-p) \\ &= (p/2)n + (p/2) + n(1-p) \\ &= (p/2 + 1 - p)n + (p/2) \\ &= (1 - p/2)n + (p/2) \end{aligned}$$

As $p \to 1$ this goes to $(n+1)/2$. As $p \to 0$ this goes to n.

this is linear... what do we really mean?

2.4 Order Notation

Order notation talks about the growth of resource consumption (within a constant). There are three popular measures.

2.4.1 BIG O (Upper Bound)

$$f(n) \in O(g(n))$$

if $f(n) \leq Cg(n) \ \ \forall n \geq n_0$ and some fixed C.

This says $f(n)$ can never catch up to $g(n)$ for large enough n and some fixed C. That is, g is an upperbound of f.

For example:

$$n \in O(n^2)$$

This says n is bounded above by a constant times n^2 for all n greater than or equal to some n_0. Another couple of examples:

$$n^3 \notin O(n^2)$$

$$8128n^2 \in O(n^2)$$

Prove $100n^2 + 17n + 360 \in O(n^2)$**:**

$$100n^2 + 17n + 360 \leq 360n^2 + 360n + 360 = 360(n^2 + n + 1) \leq 360(n^2 + n^2 + n^2) = 1080n^2$$

So for $n \geq 1$ and $C = 1080$ it is the case that $100n^2+17n+360 \leq Cn^2$ and hence $100n^2+17n+360 \in O(n^2)$.

Show $27n \in O(n^2)$**:**

$$27n \leq n^2 \ \text{ for } \ n \geq 27 \ \text{ and } \ C = 1$$

Note that we have found a C and an n_0 that satisfies the requirements for big O.

2.4.2 BIG OMEGA (Lower Bound)

$$f(n) \in \Omega(g(n))$$

if $f(n) \geq Cg(n) \ \ \forall n \geq n_0$ and some fixed C

$g(n)$ can never catch up to $f(n)$ for large enough n

$$n^3 \in \Omega(n^2)$$

2.4.3 BIG THETA (Approximately the Same)

$$f(n) \in \Theta(g(n))$$

if $C_1 g(n) \leq f(n) \leq C_2 g(n) \ \ \forall n \geq n_0$

For example, show:

$$\frac{(n-1)n}{2} \in \Theta(n^2)$$

Proof:

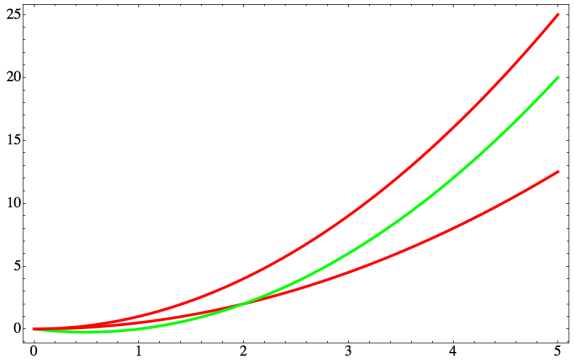
$$\leq (n-1)n \leq n^2 \ \ \text{for} \ \ n > 1$$

$$\begin{aligned}
(n-1)^2 &\leq (n-1)n \ \ \forall n > 1 \\
(n-1)^2 \frac{n^2}{n^2} &\leq (n-1)n \ \ \forall n > 1 \\
\frac{(n-1)^2}{n^2} n^2 &\leq (n-1)n \ \ \forall n > 1 \\
\frac{1}{2} n^2 \leq \frac{(n-1)^2}{n^2} n^2 &\leq (n-1)n \ \ \forall n > 2 + \sqrt{2}
\end{aligned}$$

Therefore with $C_1 = \frac{1}{2}$ and $C_2 = 1$ and $n_0 = 2 + \sqrt{2}$:

$$\frac{1}{2} n^2 \leq (n-1)n \leq n^2 \ \ \forall n > 2 + \sqrt{2}$$

You can see in the following graph that $n(n-1)$ is "pinned" between the other two functions:



2.4.4 Some Useful Theorems

These are statements about

$$f(n) \in O(f(n))$$

$$f(n) \in O(g(n)) \text{ iff } g(n) \in \Omega(f(n))$$

$$f(n) \in \Omega(f(n)) \text{ and } f(n) \in O(f(n))$$

$$\text{if } f(n) \in \Omega(f(n)) \text{ and } f(n) \in O(f(n)) \text{ then } f(n) \in \Theta(f(n))$$

$$\text{if } f(n) \in O(g(n)) \text{ and } g(n) \in O(h(n)) \text{ then } f(n) \in O(h(n))$$

$$\text{if } f_a(n) \in O(g_a(n)) \text{ and } f_b(n) \in O(g_b(n)) \text{ then } f_a(n) + f_b(n) \in O(max(g_a(n), g_b(n)))$$

This is saying that in computing the upperbound of resource consumption you only need to pay attention to the function that is taking the most amount of the resources. If an algorithm takes $O(an^3 + bn^2)$ amount of time it might as well just be counted as $O(an^3)$ or $O(n^3)$. This is true of any algorithm and any resource you choose to count.

2.4.5 Comparing Order of Growth

In selecting what algorithm of several to use you should compare the performance measures for the algorithms. How do you **compare algorithm performance** for two different algorithms in terms of the consumption of a resource (typically either space or time)? If you have the performance in order notation you are asking which grows faster $f(n)$ or $g(n)$?

Limit Method

$$\lim_{n\to\infty} f(n)/g(n)$$

$g > f$ if $lim = 0$
$g = f$ if $lim = c$ where $c > 0$
$g < f$ if $lim = \infty$ where $c > 0$

Example:

$$\lim_{n\to\infty} 10000n/(n^2) = 0$$

Example:

$$\begin{aligned}\lim_{n\to\infty} (n(n+1)/2)/(n^2) &= \lim_{n\to\infty} n^2/(2n^2) + n/(2n^2) \\ &= \lim_{n\to\infty} 1/2 + 1/(2n) \\ &= 1/2\end{aligned}$$

L'Hôpital's Rule

If you know:

$$\lim_{n\to\infty} f(n) = \infty \qquad \text{and} \qquad \lim_{n\to\infty} g(n) = \infty$$

and the derivatives of f and g exist then

$$\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} f'(n)/g'(n)$$

Example:

$$f(n) = \log(n) \qquad \text{and} \qquad g(n) = n$$

then

$$\lim_{n\to\infty} \log(n)/n = \lim_{n\to\infty} (1/n)/1 = 0$$

Some other helpful tips:

1. Logs used in "order of execution" operations are the same regardless of the base of the logarithm.

$$\Theta(\log_a(n)) = \Theta(\log_b(n))$$

   This is because they are only different by a constant factor:

$$\log_a(n)/\log_b(n) = \ln(b)/\ln(a)$$

$$(\ln(a)/\ln(b))\log_a(n) = \log_b(n)$$

2. All polynomials of the same degree are of the same class.

3. For $a > 0$

$$\log(n) < n^a < a^n < n! < n^n$$

4. a^n and b^n with $a \neq b$ have different orders of growth!

   Let's check this by taking limit.

$$\lim_{n\to\infty} a^b/b^n = \lim_{n\to\infty} (a/b)^n$$

2.5 Analysis of Nonrecursive Algorithms

The steps involved in **analyzing an algorithm**:

- How is input size measured?
- What is the **key operation** used to measure resource use?
- Check that resource consumption is a function of input size and not some other unmeasured things.
- Find formulation for resource consumption in terms of input size (may not be simple).
- Find simpler limiting function for resource consumption.

Table 2.1: Table of number of "steps" of execution for various orders of execution.

n	1	$\log_2(n)$	$\sqrt{n}$	n	$n\log_2(n)$	$n^{3/2}$	n^2	n^3	n^4	2^n	$n!$
2.	1.	1.	1.41421	2.	2.	2.82843	4.	8.	16.	4.	2.
3.	1.	1.58496	1.73205	3.	4.75489	5.19615	9.	27.	81.	8.	6.
4.	1.	2.	2.	4.	8.	8.	16.	64.	256.	16.	24.
5.	1.	2.32193	2.23607	5.	11.6096	11.1803	25.	125.	625.	32.	120.
6.	1.	2.58496	2.44949	6.	15.5098	14.6969	36.	216.	1296.	64.	720.
7.	1.	2.80735	2.64575	7.	19.6515	18.5203	49.	343.	2401.	128.	5040.
8.	1.	3.	2.82843	8.	24.	22.6274	64.	512.	4096.	256.	40320.
9.	1.	3.16993	3.	9.	28.5293	27.	81.	729.	6561.	512.	362880.
10.	1.	3.32193	3.16228	10.	33.2193	31.6228	100.	1000.	10000.	1024.	3.6288×10^6
11.	1.	3.45943	3.31662	11.	38.0537	36.4829	121.	1331.	14641.	2048.	3.99168×10^7
12.	1.	3.58496	3.4641	12.	43.0196	41.5692	144.	1728.	20736.	4096.	4.79002×10^8
13.	1.	3.70044	3.60555	13.	48.1057	46.8722	169.	2197.	28561.	8192.	6.22702×10^9
14.	1.	3.80735	3.74166	14.	53.303	52.3832	196.	2744.	38416.	16384.	8.71783×10^{10}
15.	1.	3.90689	3.87298	15.	58.6034	58.0948	225.	3375.	50625.	32768.	1.30767×10^{12}
16.	1.	4.	4.	16.	64.	64.	256.	4096.	65536.	65536.	2.09228×10^{13}
17.	1.	4.08746	4.12311	17.	69.4869	70.0928	289.	4913.	83521.	131072.	3.55687×10^{14}
18.	1.	4.16993	4.24264	18.	75.0587	76.3675	324.	5832.	104976.	262144.	6.40237×10^{15}
19.	1.	4.24793	4.3589	19.	80.7106	82.8191	361.	6859.	130321.	524288.	1.21645×10^{17}
20.	1.	4.32193	4.47214	20.	86.4386	89.4427	400.	8000.	160000.	1.04858×10^6	2.4329×10^{18}
21.	1.	4.39232	4.58258	21.	92.2387	96.2341	441.	9261.	194481.	2.09715×10^6	5.10909×10^{19}
22.	1.	4.45943	4.69042	22.	98.1075	103.189	484.	10648.	234256.	4.1943×10^6	1.124×10^{21}
23.	1.	4.52356	4.79583	23.	104.042	110.304	529.	12167.	279841.	8.38861×10^6	2.5852×10^{22}
24.	1.	4.58496	4.89898	24.	110.039	117.576	576.	13824.	331776.	1.67772×10^7	6.20448×10^{23}
25.	1.	4.64386	5.	25.	116.096	125.	625.	15625.	390625.	3.35544×10^7	1.55112×10^{25}

Table 2.2: Number of seconds of execution for a $1\mu s$ time step function for various orders of execution.

n	1	$\log_2(n)$	$\sqrt{n}$	n	$n\log_2(n)$	n^2	n^3	2^n	$n!$
10.	10^{-6}	3.3×10^{-6}	3.2×10^{-6}	0.00001	0.000033	0.0001	0.001	0.001	3.6
100.	10^{-6}	6.6×10^{-6}	0.00001	0.0001	0.00066	0.01	1.	1.3×10^{24}	9.3×10^{151}
1000.	10^{-6}	9.97×10^{-6}	0.000032	0.001	0.01	1.	1000.	1.1×10^{295}	4.0×10^{2561}
10000.	10^{-6}	0.000013	0.0001	0.01	0.13	100.	10^6	2.0×10^{3004}	2.8×10^{35653}
100000.	10^{-6}	0.000017	0.00032	0.1	1.7	10000.	10^9	1.0×10^{30097}	2.8×10^{456567}

2.5.1 Some Useful Sums

First we will need some useful sums

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + 4 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + 4^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3}$$

$$\sum_{i=1}^{n} i^3 = 1^3 + 2^3 + 3^3 + 4^3 + \cdots + n^3 = \left(\frac{n(n+1)}{2}\right)^2 \approx \frac{n^4}{4}$$

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + a^3 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \quad \text{for} \quad a \neq 1$$

Some example uses:

What is $\sum_{j=1}^{n} \sum_{i=1}^{j} i$ (note the bounds on the sums)?

Answer:

$$\begin{aligned}
\sum_{j=1}^{n}\sum_{i=1}^{j} i &= \sum_{j=1}^{n} \frac{j(j+1)}{2} \\
&= \sum_{j=1}^{n} \left(\frac{1}{2}j^2 + \frac{1}{2}j\right) \\
&= \sum_{j=1}^{n} \frac{1}{2}j^2 + \sum_{j=1}^{n} \frac{1}{2}j \\
&= \frac{1}{2}\sum_{j=1}^{n} j^2 + \frac{1}{2}\sum_{j=1}^{n} j \\
&= \frac{1}{2}\frac{n(n+1)(2n+1)}{6} + \frac{1}{2}\frac{n(n+1)}{2} \\
&= \frac{1}{2}\frac{n(n+1)(2n+1)}{6} + \frac{1}{2}\frac{n(n+1)}{2} \\
&= \frac{1}{12}(n(n+1)(2n+1) + 3n(n+1)) \\
&= \frac{1}{12}n(n+1)((2n+1)+3) \\
&= \frac{1}{12}n(n+1)(2n+4) \\
&= \frac{1}{6}n(n+1)(n+2)
\end{aligned}$$

What is $\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1$?

Answer:

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1 &= \sum_{i=1}^{n}(n-i) \\
&= \sum_{i=1}^{n} n - \sum_{i=1}^{n} i \\
&= n^2 - \frac{n(n+1)}{2} \\
&= \frac{n^2-n}{2}
\end{aligned}$$

What is $\sum_{i=1}^{n}\sum_{j=1}^{n} ij$?

Answer:

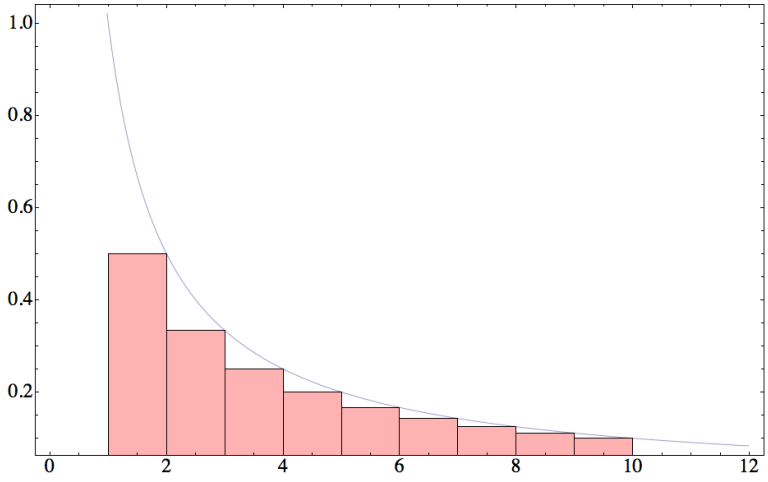
$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=1}^{n} ij &= \sum_{i=1}^{n} i \sum_{j=1}^{n} j \\ &= \sum_{i=1}^{n} i \left(\frac{n(n+1)}{2}\right) \\ &= \left(\frac{n(n+1)}{2}\right) \sum_{i=1}^{n} i \\ &= \left(\frac{n(n+1)}{2}\right) \left(\frac{n(n+1)}{2}\right) \\ &= \left(\frac{n(n+1)}{2}\right)^2\end{aligned}$$

The Harmonic Series

The Harmonic Series is useful when you get a sum of reciprocals.

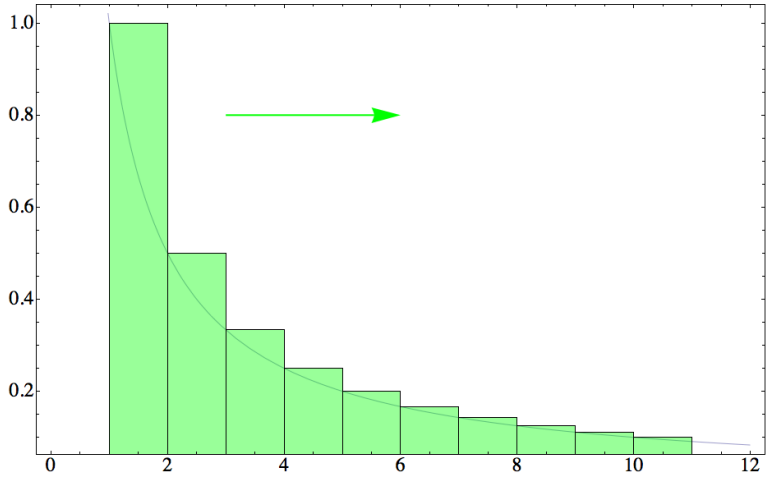
Here is an upper bound for the Harmonic Series:

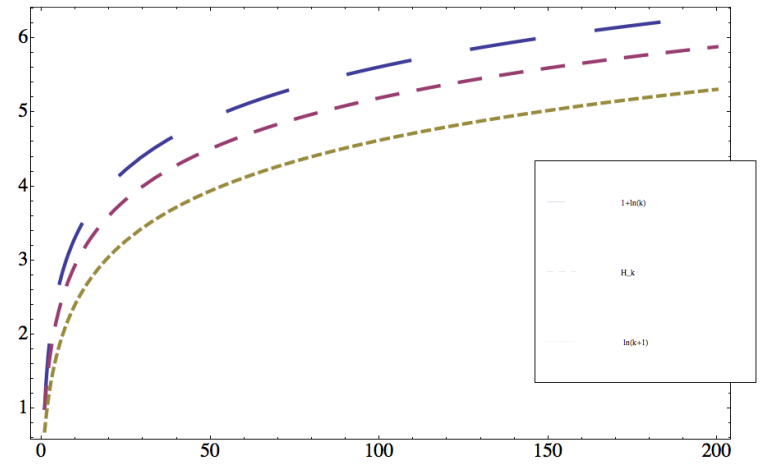
$$
\begin{aligned}
H_n - 1 &= \sum_{k=2}^{n} 1/k \\
H_n - 1 &= 1/2 + 1/3 + 1/4 + \cdots + 1/n \\
H_n - 1 &< \int_1^n \frac{1}{x}\, dx \\
H_n - 1 &< (\ln(n) - \ln(1)) \\
H_n &< 1 + \ln(n)
\end{aligned}
$$



Here is an lower bound for the Harmonic Series:

$$
\begin{aligned}
H_n &= \sum_{k=1}^{n} 1/k \\
H_n &> \int_1^{n+1} \frac{1}{x}\, dx \\
H_n &> \ln(n+1) - \ln(1) \\
H_n &> \ln(n+1)
\end{aligned}
$$





2.6 Some Example Analyses

2.6.1 Find the Maximum of a List

```
FIND THE MAXIMUM ELEMENT

def maxelem(l) :
    maxval = l[0]
    for i in range(1, len(l)) :
        if l[i] > maxval : maxval = l[i]
    return maxval
```

```
print(maxelem([31, 1, 4, 1, 5, 9, 2]))
print(maxelem([31, 1, 49, 1, 5, 9, 2]))
print(maxelem([3, 1, 4, 1, 5, 9, 265]))
```

The cost $C(n)$ for input size n assumes a key operation of testing an element to see if it is a new max: `if l[i] > maxval :  maxval = l[i]`. It happens each time through the loop therefore:

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

2.6.2 Does List Have Only Unique Values?

```
ARE ALL THE ELEMENTS UNIQUE

def unique(l) :
    for i in range(0, len(l)-1) :
        for j in range(i+1, len(l)) :
            if l[i] == l[j] : return False
    return True
```

```
print(unique([31, 1, 4, 1, 5, 9, 2]))
```

```
print(unique([31, 1, 49, 11, 5, 9, 2]))
print(unique([3, 1, 4, 11, 5, 9, 265, 3]))
print(unique([3, 1, 4, 11, 11, 5, 9, 265]))
```

The Analysis of the worst case with the test being the key operation:

$$\begin{aligned}
C_{worst}(n) &= \sum_{i=0}^{n-2} \sum_{i=i+1}^{n-1} 1 \\
&= \sum_{i=0}^{n-2} (n-1-i) \\
&= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i \\
&= (n-1)^2 - \frac{(n-2)(n-1)}{2} \\
&= \frac{2(n-1)(n-1) - (n-2)(n-1)}{2} \\
&= \frac{n-1}{2}(2n-2-n+2) \\
&= \frac{n-1}{2} n \\
&\in \Theta(n^2)
\end{aligned}$$

2.7 Matrix Multiply

The following is an example of an $O(n^3)$ algorithm where n measures the number of multiplies done as a measure of execution time.

```
MATRIX MULTIPLY

import numpy as np   # numpy only reliably is found for Python 2

# assume square matrix
def mm(a, b) :
   c = np.zeros_like(a)
    for i n range(0, len(a)) :
        for j in range(0, len(a)) :
           c[i][j] = 0
            for k in range(0, len(a)) :
                c[i][j] += a[i][k]*b[k][j]
    return c
```

```
x = np.array([[1,2], [7, 11]])
y = np.array([[1, 0], [0, 1]])

print(mm(x, y))
```

Here the computation is straight forward:

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 \in \Theta(n^3).$$

2.8 Recursive Analysis

Many algorithms are best comprehended/implemented recursively.

2.9 Space Allocation (Number as Binary String)

2.9.1 Basic Recursive Number to Binary Algorithm

Consider this algorithm where `size` is the number of bits to display of the number.

```
NUMBER TO BINARY ALGORITHM

def num2bin(n, size) :
    if size==0 : return ""
    elif n%2==0 : return num2bin(n//2, size-1) + "0"
    else : return num2bin(n//2, size-1) + "1"
```

```
# TEST CODE:
for i in range(0, 20) :
    print(num2bin(i, 8))
```

The basic unit of work (key operation) in this case is the number of characters copied when the string concatenation is performed with the + operator.

For a number that takes *bits* bits to represent: the number of characters copied at each recursive call is:

$$bits + (bits - 1) + (bits - 2) + (bits - 3) + \cdots + 1 = \frac{bits(bits + 1)}{2}$$

2.9.2 Divide and Conquer Recursive Number to Binary Algorithm

In this version of the algorithm, we divide the work in half to prevent a lot of concatenating.

```
NUMBER TO BINARY

def num2bin(n, size) :
    if size == 1 :
        if n%2==1 : return "1"
        else : return "0"
    b2 = size//2
    return num2bin(n//(2**b2), size-b2) + num2bin(n, b2)
```

The analysis of the space consumed in the this last algorithm precedes as follows:

Let $bits = 2^k$ be the size of the problem then

$$M(bits) = \begin{cases} bits + 2M(bits/2) & \text{if } bits > 1 \\ 1 & \text{if } bits = 1 \end{cases}$$

$$\begin{aligned}
M(bits) &= bits + 2M(bits/2) \\
M(bits) &= bits + 2(bits/2 + 2M(bits/4)) \\
M(bits) &= bits + 2(bits/2 + 2M(bits/4)) \\
M(bits) &= bits + bits + 4M(bits/4) \\
M(bits) &= bits + bits + 4(bits/4 + 2M(bits/8)) \\
M(bits) &= bits + bits + bits + 8M(bits/8) \\
&\vdots \qquad\qquad \vdots \\
&\text{(substitute } 2^k \text{ for } bits \text{ and notes that } bits \text{ is added } k \text{ times in that case)} \\
M(bits) &= k2^k + 2^k M(2^k/2^k)) \\
&= k2^k + 2^k \\
&= (k+1)2^k \\
&= k2^k + 2^k \quad \text{NOTE: } k2^k \text{ dominates so ignore } 2^k \\
&\text{(substitute } bits \text{ for } 2^k \text{ and } \log(bits) \text{ for } k) \\
&= \log_2(bits)bits
\end{aligned}$$

So the answer is $O(\log_2(bits)bits)$.

For example if the number of bits is 32 then $k = 5$ and so $5 * 32 + 32 = 192$ characters.

In the algorithm that adds one character at a time, if $size = 2^k$ then it takes $\frac{2^k(2^k+1)}{2}$ which for $k = 5$ is 496 characters. This is much slower to copy all those characters.

2.9.3 Using Faster Operators

As a note about Python: This algorithm has the same analysis as the previous but replaces some slower operators with faster ones. The bitwise logical and operator $n\&1$ tests to see if the **least significant bit** is 1 or 0. This is a fast test for if n is odd or even. >> is a right shift operator. For example >> 1 divides by 2, >> 2 divides by 4, >> 3 divides by 8, etc. This removes all the slow operators such as divide, mod, and exponentiation.

```
def num2bin(n, size) :
    if size == 1 :
        if n&1==1 : return "1"
        else : return "0"
    b2 = size>>1
    return num2bin(n>>b2, size-b2) + num2bin(n, b2)
```

2.10 Towers of Hanoi

The following algorithm does the tower of Hanoi and keeps track of how many disks are on each pole. Note that `from` is a keyword in Python so if we want to keep this code executable we have to alter the variable name to `frum` in order to to prevent an error if we try to execute it.

```
TOWER OF HANOI WITH DISK COUNT

def other(frum, to) : return 3 - (frum + to)

def hanoi(disks, howmany, frum, to):
    if howmany>1 : hanoi(disks, howmany-1, frum, other(frum, to))
    disks[frum] -= 1
    disks[to] += 1
    print("move", frum, "to", to, disks)
    if howmany>1 : hanoi(disks, howmany-1, other(frum, to), to)
```

```
disks = [6, 0, 0]
hanoi(disks, disks[0], 0, 2)
```

This algorithm shows that you don't even need to keep track of the number of disks on each pole in order to answer the question: what disk do you move next?

```
TOWER OF HANOI

def other(frum, to) : return 3 - (frum + to)

def hanoi(howmany, frum, to):
    if howmany>1 : hanoi(howmany-1, frum, other(frum, to))
    print("move", frum, "to", to)
    if howmany>1 : hanoi(howmany-1, other(frum, to), to)
```

Test code for the towers of Hanoi:

```
hanoi(6, 0, 2)
```

How many disks are moved when you start with d disks? This is a exercise that assumes the key operation is moving a disk which happens in one place in the code. Assuming the number of disks moved is $D(d)$ then the recurence relation is:

$$D(d) = \begin{cases} D(d-1) + 1 + D(d-1) & \text{if } d > 1 \\ 1 & \text{if } d = 1 \end{cases}$$

Using substitution again we get:

$$\begin{aligned} D(d) &= 2D(d-1) + 1 \\ &= 2(2D(d-2) + 1) + 1 \\ &= 2(2(2D(d-3) + 1) + 1) + 1 \\ &= 2^{d-1} + 2^{d-2} + 2^{d-3} + \cdots + 4 + 2 + 1 \\ &= 2^d - 1 \end{aligned}$$

Chapter 3

Divide and Conquer

3.1 The Master Theorem

The Master Theorem is useful when the problem is broken up into b equal parts with some assembly required that takes $f(n)$ to do.

Given:

$$T(n) = aT(n/b) + f(n) \quad \text{where} \quad f(n) \in \Theta(n^d),\ d \geq 0$$

Then

If $a < b^d$ then $T(n) \in \Theta(n^d)$

If $a = b^d$ then $T(n) \in \Theta(n^d \log(n))$

If $a > b^d$ then $T(n) \in \Theta(n^{\log_b(a)})$

Note: The same results hold with O instead of Θ.

Examples:

If $T(n) = 4T(n/2) + n$ then $a = 4, b = 2, d = 1$ so $4 > 2^1$ therefore $T(n) \in \Theta(n^{\log_2(4)}) = \Theta(n^2)$

If $T(n) = 4T(n/2) + n^2$ then $a = 4, b = 2, d = 2$ so $4 = 2^2$ therefore $T(n) \in \Theta(n^2 \log(n))$

If $T(n) = 4T(n/2) + n^3$ then $a = 4, b = 2, d = 3$ so $4 < 2^3$ therefore $T(n) \in \Theta(n^3)$

If $T(n) = 2T(n/2) + n$ then $a = 2, b = 2, d = 1$ so $2 = 2^1$ therefore $T(n) \in \Theta(n^1 \log(n)) = \Theta(n \log(n))$

If $T(n) = 7T(n/2) + 1$ then $a = 7, b = 2, d = 0$ so $7 > 2^0$ therefore $T(n) \in \Theta(n^{\log_2(7)}) = \Theta(n^{2.807})$

Note that if $a = b$ and $f(n) = O(n)$ which makes $d = 1$ then $T(n) = aT(n/a) + O(n)$ and by the Master Theorem $T(n) \in \Theta(n \log(n))$ regardless of the value of a.