CS3230

Lecturer: Sanjay Jain (COM2 03–59; sanjay@comp.nus.edu.sg)

Tutors:

Ziquan Lan ziquan111@gmail.com ziquan@comp.nus.edu.sg

Satyam satyam@comp.nus.edu.sg

Ruomu houruomu@gmail.com

Web Page: http://www.comp.nus.edu.sg/∼sanjay/cs3230.html

Text Book: R. Johnsonbaugh and M. Schaefer — Algorithms.

CS3230

Tutorials:

- start next week.
- Questions for tutorial will be given in class (previous week). Please submit your tutorials in IVLE in the appropriate folder. Please hand in tutorials by sunday 8PM before the week of the tutorial. Please mention your matriculation number and tutorial group in the submission.
  G5_A1234567X_Tut4_Name.pdf/doc

Prerequisites CS1010 and (CS2010 or CS2020 or CS2040 or CS2040C) and (CS1231 or MA1100)
Basically need to know substantial programming, data structures, discrete mathematics, mathematical maturity (to follow proofs, come up with proofs and write proofs etc).

Grading:

- Tutorials and Class Participation (10 %)
- Assignments (20 %)
- 1 midterm (20 %) March 13, 2018, 6PM
- 1 final (50 %)

Acknowledgements:
Thanks to Yu Haifeng, David Hsu, Lee Wee Sun and Roger Zimmermann, who taught this course in previous years.

This lecture is based on topics covered in the textbook in:
Chapter 1, 2, 3

Algorithms

An Algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

Algorithms

- Input, what are the legitimate input
- Output, what are the valid outputs based on the input; most often unique output depending on the input.
- Unambiguous and Executable: The steps (instructions) are precisely and unambiguously stated; Instructions can be “executed", followed by a “machine”, clear how to implement them
- *Deterministic: Execution of each step is unique and gives a result determined only by the input to the step. [Randomness, nondeterminism, ..., Quantum]
- *Termination: The algorithm terminates after finitely many instructions are executed

- Generality: Applicable to a wide range of inputs
- Correctness:
- Performance/Efficiency/Complexity:
  Time and Space (Memory)
  (Depends on input size and content, machine speed, ...)
- Number of Basic Operations (correlated with time)
- Asymptotic Analysis
- Worst Case
- Average Case

Pseudocode

We can give a program. This is usually unambiguous and clear.
However, this is tedious. Sometimes difficult to understand.

Pseudocode.
Slightly informal.
Still precise enough to understand exactly what steps are taken.

Example: GCD (Greatest Common Divisor)
Input: Two integers, at least one of them $\neq 0$
Output: greatest natural number > 0 which divides both of them

Notation: $\mathbb{N}$ denotes the set of natural numbers $\{0, 1, 2, \ldots\}$
Notation: $\mathbb{Z}$ denotes the set of integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
Notation: $a \bmod b$ denotes the remainder when a is divided by b.

$GCD(m,n) = GCD(m,-n)$
$GCD(m,0) = m$, for $m > 0$

```
$GCD(m,n)$
    $m, n$ are natural numbers $> 0$.
    Let $t = min(m,n)$.
    Loop
        If $t$ divides both $m$ and $n$, then output $t$
        Else $t = t - 1$
    Forever
```

Euclid's Algorithm

Uses: If $m = qn + r$, where m, n, q, r are natural numbers, then $gcd(m, n) = gcd(n, r)$

```
GCD(m,n)
(* $m, n$ are natural numbers, at least one of them $> 0$ *)
    While $n > 0$ Do
        $r = m \bmod n$
        $m = n$
        $n = r$
    EndWhile
    Output $m$
```

- Why does it work?
- If $m < n$, then first iteration will swap them.
- Each step after that will make n smaller than previous round.
- Actually it works ‘much faster’ than going down by 1 in each iteration.
- On termination, m is the GCD.

Algorithms

- Possible inputs need to be specified clearly
- Each step must be clearly defined and executable
- Many possible algorithms may exist
- Algorithm Efficiency
- Data Structures needed for doing the operations

Algorithms

- Understand the problem
- Design a method to solve
- Convert to an algorithm, step by step process to solve the problem
- Choose Data Structures to implement the algorithm
- Proving correctness of the algorithm/program
- Analyzing the algorithm
  - Time, Space, Simplicity, Generality
  - Program: Coding the algorithm

Some Important Problems

- Sorting: Stable sorting, In place sorting
- Searching
- String Processing
  - A string is sequence of symbols: $aba, a4bca, \ldots$
  - string matching, sub-string searching, etc
- Graph problems: graph traversal, shortest path, etc.
- Geometric problems: closest pair, convex hull, etc.
- Numerical problems: real numbers, approximations, etc from engineering/science

Design Strategies

- Brute Force
- Divide and Conquer (decrease and conquer, transform and conquer)
- Greedy technique
- Dynamic Programming
- Iterative improvement
- Backtracking
- Branch and bound
- P vs NP

Data Structures

Background Information: You should know about these.

- Way of organizing Data for easy manipulation
- Arrays, Linked Lists (singly linked list, doubly linked list), linear list, Stacks, Queues, Priority Queues, heaps
- Static memory allocation/dynamic memory allocation
- Directed/Undirected Graphs,
- Trees: rooted, ordered, binary trees
- Sets, Dictionaries

Data Structures

- Array is a sequence of n items (usually of the same type): refered to by $A(0)$ to $A(n-1)$ or $A(1)$ to $A(n)$ depending on context.
- Linked list is a sequence of zero of more items. They may have variable number of elements/members.
  - Constructed using pointers.
  - Starting element pointer.
  - Next element pointer from each element. Null pointer denotes end of list.
  - Doubly linked list allows traversal both ways; pointers to starting and ending element of the list.

Data Structures

- Linear list is an abstract concept, with insertion, deletion etc. It may be implemented using arrays or linked list.
- Stack is a list which is "last in first out". Operations push() and pop() to put elements in and take elements out of the list.
- Queue is a list which is "first in first out". Operations are enqueue() (to add to the end of the list) and dequeque() (to remove from front of the list).
- Priority Queue: Elements are put in the queue based on priority. Implemented using Heaps.

Graphs

- Graph is a set of points (vertices) and edges connecting vertices.
- $G = (V, E)$. V is the set of vertices. $E \subseteq V \times V$.
- Directed graph.
- Undirected graph $(u, v) \in E$ iff (v, u) in E.
- Self loops: edges of type (u, u).
- Edges may be presented using adjacency matrix $A(u, v) = 1$ iff (u, v) is an edge. Edges represented using adjacency lists: for each vertex u, use a linked list of edges $\{v : (u, v)$ is an edge $\}$.

- Complete graphs K_n. Complete Bipartite Graphs $K_{m,n}$.
- Weighted graphs: each edge has some weight, denoted by $A(u, v) = w$.
- Path: $v_1, v_2, v_3, \ldots, v_k$, where (v_i, v_{i+1}) is an edge in the graph for all i. Path from v_1 to v_k.
- Simple path (without vertices being repeated)
- Length of a path (each edge counts for 1 or weight of the edge).

- Cycle: paths where the first and last vertices are same. Simple cycle: only repetition in the path is the first and last vertex.
- Often, unless specified otherwise, we usually only talk about simple paths, cycles.
- Acyclic graph: no simple cycles
- Connected components: path from each vertex to every other vertex in the component (when viewed as "undirected graph").

Trees

- Tree is a connected acyclic graph
- Forest is a set of trees
- In a tree $|E| = |V| - 1$
- If in a connected graph, $|E| = |V| - 1$, then it is a tree.
- Rooted tree: a designated node. Leads to natural hierarchy based on distance from the root.
- Parent, Children, Ancestors, Descendants, Siblings, subtrees.
- Leaves of the tree: nodes without any children.

- Level/Depth of a node: distance from the root. Root is at Level 0.
- Depth/Height of a tree: maximum depth of any node.
- Height of a node: Leaves at Height 0.
- Ordered tree: Children are ordered (left, right; first, second, third, ...)

- Binary tree: at most two children of each node.
- Binary search tree: each node has a ‘key’
- Maximum Height of a binary tree with n nodes is $n-1$. Minimum height of a binary tree with n vertices is $\lfloor \log_2 n \rfloor$. To see this:

$$\sum_{i=0}^{h-1} 2^i + 1 \leq n \leq \sum_{i=0}^{h} 2^i$$

Which gives

$$2^h \leq n < 2^{h+1}$$

Sets, Dictionaries

- Set is a collection of unordered, distinct elements (members).
- Sets maybe represented using bit vectors (when the universal set is known) or lists, or other ways where one can do union, find (check if an element is member of a set), and other operations.
- Data structure representing a set which supports insertion, deletion, and searching is called a dictionary. Dictionary maybe represented using array, balanced search tree, hashing etc.

Abstract Data Types

- Various Data Types might be used to implementing an algorithm. If efficiency is not an issue, one ignores the exact way in which data types are implemented.
- Abstract Data type: where only operations on the data types are specified, and implementation is not specified.

Finding maximum value in an array

Input: Array A $(A[1], A[2], \ldots, A[n])$
Output: Largest element in the array.
Algorithm:

```
Findlargest(A)
    $m = A[1]$.
    For $i = 2$ to $n$ {
        If $A[i] > m$, then $m = A[i]$
            }
End
```

Number of operations:
$n - 1$ comparisons, n assignments (worst case)
$n - 1$ assignments to the loop variable, $\ldots$
Roughly takes time proportional to n or about $C * n$.

Analysis of Algorithms

- Induction
- Recurrence Relations
- Mathematical Tools
- Invariants, Loop Invariants

Some Basic notations and results

- $\log$, $\lg$ (base 2), $\ln$ (base e=2.718....)
- $\log_b a = n$ such that $b^n = a$.
- $\log_b a = \frac{\log a}{\log b}$
- $\log \frac{x}{y} = \log x - \log y$
- open interval (a, b), semi-open interval $(a, b]$ and closed interval $[a, b]$.
- Alphabet $\Sigma = \{a, b, \ldots\}$
- Strings (word) over the alphabet $ababba$; null string ϵ
- Sequences
- Boolean expressions: A and $(B$ or $\neg C)$

- binomial coefficients:
  $(x+y)^n =$
  $\binom{n}{0}x^0y^n + \binom{n}{1}x^1y^{n-1} + \ldots \binom{n}{i}x^iy^{n-i} + \ldots + \binom{n}{n}x^ny^0$.
  Also use the notation ${}^nC_i = \binom{n}{i}$
- $\frac{b^{n+1}-a^{n+1}}{b-a} = \Sigma_{i=0}^{n} a^i b^{n-i}$

Consider $(b-a) * \Sigma_{i=0}^{n} a^i b^{n-i}$.
$(\Sigma_{i=0}^{n} a^i b^{n+1-i}) - (\Sigma_{i=0}^{n} a^{i+1} b^{n-i})$.
$= b^{n+1} + (\Sigma_{i=1}^{n} a^i b^{n+1-i}) - (a^{n+1} + \Sigma_{i=0}^{n-1} a^{i+1} b^{n-i})$.
$= b^{n+1} - a^{n+1} + (\Sigma_{i=1}^{n} a^i b^{n+1-i}) - (\Sigma_{i=1}^{n} a^i b^{n+1-i})$.
$= b^{n+1} - a^{n+1}$

Thus, if $0 \leq a < b$, then

$\frac{b^{n+1}-a^{n+1}}{b-a} = \Sigma_{i=0}^{n} a^i b^{n-i} \leq (n+1)b^n$

- Limit, Infimum and Supremum of a series

  $\lim_{n\to\infty} f(n)$
  $\lim_{m\to\infty} \min\{f(n) : n \geq m\}$
  $\lim_{m\to\infty} \max\{f(n) : n \geq m\}$

- Lower Bounds and Upper Bounds
  Worst Case:
  Lower Bound $f(n)$: For each n, for some input of size n, algorithm takes time at least $f(n)$.
  Upper Bound $f(n)$: For each n, for all inputs of size n, algorithm takes time at most $f(n)$.

Asymptotic Upper Bounds

$f \in O(g)$, $f \in \Omega(g)$, $f \in \Theta(g)$.
f and g are functions from natural numbers to natural numbers.

$f \in O(g)$, iff there exist constants $c > 0$ and n_0 such that for all $n \geq n_0$, $f(n) \leq cg(n)$
Also common to use: $f = O(g)$ or f is $O(g)$ rather than $f \in O(g)$.

Some people also use the variation:
$f \in O(g)$ iff there exists constants $c > 0$ and $d > 0$, such that for all x
$f(x) \leq cg(x) + d$
For most cases, both are same/similar, though sometimes they differ when we want to use "small complexity"

– Example:
$f(n) = 50n^2 + 200n$.
$g(n) = n^3$.
Then $f(n) \leq 51g(n)$, for $n \geq 200$.
Thus, $f(n) \in O(g(n))$.

- Asymptotic Lower Bounds:

$f \in \Omega(g)$, iff there exist constants $c > 0$ and n_0 such that for all $n \geq n_0$, $f(n) \geq cg(n)$

- Asymptotic tight bounds

$f \in \Theta(g)$ iff $f \in O(g)$ and $f \in \Omega(g)$.
– Example
$f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$
Then, $f(n) \in \Theta(n^k)$.

Theorem: Suppose $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$.
(a) If $c = 0$, then $f(n) \in O(g(n))$, but $f(n) \notin \Omega(g(n))$.
(b) If $c = \infty$, then $f(n) \in \Omega(g(n))$, but $f(n) \notin O(g(n))$.
(c) If $0 < c < \infty$, then $f(n) \in \Theta(g(n))$.

Proof: By definition of limit, there exists an m such that for all $n > m$, $\frac{f(n)}{g(n)} \leq 1$.
Then, for $n \geq m$, $f(n) \leq g(n)$.
Rest: Exercise

$f = o(g)$ if for all $c > 0$, there exists a x_0 such that for all $x > x_0$, $f(x) \leq cg(x)$.
$f = \omega(g)$ if for all $c > 0$, there exists a x_0 such that for all $x > x_0$, $f(x) \geq cg(x)$.

- L' Hospital's Rule
  If $\lim_{n\to\infty} f(n)$ and $\lim_{n\to\infty} g(n)$ are both 0 or both ∞, then
  $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$.
- Showing $f \in O(g)$
  - Explicit bounds as in definition
  - Using limit theorem

  You can use either, unless explicitly asked to use one of the methods ....
- Misuse notation and say $f(n) = O(g(n))$ etc.
- Recurrence relations

Example: Finding maximum value in an array

Input: Array A ($A[1], A[2], \ldots, A[n]$)
Output: Largest element in the array.
Algorithm:

```
Findlargest(A[1:n])
    If $n = 1$, then return $A[1]$.
    Else,
        let $m = Findlargest(A[1 : n-1])$.
        If $m > A[n]$, then return $m$
        Else return $A[n]$.
End
```

$T(n) \leq T(n-1) + C_1$.
$T(1) = C_2$

Finding element in sorted array

```
FindElement$(A, i, j, key)$
    If $i > j$, return 0
    Let $k = \frac{i+j}{2}$.
    If $A[k] = key$, then return 1
    Else, if $A[k] < key$, then return FindElement(A,k+1,j,key)
    Else return FindElement(A,i,k-1,key)
End
```

$T(n) \leq T(n/2) + C$.

Some further recurrence examples:

$T(n) = T(n-1) + n$ for $n \geq 2$ and $T(1) = 1$

$T(n) = 2T(\lceil n/2 \rceil) + n$, for $n \geq 2$, and $T(1) = 1$.

$T(n) = 2T(n-1) + 1$, for $n \geq 2$, and $T(1) = 1$.

$T(n) = 9 + \Sigma_{i=1}^{n-1} T(i)$, for $n \geq 2$, and $T(1) = 7$.

Solve $T(n) = T(n-1) + 1$, for $n \geq 2$
$T(1) = 1$.

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= T(n-2) + 1 + 1 \\ &= T(n-2) + 2 \\ &= T(n-3) + 3 \\ &= \cdots \\ &= T(1) + n - 1 \\ &= n \end{aligned}$$

$T(n) = T(n/2) + n$, for $n \geq 2$
$T(1) = 1$
Solve $T(n)$ for n being power of 2.

$$\begin{aligned}
T(n) &= T(n/2) + n \\
&= T(n/4) + n/2 + n \\
&= T(n/8) + n/4 + n/2 + n \\
&= \cdots \\
&= T(1) + 2 + 4 + \cdots + n/4 + n/2 + n \\
&= 1 + 2 + 4 + \cdots + n/4 + n/2 + n \\
&= 2n - 1
\end{aligned}$$

Need not want exactly, but only in order.

$T(n) = n + T(n-1)$, $T(1) = 1$

$T(n) = n + (n-1) + (n-2) + \ldots + 1$

$T(n) \leq n * n = O(n^2)$.

Solving Recurrence Relations by Induction

- Guess the answer
- Prove by induction

Example:

$T(n) = T(n/3) + T(2n/3) + n$

$T(n) \leq C_2 n \log n$

Main Recurrence Theorem (Master Theorem)

– Useful for solving many cases (does not always work).

Theorem: Suppose $a \geq 1$, $b \geq 2$ are constants.

Upper Bound:

If $T(n) \leq aT(n/b) + f(n)$, where $f(n) = O(n^k)$, then

$$T(n) = \begin{cases} O(n^k), & \text{if } a < b^k; \\ O(n^k \log n), & \text{if } a = b^k; \\ O(n^{\log_b a}), & \text{if } a > b^k; \end{cases}$$

Lower Bound:

If $T(n) \geq aT(n/b) + f(n)$, where $f(n) = \Omega(n^k)$, then

$$T(n) = \begin{cases} \Omega(n^k), & \text{if } a < b^k; \\ \Omega(n^k \log n), & \text{if } a = b^k; \\ \Omega(n^{\log_b a}), & \text{if } a > b^k; \end{cases}$$

Exact Bound:

If $T(n) = aT(n/b) + f(n)$, where $f(n) = \Theta(n^k)$, then

$$T(n) = \begin{cases} \Theta(n^k), & \text{if } a < b^k; \\ \Theta(n^k \log n), & \text{if } a = b^k; \\ \Theta(n^{\log_b a}), & \text{if } a > b^k; \end{cases}$$

The theorem also holds if $b > 1$.

Proof Idea: Consider the case of $n = b^r$ for some natural number r. Let $b^k = d$ in the following.

$$\begin{array}{rcl}
T(b^r) & \leq & aT(b^{r-1}) + c[(b^r)^k] \\
T(b^r) & \leq & aT(b^{r-1}) + c[d^r] \\
 & \leq & a[aT(b^{r-2}) + c[d^{r-1}]] + c[d^r] \\
 & \leq & a^2T(b^{r-2}) + c[d^r + a^1 d^{r-1}] \\
 & & \ldots \\
 & \leq & a^r T(b^{r-r}) + c[\sum_{i=1}^{r} [a^{r-i} d^i]] \\
 & \leq & c'[\sum_{i=0}^{r} [a^{r-i} d^i]]
\end{array}$$

If $d > a$, then

$$\begin{aligned} T(b^r) &\leq c' \frac{d^{r+1} - a^{r+1}}{d - a} \\ &= O(d^r) = O(n^k) \end{aligned}$$

If $d < a$, then

$$\begin{aligned} T(b^r) &\leq c' \frac{a^{r+1} - d^{r+1}}{a - d} \\ &= O(a^r) = O(b^{r \log_b a}) = O(n^{\log_b a}) \end{aligned}$$

If $d = a$, then

$$\begin{aligned} T(b^r) &\leq c'[(r+1) * d^r] \\ &= O(n^k \log n) \end{aligned}$$

Consider $H(n) = aH(n/b) + cn^k$; $H(1) = T(1)$.

Note that $T(n) \leq H(n)$.
$H(n)$ is increasing function.
The bound we obtained above would give,
$H(n) \leq$ ...., for $n = b^r$.
If $b^{r-1} < n' < b^r$, then
$T(n') \leq H(n') \leq H(b^r) \leq$ .....

For example, if we have $H(n) \leq c' * n^k$, for n of the form b^r,
then for $b^{r-1} < n' < b^r = n$, we have
$T(n') \leq H(n') \leq H(n) \leq c' * n^k \leq c' * b^k (n')^k \leq c''(n')^k$.

Example:
$T(n) = T(n/2) + n$.
$a = 1$, $b = 2$, $k = 1$
So $a < b^k$
$T(n) = O(n)$

Example: $T(n) = 2T(n/2) + n$
$a = 2, b = 2, k = 1$
So $a = b^k$
$T(n) = O(n \log n)$