NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Algorithm Analysis { Counting - Iterative Q1-Q3 / - Recursive Q4-Q5 ; Computing upper & lower bound $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ { 0 $f=\Omega(g)$; c $f=\Theta(g)$; ∞ $f=O(g)$

CX2001 Algorithms **2020/21 Semester 1**

Tutorial 1: Algorithm Analysis

School of Computer Science and Engineering *Nanyang Technological University*

Week 3 (Q1-Q3):

Q1 The median of an ordered set is an element such that the number of elements less than the median is within one of the number that are greater, assuming no ties. The algorithm is as follows.

```
if(a < b)
   if(b < c) median = b;
   else if (a < c) median = c;
   else median = a;
else if (a < c) median = a;
else if (b < c) median = c;
else median = b;
```

(a) What are the different types of input cases for the algorithm, in light of the number of comparisons that are done by the algorihtm?

(b) How many comparisons does the algorithm do in the worst case? in the average case?

Q2 This is an algorithm to compute the transpose of a matrix:

```
for(i = 0; i < n; i++)
  for(j = 0; j < n; j++) read in m[i][j];
for(i = 1; i < n; i++)
  for(j = 0; j < i; j++) swap m[i][j] with m[j][i];
for(i = 0; i < n; i++)
  for(j = 0; j < n; j++) output m[i][j];
```

(a) How many swap operations are performed?

(b) What is the time complexity of this algorithm?

Q3 The function subset() below takes two linked lists of integers and determines whether the first is a subset of the second. Give the worst-case running time of subset as a function of the lengths of the two lists. When will this worst case happen?

```
public class ListNode {
  int item;
  ListNode next; ... }

//Check whether integer X is an element of linked list Q
int element (int X, ListNode Q)
{
  boolean found;  //Flag whether X has been found
  found = false;
  while ( Q != NULL && !found) {
     found = Q.item == X;
     Q = Q.next;
  }
  return found;
}

// Check whether L is a subset of M
int subset (ListNode L, ListNode M)
```

```
{
  boolean success; // Flag whether L is a subset so far
  success = true;
  while ( L != NULL && success) {
      success = element(L.item, M);
      L = L.next;
  }
  return success;
}
```

Week 4 (Q4-Q6):

Q4 For each of the following algorithm, find a recurrence equation for the number of multiplications as a function of N, and find the asymptotic growth rate of the solution to equation. To simplify the analysis, you may assume that N is a power of two; that is $N = 2^k$ for some integer K. These functions compute X to the power N, where N is a positive integer.

```
int power1 (int X, int N)
{
    if ( N == 1) return X;
    else return X * power1(X, N-1);
}

int power2 (int X, int N)
{
    int HALF, HALFPOWER;

    if ( N == 1) return X;
    else{
        HALF = N/2;
        HALFPOWER = power2(X, HALF);
        if (2*HALF == N) // if N is even
            return HALFPOWER * HALFPOWER;
        else return HALFPOWER * HALFPOWER * X;
    }
}

int power3 (int X, int N)
{
    int HALF;

    if ( N == 1) return X;
    else {
        HALF = N/2;
        if (2*HALF == N) // if N is even
            return power3(X, HALF) * power3(X, HALF);
        else return power3(X, HALF) * power3(X, HALF) * X;
    }
}
```

Q5 For the following algorithm, find a recurrence equation for the number of function calls as a function of N, and find the asymptotic growth rate of the solution to equation.

```
int magic(int N)
{
    if ( N == 1 || N== 2) return 1;
    else return magic(N-1) + magic(N-1) + magic(N-2);
}

```

Q6 For each of the following pairs of functions $f(n)$ and $g(n)$, state whether f is $\mathcal{O}(g)$; whether f is $\Theta(g)$; and whether f is $\Omega(g)$. (More than one of these can be true for a single pair.) Justify your answers.

a. $f(n) = n^{10}$ and $g(n) = 2^{\frac{n}{2}}$

b. $f(n) = n^{\frac{3}{2}}$ and $g(n) = n(\log_2 n)^2$

c. $f(n) = \log_2(n^3)$ and $g(n) = \log_2(n)$

d. $f(n) = \log_2(3^n)$ and $g(n) = \log_2(2^n)$

e. $f(n) = 2^n$ and $g(n) = 2^{\frac{n}{2}}$

Q1 The median of an ordered set is an element such that the number of elements less than the median is within one of the number that are greater, assuming no ties. The algorithm is as follows.

```
if(a < b)
   if(b < c) median = b;
   else if (a < c) median = c;
   else median = a;
else if (a < c) median = a;
else if (b < c) median = c;
else median = b;
```

(a) What are the different types of input cases for the algorithm, in light of the number of comparisons that are done by the algorihtm?

(b) How many comparisons does the algorithm do in the worst case? in the average case?

1a)

The number of possible permutations are: # of permutations: $= {}^3C_3 \times {}^2C_2 \times {}^1C_1$

$= 6$ #

a, b, c
c, b, a
b, a, c
c, a, b
a, c, b
b, c, a

– total 6 possible permutations

case	conditional statements	median	# of comparisons
a>b>c	a<b F a<c F b<c F	b	3
a>c>b	a<b F a<c F b<c T	c	3
b>c>a	a<b T b<c F a<c T	c	3
b>a>c	a<b T b<c F a<c F	a	3
c>a>b	a<b F a<c T	a	2
c>b>a	a<b T b<c T	b	2

1b)

E.g
a b c
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

6 cases Brute Force

Results	# of comparisons
2	2
2	3
2	3
2	3
2	2
2	3

Ave comparisons = $\frac{\text{total comparisons}}{\text{\# of cases}} = \frac{16}{6} = \frac{8}{3}$.

# of comparisons in worst case = 3

# of comparisons in best case = 2

# of comparisons in average case = $\frac{1}{6}(2+3+3+3+2+3) = \frac{8}{3}$ comparisons.

Q2 This is an algorithm to compute the transpose of a matrix:

```
for(i = 0; i < n; i++)
  for(j = 0; j < n; j++) read in m[i][j];
for(i = 1; i < n; i++)
  for(j = 0; j < i; j++) swap m[i][j] with m[j][i];
for(i = 0; i < n; i++)
  for(j = 0; j < n; j++) output m[i][j];
```

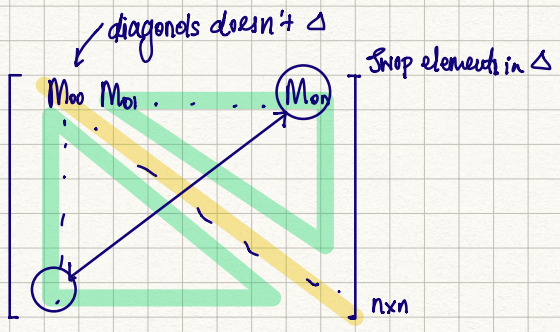
2a)

(a) How many swap operations are performed?

(b) What is the time complexity of this algorithm?

Q2a) matrix transpose

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$



No. of swaps = No. in lower/upper triangles

$i = 1, \ j = 0$ — 1 swap
$i = 2, \ 0 \le j \le 1$ — 2 swaps
$i = 3, \ 0 \le j \le 2$ — 3 swaps.
⋮
$i = n-1 \ \ 0 \le j \le n-2$ — $n-1$ swaps

formula for sum of arithmetic sequence

$S = a + (a+d) + (a+2d) + \ldots + (a+(n-1)d)$
$= (a+(n-1)d) + (a+(n-2)d) + \ldots + (a+d) + a$ ← Reverse the sequence then add together

$2S = (2a+(n-1)d) + (2a+(n-1)d) + \ldots + (2a+(n-1)d)$
$= (2a+(n-1)d)n$
$S = \frac{(2a+(n-1)d)n}{2}$

∴ total swaps $= 1+2+3+\ldots+(n-1)$ swaps
$= \frac{(n-1)n}{2}$

Q2b) Reading: n^2 iterations
transpose: $\frac{n(n-1)}{2}$ swap
writing: n^2 iterations.

Each reading operations takes a unit time
" transpose " " b "
" writing. " " c "

total time $= an^2 + \frac{bn(n-1)}{2} + cn^2$
$= (a + \frac{b}{2} + c)n^2 - \frac{bn}{2}$
According to big O notation dominator term
$= O(n^2)$

Q3 The function subset() below takes two linked lists of integers and determines whether the first is a subset of the second. Give the worst-case running time of subset as a function of the lengths of the two lists. When will this worst case happen?

```
1  public class ListNode {
2    int item;
3    ListNode next; ... }
4
5  //Check whether integer X is an element of linked list Q
6  int element (int X, ListNode Q)
7  {
8    boolean found;  //Flag whether X has been found
9    found = false;
10   while ( Q != NULL && !found) {
11      found = Q.item == X;
12      Q = Q.next;
13   }
14   return found;
15 }
16
17 // Check whether L is a subset of M
18 int subset (ListNode L, ListNode M)
{
  boolean success; // Flag whether L is a subset so far
  success = true;
  while ( L != NULL && success) {
      success = element(L.item, M);
      L = L.next;
  }
  return success;
}
```

Searching each elements 1 by 1
Naive search

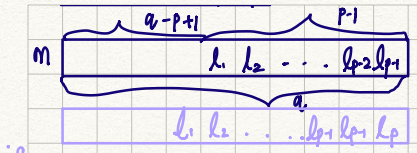
Worst case:

① the first $|L|-1$ elements of L from the last $|L|-1$ elements of M. (Each call of element time takes as much time as possible)
② the last element of L not in M (Repeat loop as many times as possible)

Case 2

L | L_1 | L_2 | L_3 | . . . | L_{p-1} | L_p |

to get worse case L is distinct, don't expect duplication.



$p \le q$

if element(l_p, M) = T

CASE	# of comparisons
element(l_1, M) = T, $l_1 \in M$	(q-p+1)+1
element(l_2, M) = T, $l_2 \in M$	(q-p+1)+2
⋮	
element(l_{p-1}, M) = T, $l_{p-1} \in M$	(q-p+1)+(p-1)
element(l_{p-2}, M) = F	(q-p+1)+(p-1) = q

↑ doesn't matter T or F but if it is T you will save time

total comparisons = $(q-p+1) \times (p-1) + \frac{(p-1)p}{2}$

Q4 For each of the following algorithm, find a recurrence equation for the number of multiplications as a function of N, and find the asymptotic growth rate of the solution to equation. To simplify the analysis, you may assume that N is a power of two; that is, $N = 2^K$ for some integer K. These functions compute X to the power N, where N is a positive integer.

How many multiplications

T_{N-1}

$N = 2^K$

$N = 2^1 = 2$
$N = 2^2 = 4$
$N = 2^3 = 8$
⋮
$N = 2^K$

```
int power1 (int X, int N)
{
    if ( N == 1) return X;
    else return X * power1(X, N-1);
}

int power2 (int X, int N)
{
    int HALF, HALFPOWER;

    if ( N == 1) return X;
    else{
        HALF = N/2;
        HALFPOWER = power2(X, HALF);
        if (2*HALF == N) // if N is even
            return HALFPOWER * HALFPOWER;
        else return HALFPOWER * HALFPOWER * X;
    }
}

int power3 (int X, int N)
{
    int HALF;

    if ( N == 1) return X;
    else {
        HALF = N/2;
        if (2*HALF == N) // if N is even
            return power3(X, HALF) * power3(X, HALF);
        else return power3(X, HALF) * power3(X, HALF) * X;
    }
}
```

each time reduce exponent by 1 ⇒ V. inefficient.

Apply 3-step method for counting in recursive algo

Step 1: check the base case

Step 2: Figure out recurrence eqn.
* must relate current term to previous terms.
T_n = some previous terms on the RHS.

Step 3: Solve recurrence eqn.

For Algo 1: Power1( )

Let no. of multiplications be f(n)

f(1) = 0

f(2) = 1 + f(1) = 1(1) + f(1) = 1(1)

f(3) = 1 + f(2)
= 1 + 1 + f(1) = 2(1) + f(1)
= 2(1)

f(4) = 1 + f(3)
= 1 + 1 + f(2)
= 1 + 1 + 1 + f(1)
= 3(1) + f(1)
= 3(1)

T_n: # of multiplications in power(n, x)

T_{n-1}: " " in power(n-1, x)

Step ①: Check the base case
n = 1, $T_1 = 0$

Step ②: T_n = some previous terms on RHS

$T_n = \boxed{1} T_{n-1} + \square$

↑ # of calls of power(n-1)

if X^{n-1} is known, how many more multiplications we need to compute X^n

Step ③: Solve Recurrence eqn

$T_1 = 0$
$T_n = T_{n-1} + 1$

$\therefore f(N) = (N-1) \cdot 1$
$= N-$

If there's one term on RHS, we just keep replacing n by n-1, until reaching base 1 case.

n-1 eqn.

$$\begin{cases} T_n = T_{n-1} + 1 \\ T_{n-1} = T_{n-2} + 1 \\ T_{n-2} = T_{n-3} + 1 \\ \vdots \\ T_2 = T_1 + 1 \end{cases}$$

Adding all the eqns together, we obtain

$T_n = T_1 + \underbrace{1 + \ldots + 1}_{n-1 \text{ ones}}$

$= n-1$

```
int power2 (int X, int N)
{
    int HALF, HALFPOWER;

    if ( N == 1) return X;
    else{
        HALF = N/2;
        HALFPOWER = power2(X, HALF);
        if (2*HALF == N) // if N is even
            return HALFPOWER * HALFPOWER;
        else return HALFPOWER * HALFPOWER * X;
    }
}
```

$T_n = \boxed{1} T_{\frac{n}{2}} + \square$

↳ Recursive call 1x

reach time we half the exponent

2 cases.

Step 1: $T_1 = 0$

Step 2: $T_n = \boxed{1} T_{n/2} + \square$

if $x^{\frac{n}{2}}$ is known, how many more multiplications to compute x^n

$$T_n = \begin{cases} T_{\frac{n}{2}} + 2 & \text{if } n \text{ is even} \\ T_{\frac{n}{2}} + 3 & \text{otherwise} \end{cases}$$

Given: $N = 2^K$

N is always even

$K = \log_2 n$

$$\begin{cases} T_{2^0} = 0 \\ T_{2^K} = T_{2^{K-1}} + 2 \end{cases}$$

Keep add eq

$$\begin{cases} T_{2^K} = T_{2^{K-1}} + 2 \\ T_{2^{K-1}} = T_{2^{K-2}} + 2 \\ T_{2^{K-2}} = T_{2^{K-3}} + 2 \\ \vdots \\ T_{2^{K-(K-1)}} = T_{2^{K-K}} + 2 \end{cases}$$

$T_{2^K} = T_{2^0} + \underbrace{2 \ldots + 2}_{K \text{ twos}}$

$= 2K$

$= 2 \log_2 n$

Same time complexity as

```c
int power3 (int X, int N)
{
    int HALF;

    if ( N == 1) return X;
    else {
        HALF = N/2;
        if (2*HALF == N) // if N is even
            return power3(X, HALF) * power3(X, HALF);
        else return power3(X, HALF) * power3(X, HALF) * X;
    }
}
```

part 1 ⇒ inefficient !

① $T_1 = 0$

② $T_n = 2T_{n/2} + \square$ — called Recursive Function

$T_n = \begin{cases} 2T_{n/2} + 2 & \text{if } n \text{ is even} \\ 2T_{n/2} + 3 & \text{otherwise} \end{cases}$

③
$$\begin{cases} T_{2^0} = 0 \\ T_{2^K} = 2T_{2^{K-1}} + 2 \quad ① \end{cases}$$

Sum
$$\begin{cases} 2T_{2^K} = 2T_{2^{K-1}} + 2 & ② \times 2 \\ 2^2 T_{2^{K-1}} = 2T_{2^{K-2}} + 2 & ③ \times 2^2 \\ \\ 2^K T_{2^{K-(K-1)}} = 2^K T_{2^{K-K}} + 2^K & \times 2^K \end{cases}$$

Geometric seq

$T_{2^K} = T_{2^0} + 2^1 + 2^2 + \dots + 2^K$

$= \frac{2^{K+1} - 2}{2 - 1}$

$= 2(2^K - 1) = 2(n-1)$

*IMP: Know what the Q^n asking!

Q5 For the following algorithm, find a recurrence equation for the number of function calls as a function of N, and find the asymptotic growth rate of the solution to equation.

```
int magic(int N)
{
    if ( N == 1 || N== 2) return 1;
    else return magic(N-1) + magic(N-1) + magic(N-2);
}
```

2 Base cases.

Step 1: magic(1) magic(2)
$T_1 = 1$, $T_2 = 1$

Step 2: $T_n = \boxed{2}T_{n-1} + \boxed{1}T_{n-2} + \boxed{1}$ — (*) (magic(N))

$T_n = 2T_{n-1} + T_{n-2} + 1$ — (**) ← Second order non-homogeneous eqn

↳ split into 3 parts, make 1 disappears, troublemaker

Assume b_n to be homogeneous.

$T_n = 2T_{n-1} + T_{n-2} + 1$ — ①

$T_n = b_n - K$ — ②

Sub ② into ①

$b_n + K = 2b_{n-1} + 2K + b_{n-2} + K + 1$

$b_n = 2b_{n-1} + b_{n-2} + (2K+1)$

since b_n is homog.: this term should not exist

$\therefore 2K + 1 = 0$

$K = -\frac{1}{2}$

$\therefore T_n = b_n - \frac{1}{2}$

(*) = (**)

$(T_n + \frac{1}{2}) = 2(T_{n-1} + \frac{1}{2}) + (T_{n-2} + \frac{1}{2})$

$b_n = T_n + \frac{1}{2}$ ∴ $b_n = 2b_{n-1} + b_{n-2}$, $b_1 = T_1 + \frac{1}{2} - \frac{3}{2} = b_2$

Characteristic func:

$x^2 = 2x + 1$

$x^2 - 2x - 1 = 0$

$x_{1,2} = \frac{2 \pm \sqrt{2^2 + 4}}{2} = 1 \pm \sqrt{2}$

$b_n = \alpha(1+\sqrt{2})^n + \beta(1-\sqrt{2})^n$

Method 2:

The solution consists of a homogeneous solution and a particular solution

$W(N) = W^h(N) + W(N)$

The homogeneous solution can be obtained as the following

$W^h(N) - 2W^h(N) - W^h(N) = 0$

$r^2 - 2r - 1 = 0$

$r = 1 \pm \sqrt{2}$

$W^h(N) = \alpha(1+\sqrt{2})^n + \beta(1-\sqrt{2})^n$

The particular solution $W(N)$ is a constant, A. A can be obtained by substituting into the recurrence eqn

$A = 2A + A + 1$

$A = -\frac{1}{2}$

$W(N) = -\frac{1}{2}$

From the algo $W(1) = W(2) = 1$. Sub into α & β

$\alpha = \frac{3}{4}(\sqrt{2}-1)$ $\beta = -\frac{3}{4}(\sqrt{2}+1)$ ⇒ $\therefore W(n) = \frac{3}{4}[(1+\sqrt{2})^{n-1} + (1-\sqrt{2})^{n-1}] + \frac{1}{2}$

$\therefore W_n \in (1)(1+\sqrt{2})^n$

Q6 For each of the following pairs of functions $f(n)$ and $g(n)$, state whether f is $\mathcal{O}(g)$; whether f is $\Theta(g)$; and whether f is $\Omega(g)$. (More than one of these can be true for a single pair.) Justify your answers.

a. $f(n) = n^{10}$ and $g(n) = 2^{\frac{n}{2}}$

b. $f(n) = n^{\frac{3}{2}}$ and $g(n) = n(\log_2 n)^2$

c. $f(n) = \log_2(n^3)$ and $g(n) = \log_2(n)$

d. $f(n) = \log_2(3^n)$ and $g(n) = \log_2(2^n)$

e. $f(n) = 2^n$ and $g(n) = 2^{\frac{n}{2}}$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0, & f(n) = o(g(n)) \\ c, & f(n) = \Theta(g(n)) \\ & g(n) = \Theta(f(n)) \\ \infty, & f(n) = \Omega(g(n)) \end{cases}$

Q6(a) $\lim_{n\to\infty} \frac{n^{10}}{2^{n/2}} = \lim_{n\to\infty} \frac{10n^9}{\ln 2 (2^{n/2}) \cdot \frac{1}{2}}$

$= \lim_{n\to\infty} \frac{10}{\ln 2}\left(\frac{n^9}{2^{n/2}}\right)$

$= \lim_{n\to\infty} \frac{10}{\frac{1}{2}\ln 2} \frac{9 \times n^8}{(\ln 2)(\frac{1}{2})(2^{n/2})}$

$= \vdots$

$= \lim_{n\to\infty} \frac{10!}{\times n^0}$

Q6(b) $\lim_{n\to\infty} \frac{n^{3/2}}{n(\log_2 n)^2}$

$= \lim_{n\to\infty} \frac{n^{\frac{1}{2}}}{(\log_2 n)^2}$ → L'Hopital Rule

$= \lim_{n\to\infty} \frac{\frac{1}{2}n^{-\frac{1}{2}}}{2\log_2 n \left(\frac{1}{(\ln 2)(n)}\right)}$

(a) L'Hopital Rule

A method use to find the limit of a func of a $\frac{0}{0}$ or $\frac{\infty}{\infty}$ condition.

Formulas of derivative.

$(n^k)' = kn^{k-1}$

Chain Rule
$((f(n))^k)' = k(f(n))^{k-1} f'(n)$

$(a^n)' = a^n \ln a$

$a^{f(n)} \ln a \cdot f'(n)$

$(\log_b n)' = \frac{1}{(\ln b)\, n}$

$(\log_b f(n))' = \frac{1}{\ln b \cdot f(n)} f'(n)$

* Always simplify the expression before applying L'Hopital Rule.

$\lim_{n\to\infty} \left(\frac{1}{2}\cdot \ln 2\right)^{10} \cdot 2^{\frac{n}{2}}$

$= \lim_{n\to\infty} \frac{10!}{\left(\frac{1}{2}\ln 2\right)^{10}} \times \frac{1}{2^{\frac{n}{2}}} = 0.$

$= \lim_{n\to\infty} \frac{\ln 2}{4} \cdot \frac{n^{\frac{1}{2}}}{\log_2 n}$

$= \lim_{n\to\infty} \frac{\ln 2}{4} \cdot \frac{\frac{1}{2} n^{-\frac{1}{2}}}{\frac{1}{\ln 2}\cdot\frac{1}{n}}$

$= \lim_{n\to\infty} \frac{(\ln 2)^2}{8} \cdot n^{\frac{1}{2}}$

$= \infty$

(b c) $\lim_{n\to\infty} \frac{\log_2(n^3)}{\log_2(n)}$

$= \lim_{n\to\infty} \frac{3\log_2(n)}{\log_2 n}$

$= \lim_{n\to\infty} 3 = 3$ //

(b d) $\lim_{n\to\infty} \frac{\log_2(3^n)}{\log_2(2^n)}$

$= \lim_{n\to\infty} \frac{n\log_2(3)}{n\log_2(2)}$ $= \lim_{n\to\infty} \log_2(3)$

$\log_2(2) = 1$

$= \log_2(3)$

(b e) $\lim_{n\to\infty} \frac{2^n}{2^{n/2}}$

$= \lim_{n\to\infty} 2^{n/2}$

$= \infty$ //