Asymptotic Notations & Complexity

O = upper bound, o = loose upper bound, Ω = lower bound, Θ = upper and lower bound.

$f(n) = O(g(n))$ if $f(n) \leq cg(n)$
$f(n) = \Theta(g(n))$ if $f(n)$ lies between $c_1g(n)$ and $c_2g(n)$, implies $O(g(n))$ and $\Omega(g(n))$
$f(n) = \Omega(g(n))$ if $f(n) \geq cg(n)$
$f(n) = o(g(n))$ if upper bound is not tight
E.g. $f(n) = O(g(n))$ means f is upper bounded by g.

A	B	O	o	Ω	Θ
$\log^k n$	n^ε	Yes	Yes	No	No
n^k	c^n	Yes	Yes	No	No
$\sqrt{n}$	$n^{sin(n)}$	No	No	No	No
2^n	$2^{n/2}$	No	No	Yes	No
n^{logc}	c^{logn}	Yes	No	Yes	Yes
$log(n!)$	$log(n^n)$	Yes	No	Yes	Yes

Listed in increasing order:
1, $\log^* n$, loglog n, $\sqrt{logn}$, $log^2 n$, log n, $\sqrt{n} = \sqrt{2}^{logn}$, n, n log $n = log(n!)$, $n^2 = 4^{logn}$, n^3, $(3/2)^n$, $2^n = 2^{n-1}$, $n2^n$, $\cdots$

1	constant time
log n	logarithmic
n	linear (also n log n)
n^2	quadratic
2^n	exponential
anything before n	sub-linear

What is the upper bound of each function?
$\frac{(nlogn)}{2} + f(n)$, where $f(n) = o(nlog(n^{100})) \rightarrow O(n \log n)$
$1 + 3 + 5 + ... + (n-1)$, where n is even $\rightarrow O(n^2)$
$log^2 n + log(n^3) \rightarrow O(log^2 n)$
$4^{logn} \rightarrow O(n^2)$
$1 + 1/2 + 1/2^2 + ... + 1/2^n \rightarrow O(1)$
$n^1 + n^2 + n^3 + ... + n^k + 2^n$, where $k > 0 \rightarrow O(2^n)$
$4n^{3/4} + 5n \log n + 2n \log\log n \rightarrow O(n \log n)$
$2010 + sin(n) \rightarrow O(1)$
$t(n) = 2t(n-1) + 1$, $t(1) = 1 \rightarrow O(2^n - 1)$ Tower of Hanoi
$(n^2 - 1)/(n + 1) \rightarrow O(n)$

Solving Recurrence Relations

1. Iterations

Continue the replacement procedure until $T(1)$ is the only value of T on the RHS. A pattern will emerge.

Example 1: Solve the recurrence $T(1) = 1$,
$T(n) = 3T(n-1) + 2$
$\cdots = 3(3T(n-2) + 2) + 2$, substitution for T(n-1)
$\cdots = 3^2T(n-2) + 6 + 2$
$\cdots = 3^2(3T(n-3) + 2) + 6 + 2$, substitution for T(n-2)
$\cdots = 3^3T(n-3) + 18 + 6 + 2$
$\cdots$
$\cdots = 3^kT(n-k) + 3^k - 1$, thus $k = n - 1$
$\cdots = 3^{n-1}T(n-(n-1)) + 3^{n-1} - 1$
$\cdots = 3^{n-1}T(1) + 3^{n-1} - 1$, where T(1) = 1
$\cdots = 3^{n-1} + 3^{n-1} - 1$
$\cdots = O(3^{n-1})$ (optional)

Example 2: Solve the recurrence $T(1) = c$, $T(n) = 2T(\frac{n}{2}) + cn$
$\cdots = 2(2T(\frac{n}{4}) + c\frac{n}{2}) + cn$, substitution for T(n/2)
$\cdots = 2^2T(\frac{n}{4}) + 2cn$
$\cdots = 2^2(2T(\frac{n}{8}) + c\frac{n}{4}) + 2cn$, substitution for T(n/4)
$\cdots = 2^3T(\frac{n}{8}) + 3cn$
$\cdots$
$\cdots = 2^kT(1) + kcn$, where $n = 2^k$
$\cdots = n * c' + kcn$, because $n = 2^k$ and $T(1) = c'$
$\cdots = nc' + cn \log n$, because $k = log_2 n$
$\cdots = O(n \log n)$

2. Substitution

Guess a solution (i.e. runtime), then prove by induction.

Example 1: $t(n) = \sum_{i=1}^{k} t(a_in) + n$ where $\sum_{i=1}^{k} a_i < 1$
Guesstimate $t(n) = O(n)$.
Proof: assume $t(a_in) \leq ca_in$ for $i = 1, 2, ..., k$. We need to show that $t(n) \leq cn$:
$t(n) = \sum_{i=1}^{k} t(a_in) + n$
$\cdots \leq \sum_{i=1}^{k} ca_in + n$
$\cdots \leq cn\sum_{i=1}^{k} a_i + n$, when reorganized
$\cdots \leq c\alpha n + n$, where $\alpha = \sum_{i=1}^{k} a_i$
$\cdots \leq cn$, because $\alpha < 1$

Example 2: Merge Sort $t(n) = 2t(\frac{n}{2}) + n$, $t(1) = 1$
Guesstimate $t(n) = O(n \log n)$.
Prove that $t(n) \leq cn \log n$ for some c.
Inductive base: prove the inequality holds for some small n.
$n = 1 \rightarrow t(1) \leq c * 1 * log * 1$? No.
$n = 2 \rightarrow t(2) \leq c * 2 * log * 2$? Yes, for any $c \geq 2$.
Induction assumption: assume the bounds hold for $n/2$.
$t(n) = 2t(\frac{n}{2}) + n$
$\cdots \leq 2c\frac{n}{2}log\frac{n}{2} + n$
$\cdots \leq cn \log \frac{n}{2} + n$
$\cdots \leq cn(\log n - \log 2) + n$
$\cdots \leq cn \log n - cn + n$
$\cdots \leq cn \log n$, holds if $c \geq 1$

3. Master Theorem

To solve a problem of size n, divide it into sub-problems of size $\frac{n}{b}$ each. The time to divide and/or combine the solutions is $f(n)$.
Must be of the form $T(n) = aT(\frac{n}{b}) + f(n)$, then you compare $f(n) : n^{log_b a}$.
case 1: $f(n) = O(n^{log_b a - \varepsilon})$, then $T(n) = \Theta(n^{log_b a})$.
case 2: $f(n) = \Theta(n^{log_b a})$, then $T(n) = \Theta(n^{log_b a}logn)$.
case 3: $f(n) = \Omega(n^{log_b a + \varepsilon})$ AND if $af(\frac{n}{b}) \leq cf(n)$ for $c < 1$ and large n, then $T(n) = \Theta(f(n))$.
For case 1 & 2: $\varepsilon > 0$
Check case 1 if $f(n)$ is smaller, case 2 if $f(n)$ is equal, case 3 if $f(n)$ is greater.

Example 1: Solve recurrence $T(n) = 3T(n/2) + n$ log

n, using the master method.
$a = 3$, $b = 2$, $f(n) = n \log n$, $n^{log_b a} = n^{log_2 3} = n^{1.58}$
$f(n) < n^{log_b a}$, thus $f(n) = O(n^{log_2 3-\varepsilon}) \rightarrow$ Case 1. $T(n) = \Theta(n^{log_2 3})$.

Example 2: $t(n) = t(2n/3) + 1$
$a = 1$, $b = 3/2$, $f(n) = 1$, $n^{log_b a} = n^{log_{3/2} 1} = n^0 = 1$
Compare $f(n) : n^{log_b a} \rightarrow 1 == 1$. Equal, thus Case 2.
So $t(n) = \Theta(n^{log_b a} \log n) = \Theta(1 \log n) = \Theta(\log n)$

Example 3: $t(n) = 3t(n/4) + n \log n$
$a = 3$, $b = 4$, $f(n) = n \log n$, $n^{log_b a} = n^{log_4 3} = n^{0.79}$
$f(n) > n^{log_b a}$, thus verify the regularity condition: $af(\frac{n}{b}) \leq cf(n)$ for $c < 1$ and large n.
$3\frac{n}{4} \log \frac{n}{4} \leq cn \log n$? Yes. Thus Case 3 $\rightarrow t(n) = \Theta(f(n)) = \Theta(n \log n)$.

4. Decision/Recursion Tree
Visualize the iteration. Write down all the work it has to do, then sum it up.

Example 1: $t(n) = \sum_{i=1}^{k} t(a_i n) + n$ where $\sum_{i=1}^{k} a_i < 1$
The root is n, its k children are $\{a_1 n, a_2 n, \cdots, a_k n\}$, and whose children, in turn, are $\{a_1 a_1 n, a_1 a_2 n, \cdots, a_1 a_k n\}, \cdots, \{a_k a_1 n, a_k a_2 n, \cdots, a_k a_k n\}$, and so on.
To make notation easier, let $\sum_{i=1}^{k} a_i = \alpha$. Then each level has the following amount of work to do: root $\rightarrow n$, $2^{nd} level \rightarrow \alpha n$, $3^{rd} level \rightarrow \alpha^2 n$, and so on.
So the total work on all levels is:
$t(n) = n + \alpha n + \alpha^2 n + \cdots$
$\cdots = n(1 + \alpha + \alpha^2 + \cdots)$
$\cdots$ remember that $\alpha = \sum_{i=1}^{k} a_i < 1$, thus
$\cdots = O(n)$

Example 2: $t(n) = t(\frac{n}{2})t(\frac{n}{2}) + n^2 = 2t(\frac{n}{2}) + n^2$
The root is n^2, its 2 children are $\{(n/2)^2, (n/2)^2\}$, and all their children are $(n/2/2)^2 = (n/2^2)^2$, and so on.
Total work on all levels is:
$t(n) = n^2 + \frac{n}{2}^2 + \frac{n}{2^2}^2 + \cdots$
$\cdots = n^2(1 + \frac{1}{2} + \frac{1}{2^2} + \cdots)$
$\cdots = n^2 * 2$, pattern converges to 2
$\cdots = O(n^2)$

Example 3: $t(n) = t(\frac{n}{10}) + t(\frac{9n}{10}) + n$
The root is n, its 2 children are $\{\frac{n}{10}, \frac{9n}{10}\}$, and their children are $\{\frac{n}{10^2}, \frac{9n}{10^2}\}, \{\frac{9n}{10^2}, \frac{9^2 n}{10^2}\}$, and so on.
This is an unbalanced tree; the left-most side is a much shorter path to 1 than the right-most path. To find out the total work, we have to use the maximum height of the tree.
LHS: $\log_{10} n$. RHS: $(\frac{9}{10})^k n \leq 1, n \leq (\frac{10}{9})^k), k \approx \log_{\frac{10}{9}} n$
Every level has work n. We just need to figure out how many n's the longest path is.
$t(n) = n *$max height
$\cdots = n \log_{\frac{10}{9}} n$
$\cdots = O(n \log_2 n)$

Homogeneous Linear Recurrence w/ Constant Coefficients
Example 1: Solve $f(n) = 2f(n-1) - f(n-2)$, for $n > 1$, and $f(0) = 1$, $f(1) = 1$.

Characteristic equation is $x^2 - 2x + 1 = 0$ with $x = 1$ as a double root.
General solution is $f(n) = c_1 1^n + c_2 n 1^n = c_1 + c_2 n$.
Base cases: $f(0) = c_1 = 1$, and $f(1) = c_1 + c_2 = 1$
which means $c_1 = 1$ and $c_2 = 0$. Thus $f(n) = 1$.

Example 2: Solving it my way
1. Look for a simple solution of polynomial form cr^n where c is constant and $c \neq 0$
2. Plug it in: $cr^n = 2cr^{n-1} - cr^{n-2}$
3. Divide by the lowest term, which is cr^{n-2}:
$r^2 = 2r - 1$
$r^2 - 2r + 1 = 0$
4. Solve for r:
$r = 1$
5. General solution: $f(n) = cr^n$
6. Base cases:
$f(0) = cr^0 = c1^0 = 1$
$f(1) = cr^1 = c1^1 = 1$
7. Solve constants: $c = 0$
8. The recurrence equation solution is $f(n) = 1^n = 1$

Dynamic Programming
For optimization problems, solve smaller problems and save their solutions. A solution to a bigger problem uses solutions from a smaller problem if there is a Principle of Optimality or Optimal Sub-structure, i.e. In an optimal sequence of decisions, each subsequence is also optimal.

1) Verify that the principle of optimality holds
2) Come up with a recurrence that expresses the solution for the problem of size i in terms of the solution for problems of size $i-1, i-2, \cdots$
NOTE: Even though it's a recurrence relation, you do not solve it recursively, defeats the purpose of DP.
NOTE: Doesn't always guarantee best solution (sometimes it's not most efficient), but it does beat brute force.

Example 1: Longest Increasing Subsequence (LIS)
Let S be a sequence of n distinct integers $S[1..n]$.
e.g.) $11, 17, 5, 8, 6, 4, 7, 12, 3$ then the LIS is $5, 6, 7, 12$.
1) Show that it satisfies the principle of optimality: If you have the best path, a subpath of it will also be optimal.
2) Define a proper function and find the recurrence for this function:
Let C_i be the length of a LIS in $S[1..i]$ ending at $x_i = S[i]$, $1 \leq i \leq n$. This means that $S[i]$ is the last element in the LIS in $S[1..i]$. The recurrence is:
$C_1 = 1$

$C_i = max\{C_k + 1_{1 \leq k \leq i1}$, if $S[i] > S[k]$
$C_i = max\{1$ otherwise
We first compute $C_1, C_2, ..., C_n$, then find $\max_{1 \leq i \leq n}\{C_i\}$. Computing C_i takes $O(i)$ time. Thus it takes $O(n^2)$ time to get all $C_i s$. Finding the max requires $O(n)$ time. Therefore, the total time is $O(n^2)$.

Example 2: Change-making problem
Given an amount n and unlimited qty of coins, each of the denominations $d_1, d_2, ..., d_m$, find the smallest number of coins that add up to n or indicate that the problem does not have a solution.
1) Principle of optimality & proper function:
Define $C(x)$ to be the smallest number of coins for value x. If the change includes a coin of denomination d_i, then clearly $C(x - d_i)$ is the smallest number of coins for value $x - d_i$.
2) Recurrence of this function:
$C(d_1) = 1$
$C(d_2) = 1$
$\cdots$
$C(d_m) = 1$
$C(x) = 1$ if $x = d_1, d_2, .., d_m$
$C(x) = \min_{1 \leq i \leq m}\{C(x - d_i) + 1\}$ if $C(x - d_i)$ exists
$C(x) =$ No solution otherwise
The final answer is in $C(n)$.

String Matching

Useful in DNA, text search (editors, web search), etc.
We have text $T[1..n]$ and pattern $P[1..m]$, where $m \leq n$. If there exists an s ($0 \leq s \leq n - m$) such that $T[s + 1..s + m] = P[1..m]$, then s is a valid shift.

For example

T	a	b	c	d	a	b	c	d	a	b
	1	2	3	4	5	6	7	8	9	10
P	d	a	b							

Valid shifts: $s = 3$ and $s = 7$.

Naive String Matcher (Brute Force)

for $s = 0$ to $n - m$ **do** $\{O(n - m + 1)\}$
  if $P[1..m] == T[s + 1..s + m]$ **then** $\{O(m)\}$
    print "Valid shift s =" s
  end if
end for

$\Rightarrow O((n - m + 1)m) = O(nm)$. This is a very tight bound.

Rabin-Karp Matcher

The idea is to treat strings of characters as radix-d digits, because numbers can be compared in constant time (unless very large). Then compute the value of pattern $P : p \rightarrow O(m)$. Then compute the values of T as $t_0, t_1, t_2, ..., t_{n-m} \rightarrow O(n)$. Compare p with $t_0, t_1, t_2, ..., t_{n-m} \rightarrow O(n)$.

Alphabet Σ, size of alphabet: $|\Sigma| = d$.
E.g.) $|\Sigma| = 10$ (radix-10 is decimal), $m = 5$, $P = 314152$.
$P[1..m] = P[1] * 10^{m-1} + P[2] * 10^{m-2} + \cdots + P[m] \rightarrow O(n)$.
Then compute t_0 in $O(m)$ time, same as p.
How to get t_{s+1} if we have t_s? That is, $t_s = 31415$ to $t_{s+1} = 14152$. Then $t_{s+1} = (31415 - 30000) * 10 + 2$.
In general, $t_{s+1} = (t_s - T[s+1] * 10^{m-1}) * 10 + T[s+m+1]$. This is done in $O(1)$ time, because 10^{m-1} is precomputed in the $O(m)$ step.

For very large numbers: $p == t_s$?
do: $p \bmod q == t_s \bmod q$
If the modded result is not equal, then neither is the other.

Finite State Machine

Linear string matching using a FSM. Each FSM has a finite set of states Q, set of alphabet symbols Σ, set of final states F, and transition functions δ.

Consider a FSM that accepts strings that contain a substring AABC. Our alphabet is {A,B,C}. It has 5 states: q_0, q_1, q_2, q_3, q_4, and q_4 is the final state.
$\delta(q_0, A) = q_1$ and $\delta(q_0, \{B, C\}) = q_0$
$\delta(q_1, A) = q_2$ and $\delta(q_1, \{B, C\}) = q_0$
$\delta(q_2, A) = q_2$ and $\delta(q_2, B) = q_3$ and $\delta(q_2, C) = q_0$
$\delta(q_3, A) = q_1$ and $\delta(q_3, B) = q_0$ and $\delta(q_3, C) = q_4$
$\delta(q_4, \{A, B, C\}) = q_4$

Approximate String Matching

Need to measure the distance between Text and Pattern. Distance is measured with 3 char operations:
1) replace (cost 1)
2) insert (cost 1)
3) delete (cost 1)
Convert T to P using the 3 operations. We want the solution with the minimum cost (this distance is called "edit" or "Levenshtein distance"). We use DP to solve this.

$c(i, j)$: min cost of converting $t_1, t_2, ..., t_i$ to $p_1, p_2, ..., p_j$.
$c(n, m)$: edit distance (i.e. min distance)
initial conditions:
$c(i, 0) = i$ (deleting all to empty string)
$c(0, j) = j$ (inserting all)
$c(i, j) = \{c(i - 1, j) + 1$ (delete t_i)
$c(i, j) = \{c(i, j - 1) + 1$ (insert p_j)
$c(i, j) = \{c(i - 1, j - 1) + 1$ (replace t_i with p_j)
$c(i, j) = \{c(i - 1, j - 1)$ (do nothing, $t_i == p_j$)

Knuth-Morris-Pratt (KMP) Algorithm

KMP shifts as far as possible, in comparison with naive method which shifts one position. This is done by looking at the largest prefix equal to suffix. A proper prefix is not empty. Note: The table must start at -1.
Tip: cover the next column with your hand and look at prefix & suffix.

i	1	2	3	4	5	6	7	8	9	10
P[i]	y	o	y	o	m	a	y	o	y	o
next(i)	-1	0	0	1	2	0	0	1	2	3

Greedy Algorithms

Greedy is applying a "greedy" idea to solve an optimization problem. Optimization problems are very hard (NP-Complete or NP-Hard).

Prove Greedy doesn't work

Greedy is not guaranteed to give optimal solution. Try to find a solution that may not be optimal, but better than greedy.

Example 1: Chained matrix multiplication problem

You find an optimal order by which to calculate $M = M_1 \times M_2 \times \cdots \times M_n$, we know that we can use the technique of DP to solve the problem. For each of the following suggested greedy ideas, provide a counter example where the greedy solution does not work:

(a) Multiply the matrices whose common dimension r_i is smallest first.

Consider $M_{2\times 1} \times M_{1\times 2} \times M_{2\times 3}$:

•Greedy solution: $((M_{2\times 1} \times M_{1\times 2}) \times M_{2\times 3})$: cost $= (2*3*(2*1*2)+(2*1*2)) = 16$

•Another solution $(M_{2\times 1} \times (M_{1\times 2} \times M_{2\times 3}))$: cost $= (2*1*(1*2*3)+(1*2*3)) = 12$

(b) Multiply matrices whose common dimension r_i is largest first.

Consider $M_{1\times 2} \times M_{2\times 3} \times M_{3\times 4}$:

•Greedy solution: $(M_{1\times 2} \times (M_{2\times 3} \times M_{3\times 4}))$: cost = 32

•Another solution $((M_{1\times 2} \times M_{2\times 3}) \times M_{3\times 4})$: cost = 18

Prove Greedy does work

One way to prove the correctness of a greedy algorithm is to prove the greedy choice property and optimal substructure property.

TODO

Non-deterministic Algorithms

1) Non-det. phase: allowed to guess at each step. If a solution exists, this part will always guess correctly a solution called "certificate".

2) Det. phase: it takes the certificate from phase 1 and verifies deterministically that it is indeed a solution.

Because we're only interested in decision version of the problem, if a certificate exists, it means the answer to the problem is Yes.

Example 1:

TODO

(Polynomial) Reduction

Reducing a problem because the original problem has already been solved.

To show problem A is NPC:

1) A $\in$ NP

2) A is NP-hard

$Q_1 \leq_p Q_2$ means Q_2 is at least as hard as Q_1!

Example 1: Reduce the problem of LIS to the problem of finding a longest path in a weighted DAG (directed acyclic graph).

Given an instance of LIS: distinct integers: $x_1, x_2, ..., x_n$ we create an instance of a longest path problem in a DAG as follows: DAG = (V, E) where $V = x_1, x_2, ..., x_n$ and (x_i, x_j) is an edge if

(a) $i < j$ and

(b) $x_i < x_j$

Clearly, a longest path in the DAG gives us an longest increasing subsequence.

Example 2: Given the decision version of the following two problems, show that SSP reduces to KP:

Subset Sum Problem:

Given a set S of (non-negative) integers $s_1, s_2, \cdots, s_n$ and b, is there a subset of these numbers with a total sum of b?

0-1 Knapsack Problem:

Given weights $w_1, w_2, \cdots, w_n$ and values $v_1, v_2, \cdots, v_n$ and k, is there a subset of weights with total weights at most b such that the corresponding profit (i.e. total value) is at least k?

1. Show that knapsack is in the NP class.

Given a certificate, check if the total weight is at most b and if the corresponding profit is at least k. This takes linear time to add the weights and profits of all the goods to find the true/false result.

2. Show that a problem which is known to be NP-Complete (in this case the Subset Sum Problem) can be reduced to the Knapsack problem in polynomial time.

Subset Sum Problem	$\leq_p$	Knapsack Problem
$\{s_1, s_2, \cdots, s_n\}$		$\{w_1, w_2, \cdots, w_n\}$, $\{v_1, v_2, \cdots, v_n\}$
		$w_i = v_i = s_i$
		$k = b$

Example 3:

Hamiltonian-cycle problem	$\leq_p$	TSP
$G = (V, E)$		$K_n = (V, E) \leftarrow$ complete graph
		$k = n$

TSP has a Yes answer iff H-cycle has a Yes answer.

Decision Problems vs Optimization Problems

Many optimization problems can be made into decision problems, which are much easier to solve (yes–no anwers).

Optimization version $\Rightarrow$ Decision version – Always

If the optimization version is solvable in polynomial time, the decision version is also.

Optimization version $\Leftarrow$? Decision version – Not always

Linear Selection

TODO

Loss of Generality
TODO

Logarithms
$2^k = n \rightarrow k = log_2 n$
$log_b a = \frac{log_{b'} a}{log_{b'}} b$
$b^{log_b a} = a$
$log(a) \times log(b) = log(a \times b)$
$log(a) - log(b) = log(\frac{a}{b})$
$log(a^b) = blog(a)$

Arbitrary
•Any tree with m leaves will have a height $\Omega(\log m!) = \Omega(m \log m)$.
There is no comparison-based sorting algorithm whose running time is linear for at least half of the n! inputs. Its decision tree would have a subtree with $n!/2$ leaves whose height is $O(n)$, which is wrong because the height would really be $log(n!/2) = \Theta(n \log n)$.

•$\alpha + \beta = 1$

•Master Method doesn't apply to $T(n) = 2T(n/2) + n \log n$.
$f(n) > n^1$, so check Case 3. BUT! regularity condition fails: $f(n) \neq \Omega(n^1 + \varepsilon)$!
Still solveable, but takes more work. Look at the recursion tree.

```
   root    input size n, work n log n.
   /  \
  x    x   input size n/2, work 2 * n/2 log n/2
 / \  / \                       = n log n/2
x  x  x  x input size n/4, work 4 * n/4 log n/4
                                = n log n/4

  etc.
```

So the total work is
$\sum_{i=1}^{log(n)} n \log (n/2^i)$
$= n \sum_{i=1}^{log(n)} (logn - 2^i)$
$= n(log^2 n - \sum_{i=1}^{logn} log(2^i))$
$= n(log^2 n - \sum_{i=1}^{logn} i)$
$= n(log^2 n - (log^2 n)/2)$
$= \Theta(nlog^2 n)$

•Horner's Rule
Compute $a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$
$\Rightarrow (\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0$
The idea is to multiply, add, multiply, add, etc.

•Complexity
Tractable: efficient algorithms (polynomial time, eg $n, logn, n^3$)
Intractable: inefficient algorithms (super polynomial, eg 2^n)
1) Most algorithms are lower order polynomials
2) A polynomial algorithm runs in polynomial time regardless of the computational model
3) Closure property: multiple polynomial algorithms in sequence still results in polynomial time

•NP: the set of decision problems solvable in polynomial time *non-deterministically.*
•P: the set of decision problems solvable in polynomial time *deterministically.*

NP: Loosely speaking, NP contains those problems whose solutions can be verified in polynomial time deterministically.
P: Simply a special case of NP.

•By definition $P \in NP$

NP-Complete problems are the hardest problems in NP, in that if it is solvable in polynomial time deterministically, then *all* problems in NP are solvable in polynomial time deterministically.
That is, <u>all problems in NP are polynomially reducable to it</u>.

•Fractional Knapsack: Greedy chooses heighest value to weight ratio first.