Algorithm Basics

Acknowledgement: All class notes have drawn heavily from the following books: (1) Éva Tardös and Jon Kleinberg, “Algorithm Design”, Addison-Wesley, (2) Michael T. Goodrich and Roberto Tamassia,“Algorithm Design and Applications”, Wiley, (3) Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, “Introduction to Algorithms”, The MIT Press.

Class Web Page, Syllabus, Reminders

- **Email Policy**: Announcements will be made using Canvas using your Clemson id. Use a Subject header like "CpSc 3120 – "Brief problem Description"". **Do not use chain emails; do not reply to group emails**…
- **Reasonable familiarity** with C/C++ and Pointers, Prerequisites, Linux Systems, Basic Data Structures, …
- **No Make-ups**, except in emergencies with prior approval.
- Programming Assignments should be done on School Linux Machines.
- **Any grade appeal must be made within one week of posting of grades; check frequently.**
- **Read a textbook; class notes are not enough**.
- Emphasis is on problem solving [overall goal/objective] by using algorithms and data structures and evaluation of algorithms . We will use both C and C++ in our examples in class [choice of language is not the focus].
- **Programming ≠ Algorithmic Problem Solving; "Computer Science is no more about computers than astronomy is about telescopes" – folklore, sometimes attributed to E. Dijkstra.**

What is an Algorithm

- Informally speaking, an algorithm is a *step-by-step* procedure to solve a problem in *finite* time; most algorithms transform input objects to output objects. Again, lacking formal mathematical rigor, a good algorithm [e.g., Google Map] has several properties:
  - **Precision**: The steps are precisely (unambiguously) stated.
  - **Uniqueness**: The algorithm accepts some input and produces some output; it always generates the same output given identical input.
  - **Termination** (Finiteness): The algorithm terminates in *finite* time for a *vali*d input.

Caveats: There are exceptions: Probabilistic algorithms (Monte Carlo, simulated annealing, parallel & Distributed Algorithms, etc.)

Examples: Google map, DNA sequencing, Weather prediction, Aircraft Design, stock market prediction, geometric modeling, and many many others.

- How do we evaluate an algorithm?
  - One way (**Experimental**): write a program, use system time counter to measure the actual running time of the algorithm; repeat the experiment varying input size, plot running time against input size, draw conclusions; limitations: (1) dependence on platform dependent parameters, (2) accurate measurement is expensive, not probably needed in the design phase of a system, (3) others.
  - Second way (**Theoretical**): use a high-level description (pseudo-code) of the algorithm, characterize the running time as a function of the input size (consider all inputs). Advantages: (1) evaluation is platform independent (2) easier to compare multiple choices for a given problem in a platform independent way. Limitations: does not consider platform dependent parameters which may be more important in certain situations.

Pseudocode

- A structured high-level description of an algorithm – more structured than plain English but hides language specific details – objective is to provide the details without any language syntax, it's mechanical to covert it to an implementation in any language on any platform.

Control flow: if … then … [else …], while … do…, repeat… until…, for … do…,
Method declaration: Algorithm method (arg [, arg…])
Input …, Output …
Expressions: Assignment, equality testing, others

Implicit Assumptions: (1) Single CPU, potentially unbounded number of memory cells that can hold an arbitrary number; (2) each memory cell can be accessed in unit time (RAM).
Primitive operations: basic computations identified in the pseudocode: evaluating an expression, assignment, indexing an array, comparison, returning from a module etc. [Not all of these are always considered; we will see simple examples]

```
Algorithm Array_Max (A, n)
  Input: Array A of size n
  Output: maximum element
  of A
  Max = A[0]
  for i = 1 to n−1 do
    {if A[i] > max then
       max = A[i] }
  return max
```

Estimating (Analyzing) Execution Time

- There are 3 cases that need be considered: **Best Case, Worst Case** and **Average Case**. Best case analysis is mostly used for benchmarking, so is worst case, although in many application scenarios worst case analysis is crucial (online games, finance, robotics, weather prediction, other hard deadline dependent applications. Average case analysis is more problematic. While "average" is informally defined to be mean over all possible scenarios, it may be tricky. We will use the simple algorithm Arrray_Max to get some more intuitive ideas.

```
Algorithm Array_Max (A, n)
  Input: Array A of size n
  Output: maximum element of A
  Max = A[1]
  for i = 2 to n do
     {if A[i] > max then
        max = A[i] }
  return max
```

- Note that #element-comparison is always n – 1, or O(n), no matter what the input is; all three measures are the same.
- Assume A is an array of integers; **note** that absolute values of the integers do not matter; what matters is the relative positioning of the integers in the array.
- Let T(n) be the number of times the assignment "max=A[i]" is executed. Note: $T_{max}(n) = n-1$ and $T_{min}(n) = 0$. What is the average case behavior?
- Consider the case n = 3; there are 6 possible inputs; the adjoining table shows $T_{avg}(3) = (2 + 1 + 1 + 1 + 0 + 0)/6 = 0.833..$ [we assume each input instance is equally probable]; what does that mean? [Elaborate]
- It is obvious $T_{max}(4) = 3$ and $T_{min}(4) = 0$. Can you compute $T_{avg}(4)$? **Do it** by exhaustive enumeration.
- *Average case analysis, even for simpler algorithms, is relatively complicated*.
- We need to learn a few things before we do some of that towards the later half of the semester.

A(1)	A(2)	A(3)	T(3)
1	2	3	2
1	3	2	1
2	1	3	1
2	3	1	1
3	1	2	0
3	2	1	0

Execution Time (contd.)

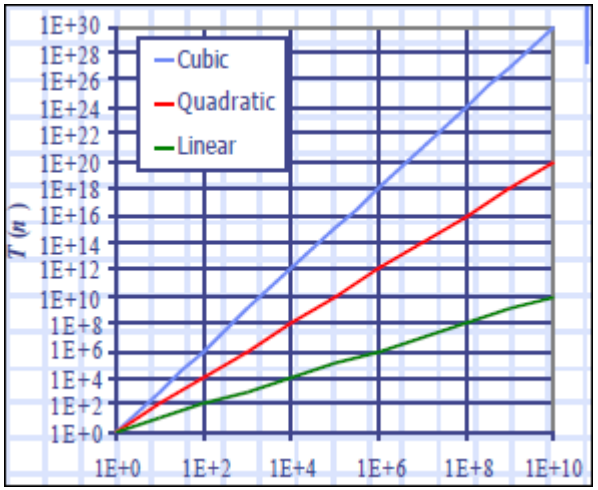
- Note: We can also account for other primitive operations performed by the algorithm like assigning a value to a variable, indexing into an array, calling a procedure or returning from a procedure, etc. [max=A[1] ≡ index an array + 1 assignment, before entering the loop, i is assigned to 2, compare i with 2, inside the loop i<n is executed n-1 times; and so on]

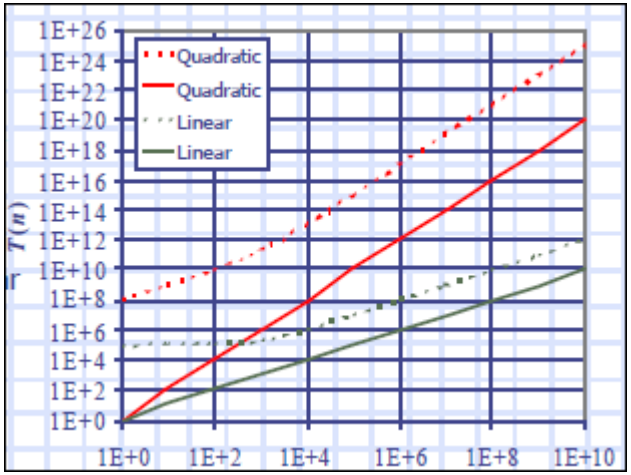
Algorithm Array_Max (A, n)	#operations
Input: Array A of size n	
Output: maximum element of A	
max = A[1]	2
for i = 2 to n **do**	2+n
{if A[i] > max **then**	2(n-1)
max = A[i] }	**2(n-1)**
{increment i by 1}	2(n-1)
return max	1

- Most of the time, we disregard primitive operations and concentrate on what is the *metric of performance relative to the algorithm* [e.g., the # of element comparisons in sorting related problems]
- Either way, the fact that **T(n) = c.n,** where n is size of the input array and c is a constant, indicates that the execution time of the algorithm is linear in n (size of the array); the platform dependent details will affect the proportionality constant and will not change the intrinsic characteristic of the algorithm. We will elaborate on the issue with other examples. We need some tools.

Growth Rate of functions

- Consider simple functions: $f_1(n) = c_1 n$ (linear), $f_2(n) = c_2 n^2$ (quadratic), $f_3(n) = c_3 n^3$ (cubic), $f_4(n) = c_4 \log n$ (logarithmic), $f_5(n) = c_5 2^n$ (exponential) and so on. Note:
  - The slopes of the functions can easily be seen by differentiating w.r.t. n (considering n as a continuous variable, of course). Or, use spreadsheet to do log-log charts [workout manually].
  - No matter what the constants are, for sufficiently large values of n, growth rate of quadratic functions are greater than that of liner functions as well as will be bigger than them. Informally speaking, the fact is expressed as "quadratic functions are **asymptotically** bigger than linear functions and so on ["asymptotically" means when the argument of the function is very large]; Explain.
  - Constant factors and lower order terms do not affect the growth rates, e.g., $4n^2$ and $n^2 + 2n$ are both quadratic functions of n (why?)





Growth Rate and Big-Oh Notation

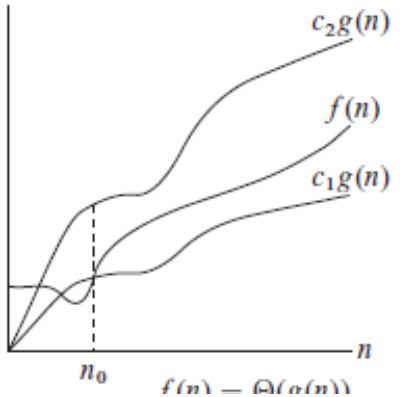
- Consider 2 functions f(n) and g(n); we say **f(n) = O(g(n)) if there exist 2 +ve constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$, i.e., g(n) is an asymptotic upper bound on f(n).**
  Examples:
  - 4n + 1000 is O(n): $4n + 1000 \leq cn \Rightarrow (c - 4)n \geq 1000 \Rightarrow n \geq 1000/(c-4) \Rightarrow$ Use c = 5, n_0=1000.
  - 4n – 1000 is O(n): $4n - 1000 \leq cn \Rightarrow (c - 4)n \geq 1000 \Rightarrow n \geq 1000/(c-4) \Rightarrow$ Use c = 5, n_0 =1
  - $3n^2 + 100n + 5$ is $O(n^2)$: $3n^2 + 100n + 5 \leq cn^2 \Rightarrow (c - 3)n^2 - 100n \geq 5 \geq$ choose c =4, then we need $n^2 - 100n \geq 5 \Rightarrow$ Choose n_0 = 101 [*there is no requirement that choice of c and n_0 must be minimum*]
  - 3 log n + log log n is O(log n): $3 \log n + \log \log n \leq c \log n \Rightarrow (c - 3)\log n \geq \log \log n \Rightarrow$ Use c = 4 and n_0 =2.
- The big-Oh notation gives an *upper bound on the growth rate* of a function. The statement "f(n) is O(g(n))" means that the growth rate of f(n) is *no more* than the growth rate of g(n). We use the big-Oh notation to rank functions according to their growth rate. The big-Oh notation allows us to say that a function of n is "less than or equal to" another function (by the inequality "<" in the definition), *up to a constant factor* (by the constant c in the definition) and in the asymptotic sense as n grows toward infinity (by the statement "$n \geq n_0$" in the definition). Guidelines:
  - If is f(n) a polynomial of degree d, then f(n) is $O(n^d)$, i.e., drop lower-order terms and constant factors
  - Use the smallest possible class of functions: Say "2n is O(n)" instead of "2n is $O(n^2)$"
  - Use the simplest expression of the class: Say "3n+ 5 is O(n)" instead of "3n + 5 is O(3n)"

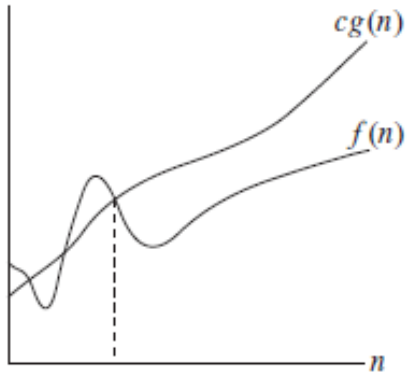
Big-Omega and Big Theta Notations

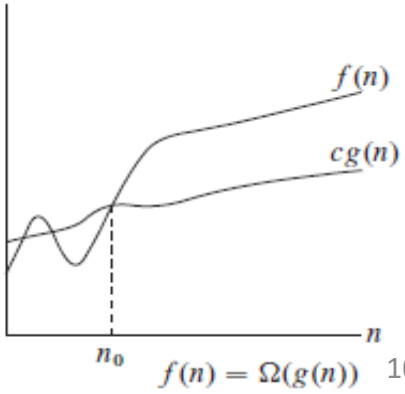
- **Note:** Sometimes we wish to give the exact leading term in an asymptotic characterization. In that case, we would say that "f(n) is g(n) + O(h(n))," where h(n) grows slower than g(n). For example, we could say that $2n \log n + 4n + 10\sqrt{n}$ is $2n \log n + O(n)$.
- **Big-Omega**: a function f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c.g(n)$ for $n \geq n_0 \Rightarrow$ f(n) is $\Omega(g(n))$ if f(n) is asymptotically greater than or equal to g(n), asymptotic lower bound.
- **Big-Theta**: a function f(n) is $\Theta(g(n))$ if there are constants $c_1 > 0$, $c_2 > 0$ and an integer constant $n_0 \geq 1$ such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for $n \geq n_0 \Rightarrow$ f(n) is $\Theta(g(n))$ if f(n) is asymptotically equal to g(n).

Graphic examples of the O, Ω and Θ Notations

- The first figure gives an intuitive picture of functions f(n) and g(n), where f(n) = Θ(g(n)). For all values of n at and to the right of n_0, the value of f(n) lies at or above $c_1g(n)$ and at or below $c_2g(n)$. In other words, for all n n_0, the function f(n) is equal to g(n) to within a constant factor. We say that **g(n) is an asymptotically tight bound for f(n)**.
- The second figure shows O-notation gives an upper bound for a function to within a constant factor.
- The third figure shows that the Ω gives a lower bound for a function to within a constant factor.
- The definition of Θ(g(n)) requires that every member f(n) and g(n) be asymptotically nonnegative, that is, that f(n) be nonnegative whenever n is sufficiently large. (An asymptotically positive function is one that is positive for all sufficiently large n.) Consequently, the function g(n) itself must be asymptotically nonnegative. We shall therefore assume that every function used within Θ - notation is asymptotically nonnegative.







Useful Algebra from High School (Logarithms)

- Logarithms and Exponents: **$\log_b a = c$ if $a = b^c$**. As is the custom in the computing literature, we omit writing the base b of the logarithm when b = 2. For example, log 1024 = 10 (or, lg 1024 =10, binary logarithm). We need to remember the following simple rules (remember how to prove them):

$\log_b ac = \log_b a + \log_b c$; $\log_b a/c = \log_b a - \log_b c$; $\log^k(n) = (\log n)^k$

$\log_b a^c = c\log_b a$; $\log_b a = (\log_c a)/(\log_c b)$; $b^{\log_c a} = a^{\log_c b}$; $\log(\log n)$ $\log \log (n)$

$(b^a)^c = b^{ac}$; $b^a b^c = b^{a+c}$; $b^a/b^c = b^{a-c}$;

$\log_b(1/a) = -\log_b a$; $\log_b a = 1/\log_a b$; $a^{\log_b c} = c^{\log_b a}$

For all n and $a \geq 1$, a^n is monotonically increasing in n (we normally use $0^0 = 1$). We relate the rates of growth of polynomials and exponentials by the following fact. For all real constants a and b such that $a > 1$, $\lim_{a\to\infty}(n^b/a^n) = 0$ [Use L'Hospital rule to prove if you want]; we conclude $n^b = o(a^n)$; thus, any exponential function with a base strictly greater than 1 grows faster than any polynomial function.

- Consider some interesting cases in algorithm analysis (they all can be easily shown to be true using the above rules)

$$\log(2n\log n) = 1 + \log n + \log\log n; \quad \log(n/2) = \log n - 1; \quad \log\sqrt{n} = \frac{\log n}{2}$$

$$\log\log\sqrt{n} = \log\log n - 1; \quad \log_4 n = \frac{\log n}{2}; \quad \log^{2^n} = n; \quad 2^{\log n} = n; \quad 2^{2\log n} = n^2$$

$$4^n = 2^{2n}; \quad n^2 2^{3\log n} = n^2 n^3 = n^5; \quad \frac{4^n}{2^n} = 2^n;$$

Ceiling and Floor

- $\lceil x \rceil$ = smallest integer greater than or equal to x ("ceiling") and $\lfloor x \rfloor$ = the largest integer less than or equal to x ("floor"). Note, for any real x,

$$\mathbf{x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1}$$

- For any integer n,

$$\lfloor n \rfloor + \lceil n \rceil = 2n.$$

Also note that for any real number $x \geq 0$ and integers m, $n > 0$

$$\left\lceil \frac{\lceil x/n \rceil}{m} \right\rceil = \left\lceil \frac{x}{nm} \right\rceil, \quad \left\lfloor \frac{\lfloor x/n \rfloor}{m} \right\rfloor = \left\lfloor \frac{x}{nm} \right\rfloor, \quad \left\lceil \frac{n}{m} \right\rceil \leq \frac{n+(m-1)}{m}, \quad \left\lfloor \frac{n}{m} \right\rfloor \geq \frac{n-(m-1)}{m}$$

Natural Logarithm, Euler constant ***e***

- Using **e** to denote 2.71828 … , the base of the natural logarithm function, we have for all real x, we can write

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} \ldots \quad \text{indicating } e^x \geq 1 + x$$

- The number **e** is a mathematical constant known as Euler's constant. The number e is of eminent importance in mathematics, alongside **0**, **1**, **π** and **i**. All five of these numbers play important and recurring roles across mathematics (and computer science). It can be computed by the series

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

- In the inequality $e^x \geq 1 + x$, the inequality hold only when x = 0. When $|x| \leq 1$, we use the approximation $1 + x \leq e^x \leq 1 + x + x^2$

- When $x \to 0$, the approximation of e^x by 1 + x is quite good: $e^x = 1 + x + \Theta(x^2)$
- Also, for all x,

$$\lim_{n \to \infty} \left(1 + \frac{x}{n}\right) = e^x$$

Modulo Numbers

- Given two positive integers, m and n, we define m modulo n (or, m mod n) as the remainder of the integer division of m by n [n is the modulus], i.e., 8 mod 3 = 2, 16 mod 4 = 0.
- **Modulo arithmetic**: Explain the basics, like (8 + 9) mod 7 = 3 = $8_7 + 9_7$, modulo n numbers are (0, 1, 2, …, n – 1).
- **Integers $\mathbb{Z}$ = {…, -2, -1, 0, 1, 2,}**
- Consider the integers $\mathbb{Z}$. For a, b $\in \mathbb{Z}$, we say that b divides a, or that a is divisible by b, if there exists c $\in \mathbb{Z}$ such that a = bc. If b divides a, then b is called a divisor of a, and we write a|b.
- Fix a +ve integer N (called **modulus**): for two integers a and b we write a = b (mod N) if N divides b − a, and we say that a and b are **congruent modulo N**.
- Consider (mod N) as a postfix operator on an integer which returns the smallest non-negative value equal to the argument modulo N, e.g., 16 (mod 5) = 1, -16 (mod 5) = 4; note the difference with % operator in C or Java; (-3) mod 2 = 1 while (-3)%2 = -1.
- Define the set N ={0,1,2, ..,N-1} = set of all possible remainders mod N. Addition and multiplication are defined the usual way on N. (13+12) mod 15 = 10, (5*6) mod 15 = 5, (8+4) mod 15 = 12, etc.
- Addition and multiplication mod N work almost the same as arithmetic over the reals or the integers. (1) Addition is closed, commutative, associative with identity 0; (2) multiplication is closed, commutative, associative with identity 1; (3) additive and multiplicative identities always exist; (4) multiplication and addition satisfy the distributive law.

Review of Simple Series

- Simple Series of integers (+ve) that are often useful in evaluating an algorithm: [there are many other kinds of series in mathematics though]
  1. 3, 7, 11, 15, 19, …
  2. 2, 6, 18, 54, 162, …
  3. 1, 4, 9, 16, 25, 36, 49, …
  4. 1, 1, 2, 3, 5, 8, 13, …
- Assume in a series the elements are named as a_1, a_2, a_3, … . Often, we need to compute the n^{th} element a_n as well as compute the sum $S_n = \Sigma_{i=1\ldots n} a_i$. An important observation is that an element depends on a finite number of immediately preceding elements (**finite history**).
  - In (1), the first number is 3 and $a_i = a_{i-1} + 4$ for all i; such series is known as **Arithmetic progression (A.P.)**. If the first number is x and the common difference is y, then $a_n = x + (n-1)y$ and the sum S_n of the first n numbers is $(n/2)\{2x + (n-1)y\}$. The set $\{a_i = a_{i-1} + 4, a_1=x\}$ is called the recurrence relation for the series; it precisely describes the series. A special case: if x = 1and y = 1, then $S_1(n) = n(n+1)/2$ gives the sum of all positive integers up to n.
  - In (2), the recurrence relation is $\{a_i = ra_{i-1}, a_1 = x = 2$, r (common ratio) $=3\}$ for all i; such series is known as **Geometric progression (G.P.)**. Here, $a_n = xr^{n-1}$ and S_n is given by

$$S_n = \begin{cases} nx, & \text{if } r = 1 \\ \dfrac{r^{n-1} - 1}{r - 1}, & \text{otherwise} \end{cases}$$

Compute GCD[*] *of 2 positive integers*

- If we were able to factor a and b into primes, computing gcd is easy. For example, if a = 230 895 588 646 864 = $2^4 \cdot 157 \cdot 4513^3$, b = 33 107 658 350 407 876 = $2^2 \cdot 157 \cdot 2269^3 \cdot 451^3$, gcd(a, b) = $2^2 \cdot 157 \cdot 4513$ = 2 834 164. But, factoring integers, especially large ones, is extremely expensive.
- The Euclidean algorithm essentially works because the map (a, b) → (a (mod b), b) for a ≥ b, is a gcd preserving mapping.

Facts:

1. A very good article can be found here. The literature is extensive.
2. Discuss Fibonacci numbers briefly in this context.
3. The Euclidean algorithm always needs less than O(h) divisions, where h is the number of digits in the smaller number b; it can be shown that #steps ≤ 5 $\log_{10}$b

```
function gcd(a, b)
while b ≠ 0
{t := b, b := a mod b, a := t}
return a
```

*Greatest Common Divisor

Binary Euclid Algorithm

- The Euclidean algorithm essentially works because the map (a, b) → (a (mod b), b) for a ≥ b, is a gcd preserving mapping. The trouble is that computers find it much easier to add and multiply numbers than to take remainders or quotients. Let us consider another gcd preserving mapping:

$$(a,b) \mapsto \begin{cases} ((a-b)/2, b) & \text{if a and b are odd.} \\ (a/2, b) & \text{if a is even and b is odd.} \\ (a, b/2) & \text{if a is odd and b is even.} \end{cases}$$

- Recall that computers find it easy to divide by two, since in binary this is accomplished by a cheap bit shift operation. This latter mapping gives rise to the binary Euclidean algorithm, which is the one usually implemented on a computer. Essentially, this algorithm uses the above gcd preserving mapping after first removing any power of two in the gcd. We assume a and b are 2 positive integers.

Binary Euclid's Algorithm to compute gcd

```
g = 1
/* Remove powers of two from the gcd */
while (a  mod 2 = 0) and (b  mod 2 = 0) do
    a = a/2
    b = b/2
    g = 2 · g
end
/* At least one of a and b is now odd */
while a ≠ 0 do
    while a  mod 2 = 0 do   a = a/2
    while b   mod 2 = 0 do   b = b/2
    /* Now both  a and b are odd */
          if a ≥ b then a = (a − b)/2
          else  b = (b − a)/2
end
return g · b
```

Power Summations

- Consider (3): 1, 4, 9, 16, 25, 36, 49, … i.e., the squares of the natural numbers. So, the n-th term is n^2 [recurrence relation, i.e., no history needed, only the index number of the term] and the sum of first n terms is $S_2(n) = \sum_{i=1 \text{ to } n} i^2$. Similarly, we can have series sums of the form $S_k(n) = \sum_{i=1 \text{ to } n} i^k$. There are two ways to solve such power series: (1) Using induction if we know the result; (2) using binomial theorem if we do not know the answer. We will illustrate with $S_2(n) = \sum_{i=1 \text{ to } n} i^2 = n(n+1)(2n+1)/6$

1. **Use induction**: Base case: the claim is obviously true when n = 1 since $1^2 = 1 = 1.2.3/6$. Induction hypothesis: the claim is true for n – 1 or $\sum_{i=1 \text{ to } n-1} i^2 = (n-1)n(2n-1)/6$. Induction Step: $S_2(n) = \sum_{i=1 \text{ to } n} i^2 = \sum_{i=1 \text{ to } n-1} i^2 + n^2 = (n-1)n(2n-1)/6 + n^2 = n(2n^2 – 3n + 1 + 6n)/6 = n(n+1)(2n+1)/6$
2. **Using binomial theorem** (when we do not know the result): Consider the binomial equation (cubic) after moving the leading term from RHS to LHS: $(i + 1)^3 – i^3 = 3i^2 + 3i + 1$. Write out the identity for all i, $1 \leq i \leq n$, and all the LHS and RHS to get the identity

$$(n +1)^3 – 1 = 3(1^2 + 2^2 + \ldots + n^2) + 3(1 + 2 + \ldots + n) + n.1$$

$$= 3S_2(n) + 3n(n+1)/2 + n$$

Rearranging, $S_2(n) = (2(n+1)^3 – 2 – 2n – 3n(n + 1))/6 = n(n + 1)(2n + 1)/6$

3. **Exercise: Prove $S_3(n) = \sum_{i=1 \text{ to } n} i^3 = n^2 (n^2 + 2n + 1)/4 = (S_1(n))^2$**

Fibonacci Series

- Consider the series: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, … [indexed starting from 1, i.e., F(1) = 1, F(2) = 1 and so on; sometimes if we index from 0, we stick a F(0) = 0 (does not change anything)]. What is F(n)? **{F(n) = F(n-1) + F(n-2), given F(1) = 1 and F(2) = 1}** $\Rightarrow$ recurrence relation; it says the any term can be computed from the knowledge of the immediately preceding two terms.
- Fibonacci was interested in population dynamics, studied population of rabbits under 3 assumptions: (1) a pair of rabbits has a pair of children every year. (2) These children are too young to have children of their own until two years later (3) Rabbits never die [not a realistic assumption; later he assumed the rabbits will die in 10 years; that's another story, outside of our scope in this course]. We start with 1 pair of rabbits – you get the drift.
- Simplest algorithm to compute F(n) is straightforward [and most inefficient]: int fib (int n) {if $n \leq 2$ return 1 else return (fib(n – 1) + fib (n – 2)))}. Short, simple, sweet!! Why inefficient? It's execution is exponential in n [$O(\varphi^n)$, where φ =golden ratio = $(\sqrt{5}+1)/2)$] because of same computations again and again, let alone cost of recursion stack processing so many times. Draw the tree, you will see empirically; we will do rigorously later.
- Fibonacci Series has many more diverse applications than population dynamics: stock market analysis, music synthesis, external sorting etc.

O(n) algorithm to compute F(n)

- **int fib2 (int n)**

```
{if n ≤ 2 return 1
  else  {int a=1, b=1; for (i=2; i ≤ n; i++)
            {int c = a + b; a = b; b = c};
  return c}
```

- Worst case and best-case time complexity are O(n) and space complexity is O(1).
- Can we have a logarithmic algorithm? Yes. Use the following identity

$$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^n = \begin{bmatrix}F(n+1) & F(n)\\ F(n) & F(n-1)\end{bmatrix}$$ Note that this is obviously true for n = 1. Use induction to prove

Assume $\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^n = \begin{bmatrix}F(n+1) & F(n)\\ F(n) & F(n-1)\end{bmatrix}$. Then $\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n+1} = \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^n \begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix} = \begin{bmatrix}F(n+1) & F(n)\\ F(n) & F(n-1)\end{bmatrix}\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}$

$$= \begin{bmatrix}F(n+1)+F(n) & F(n+1)+0\\ F(n)+F(n-1) & F(n)+0\end{bmatrix} = \begin{bmatrix}F(n+2) & F(n+1)\\ F(n+1) & F(n)\end{bmatrix}.$$

Note: If we use exponentiation by squaring, we get an O(log n) time and O(1) space algorithm with the caveat that this is counted in terms of the number of bigint arithmetic operations, not primitive fixed-width operations.

We will return to Fibonacci numbers when we do Fibonacci heaps.

Notes on Exponentiation

- We often need to compute x^n, for large integers n, in many applications (e.g., modular arithmetic in cryptography, additive semigroups like elliptic curves, powering of matrices, shortest path computations in large graphs); the simplistic O(n) algorithm of repeated multiplications is slow.
- First, we observe a basic fact: $$x^n = \begin{cases} x(x^2)^{\frac{n-1}{2}}, & \text{if n is odd} \\ (x^2)^{\frac{n-1}{2}}, & \text{if n is even} \end{cases}$$
- We can use this to design a logarithmic algorithm

```
Function EXP (x, n)
if n < 0 then {n = – n; x = 1/x}; [why?]
if n = 1 then return 1;
while n > 1 do {if n is even then {x = x*x; n = n/2}
                else {y = y*x; x = x*x; n = n/2}  }
return x*y;
```

- A brief analysis shows that such an algorithm uses $\lfloor \log_2 n \rfloor$ squaring and at most $\lfloor \log_2 n \rfloor$ multiplications. More precisely, the number of multiplications is one less than the number of ones present in the binary expansion of *n*. For *n* greater than about 4 this is computationally more efficient than naively multiplying the base with itself repeatedly. We want to remember that each squaring results in approximately double the number of digits of the previous, and so, if multiplication of two *d* digit numbers is implemented in $O(d^k)$ operations for some fixed *k* then the complexity of computing x^n is given by $O((n \log(x))^k)$.

How to compute a square root?

- **Without a calculator, of course!!** First, we observe that √x, when x is not a square, is an irrational number; when we compute √x, we do so up to a n^{th} decimal place, n = 1, 2, … Newton-Raphson method is such an approximation algorithm: (1) Select an estimate, say y, (arbitrary); (2) divide the radicand by the estimate; (3) get the average of the estimate and the quotient, say q, in (2) to get the new estimate (y + q)/2 and repeat.
- **Example**: x = 45; R1: y = 6, q = 7.5 (R2) y = (6+7.5)/2 = 6.75, q = 45/6.75 = 6.66 (R3) y = 6.705, q = 6.711 (R4) y = 6.708, q = 6.7084 and so on as long as we want. If we assume that say is error < 0.001, we know where to stop. **Write a program** to compute the square root of any positive integer. Note: In general, this method quickly converges and does so even more quickly if you choose the initial estimate judiciously – *choose the largest integer whose square is less than equal to the radicand.*
- There are many other algorithms to do this with different convergence times under different scenarios; check here to see how we used to do it in our school days without any calculator or computers. If you are more intrigued or want to know how to prove that the algorithms really converge for all cases, check Wikipedia and references therein.

An irrational number is a type of real number which cannot be represented as a simple fraction. It cannot be expressed in the form of a ratio

o- and ω- Notations

- Let f(n) and g(n) be functions mapping integers to real numbers. The asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. We say that f(n) is **o(g(n))** (pronounced "f(n) is **little-oh** of g(n)") if, for any constant c > 0, there is a constant $n_0 > 0$ such that $f(n) \leq cg(n\}$ for $n \geq n_0$. The definitions of O-notation and o-notation are almost similar; Intuitively, in o-notation, the function f (n) becomes insignificant relative to g(n) as n approaches infinity; that is, $\mathbf{lim_{n\to\infty}}$ **(f(n)/g(n)) = 0**
- Likewise, we say that f(n) is **ω(g(n))** (pronounced "f(n) is **little-omega** of g(n)") if g(n) is o(f (n)), that is, if, for any positive constant c > 0, there is a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for $n \geq n_0$. For example, $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$. The relation f(n) = ω(g(n)) implies that $\mathbf{lim_{n\to\infty}}$ **(f(n)/g(n)) =** ∞**,** if the limit exists.
- Intuitively, o(·) is analogous to "less than" in an asymptotic sense, and ω(·) is analogous to "greater than" in an asymptotic sense. Examples: f(n) = 2n +6 is $o(n^2)$ and ω(n).
- **Comparing functions**: Many of the relational properties of real numbers apply to asymptotic comparisons as well. For the following, assume that f (n) and g(n) are asymptotically positive.

Comparing Functions

Transitivity:

$$
\begin{aligned}
f(n) &= \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) \text{ imply } f(n) = \Theta(h(n)) ,\\
f(n) &= O(g(n)) \text{ and } g(n) = O(h(n)) \text{ imply } f(n) = O(h(n)) ,\\
f(n) &= \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \text{ imply } f(n) = \Omega(h(n)) ,\\
f(n) &= o(g(n)) \text{ and } g(n) = o(h(n)) \text{ imply } f(n) = o(h(n)) ,\\
f(n) &= \omega(g(n)) \text{ and } g(n) = \omega(h(n)) \text{ imply } f(n) = \omega(h(n)) .
\end{aligned}
$$

Reflexivity:

$$
\begin{aligned}
f(n) &= \Theta(f(n)) ,\\
f(n) &= O(f(n)) ,\\
f(n) &= \Omega(f(n)) .
\end{aligned}
$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n)) .$$

Transpose symmetry:

$$
\begin{aligned}
f(n) &= O(g(n)) \text{ if and only if } g(n) = \Omega(f(n)) ,\\
f(n) &= o(g(n)) \text{ if and only if } g(n) = \omega(f(n)) .
\end{aligned}
$$

Because these properties hold for asymptotic notations, we can draw an analogy between the asymptotic comparison of two functions f and g and the comparison of two real numbers a and b:

$$
\begin{aligned}
f(n) &= O(g(n)) \text{ is like } a \le b ,\\
f(n) &= \Omega(g(n)) \text{ is like } a \ge b ,\\
f(n) &= \Theta(g(n)) \text{ is like } a = b ,\\
f(n) &= o(g(n)) \text{ is like } a < b ,\\
f(n) &= \omega(g(n)) \text{ is like } a > b .
\end{aligned}
$$

We say that $f(n)$ is ***asymptotically smaller*** than $g(n)$ if $f(n) = o(g(n))$, and $f(n)$ is ***asymptotically larger*** than $g(n)$ if $f(n) = \omega(g(n))$.

An Interesting Observation

- One property of real numbers, however, does not carry over to asymptotic notation:

  Trichotomy: For any two real numbers a and b, exactly one of the following must hold: $a < b$, $a = b$, or $a > b$.

- Although any two real numbers can be compared, not all functions are asymptotically comparable. That is, for two functions f(n) and g(n), it may be the case that neither $f(n) = O(g(n))$ nor $f(n) = \Omega(g(n))$ holds. For example, we cannot compare the functions n and $n^{1+\sin n}$ using asymptotic notation, since the value of the exponent in $n^{1+\sin n}$ oscillates between 0 and 2, taking on all values in between.

Some Examples

- In each of the following situations, indicate whether f = O(g), or f = Ω(g), or both (in which case f = Θ(g).

	$f(n)$	$g(n)$
(a)	$n - 100$	$n - 200$
(b)	$n^{1/2}$	$n^{2/3}$
(c)	$100n + \log n$	$n + (\log n)$
(d)	$n \log n$	$10n \log 10n$
(e)	$\log 2n$	$\log 3n$
(f)	$10 \log n$	$\log(n^2)$
(g)	$n^{1.01}$	$n \log^2 n$
(h)	$n^2 / \log n$	$n(\log n)^2$
(i)	$n^{0.1}$	$(\log n)^{10}$
(j)	$(\log n)^{\log n}$	$n / \log n$
(k)	$\sqrt{n}$	$(\log n)^3$
(l)	$n^{1/2}$	$5^{\log_2 n}$
(m)	$n2^n$	3^n
(n)	2^n	2^{n+1}
(o)	$n!$	2^n
(p)	$(\log n)^{\log n}$	$2^{(\log_2 n)^2}$
(q)	$\sum_{i=1}^{n} i^k$	n^{k+1}

Some useful hints:

- $y \cdot (\log n)^k$ is $O(n^\varepsilon)$ for any given constants y and k, and any $\varepsilon > 0$; we need to show that there exists two constants c and n_0 such that $y \cdot (\log n)^k \leq c(n^\varepsilon)$ for any $n > n_0$. Take log of both sides to get $\log y + k \log \log n \leq \log c + \varepsilon \log n$