CS2705: Programming & Data Structures

```
define gcd(x, y):
    if x < y:
        swap(x, y)
    if y == 0:
        return x
    return gcd(x % y, y)
```

→

```
define gcd(x, y):
    if y == 0
        return x
    return gcd(y, x % y)
```

Let $x = g \cdot a$, $y = g \cdot b$ with a, b co-prime, $a \geq b$

$\gcd(g \cdot b, g(a-b)) = \gcd(g \cdot a, g \cdot b)$

Because $a - b$ is coprime to b otherwise a would not be coprime to b.

Correctness - by proofs

Efficiency - time or space complexity

Euclid's GCD: Complexity varies with ? $2\lceil \log_2 x \rceil$ is size

For arrays, we assume the size of each element is a constant, eg 32 or 64 bits.

Exponential - c^n Polynomial - n^c Logarithmic - $\log_2 n$

If $x \geq y$, $x \% y \leq x/2$ => larger number loses MSB

$x = x + y$

~~$x = x + 2y$~~

$y = x - y$

$x = x - y$

$\approx$ constant

```
for i = 1 to n:
    A[i] = 0
```

$\approx n$

```
for i = 1 to n:
    for j = 1 to n:
        A[i][j] = 0
```

$\approx n^2$

FABER-CASTELL
Date
Page No.

Big O

$f(n) = O(1)$ iff $\exists$ +ve constants c and n_0 :
$f(n) \leq c \quad \forall \quad n \geq n_0$

$f(n) = O(n)$ iff $\exists$ +ve constants c and n_0 :
$f(n) \leq cn \quad \forall \quad n \geq n_0$

$f(n) = O(g(n))$ iff $\exists$ +ve constants c and n_0 :
$f(n) \leq c \cdot g(n) \quad \forall \quad n \geq n_0$

Terms:

1. Algorithmic problem - A problem that can be solved by following a logical list of steps.
2. Algorithm - A logical list of steps to be executed
3. Data structure - A way of storing, organizing and obtaining data.
4. Size of input - The space required to store the input.
5. Analysis of Algo - The study of the correctness and space-time complexity of an algorithm.

~~$f(2n^2+$~~ $f(n) = 2n^2 + 3n + 8$

$f'(n) = 3n^2 + 2n + 5$

$f''(n) = n^3 + 6n + 8$

$f \approx f'$ but f best

$f(n) = O(n^2)$ but also $f(n) = O(n^3)$ etc.

$f(n) = \Omega(g(n))$ iff $\exists$ +ve constants c and n_0 :
$f(n) \geq c \cdot g(n) \quad \forall \quad n \geq n_0$

$f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ ~~Θ~~
and $f(n) = \Omega(g(n))$

FABER CASTELL

Date

Page No.

$f(x)$: $1, x, x^2, x\log x, 2^x$

$f(n) = n^2 + 2n + 1 = O(n^k) = \Omega(n^k)$

Maximum Subsequence Sum Problem

n integers in an array A

Sum of subsequence $(i,j) = \sum_{k=i}^{j} A[k]$

Output maximum such sum

Inputs all ≥ 0 => Output = $\sum A[k]$ $\forall k$

Inputs all ≤ 0 => Output = 0

Claims: 1 Optimal sol^n is unique

2 Optimal sol^n contains max value

3 Optimal sol^n does not contain neg. values

All False!

```
maxsum = 0
for i = 0 to n-1                          -> O(n)
    for j = i to n-1                      -> O(n)
        currsum = 0
        for k = i to j                    -> O(n)
            currsum += A[k]               -> O(1)
        if currsum > maxsum
            maxsum = currsum
return maxsum                             ∴ O(n^3)
```

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j} 1 = \sum_{i=0}^{n-1}\sum_{j=i}^{n-1} j-i+1 = \sum_{i=0}^{n-1}\sum_{k=1}^{n-i} k = \sum_{i=0}^{n-1}\frac{(n-i)(n-i+1)}{2}$$

$$= \sum_{i=0}^{n-1}\frac{n^2-2ni+i^2+n-i}{2} = \frac{n^2+n}{2}n + \frac{(n-1)(n)(2n-1)}{12} - \frac{2n+1}{2}\frac{(n-1)n}{2}$$

$$= \frac{n-1}{12}\left(6n^2+6n+2n^2-n-6n^2-3n\right) = \frac{(n-1)(n^2+n)}{6} = n^3$$

$$= \frac{n}{12}\left(6n^2+6n+2n^2-3n+1-6n^2+3n+3\right) = \frac{n^3+3n+2}{6} = \Theta(n^3)$$

	0	1	2	3	4	5	6	7
A:	7	-2	8	10	-20	-12	13	16
B:	7	5	13	23	3	-9	13	29

answer is max element of B or 0

$B[i] = \max(B[i-1], 0) + A[i]$

```
max_sum = 0                                  } → Θ(1) space
curr_sum = 0
for i = 0 to n-1                             → Θ(n) time
    curr_sum = max(curr_sum, 0) + A[i]
    max_sum = max(max_sum, curr_sum)
return max_sum
```

Online Algorithm: Array A could be a stream of inputs instead.

Missing Value Problem

Array A of n integers, find the smallest positive missing value.

A = 2, -1, 14, -7, 3, 4, -6, 15, -8 : 1

B = 1, 2, 3, 4, 5, 6 : 7

~~for i = 0 to n-1~~

~~if A[i] >= 1 and A[i] <= n and A[i] != A[A[i]-1]~~

~~swap(A[i], A[A[i]-1])~~

```
i = 0
while i < n
    if A[i] > 0 and A[i] <= n and A[i] != A[A[i]-1]
        swap(A[i], A[A[i]-1])
    else
        i++
```

(continued on next page)

FABER CASTELL
Date
Page No.

```
for i=0 to n-1
    if A[i] != i+1
        return i+1
return n+1
```

Binary Search:
Iterative or Recursive
In a sorted array, compare the 'middle' element with the element to be found, choose one half to find the element in.

Time complexity → $T(n) \leq 1 + T(n/2)$, $T(1) = 1$

Repeated substitution

$T(n) \leq 1 + 1 + T(n/4) \leq 1+1+1 + T(n/8) \leq \log_2 n + T(1)$

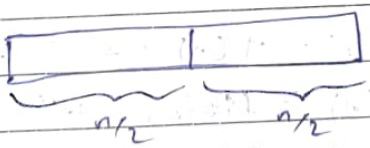
$k + T(n/2^k)$

$\therefore T(n) \leq \log n \Rightarrow T(n) = O(\log n)$

Find Maximum in A:

$T(n) \leq 1 + T(n-1)$

$T(1) = 1$



$T(n) \leq 2\,T(n/2)$

$T(n) \leq 1 + T(n-1) \leq 1+1 + T(n-2) \leq n-1 + T(1) \leq n$

$k + T(n-k)$

$T(n) \leq 2^{+1} T(n/2) \leq 2^{1+2} T(n/2^2) \leq 2^{\log n} T(n/n) \leq 2n-1$

$2^k\, T(n/2^k) \cdot 2^k - 1$

$\therefore T(n) \leq = \Theta(n)$

```
def max_in_A(A):
    if [A.length() == 1]:
        return A[0]
    return max(max_in_A(A[0... n/2 - 1]),
```

~~Deep~~ Shallow, O(log n) stack space

```
def maximum(A):
    n = A.length()
    if (n == 1):
        return A[0]
    return max(maximum[A[0... n/2 - 1]),
               maximum(A[n/2 ... n-1]))
```

~~Shallow~~ Deep, O(n) stack space

```
def maximum(A):
    if [A.length() == 1]:
        return A[0]
    return max(A[0], maximum(A[1... n-1])
```

Pseudopolynomial Algorithm: Polynomial in the value of the input as opposed to bits in the binary representation.

Fibonacci: $T(n) = T(n-1) + T(n-2) + 1$

$T(n) \geq fib(n) \geq 1.5^n \quad \forall\ n > 4$

$\therefore T(n) = \Omega(2^n)$

FABER CASTELL
Date
Page No.

ABSTRACT DATA TYPE

Interface is specified, time complexity is usually specified

List

Elements:

$A_1, A_2, \ldots, A_i, \ldots, A_n$

Access index, insert at index, remove from index, sort list, find element, reverse list

remove first/all/last occurence(s) of element

Let us say: Delete all occurences of value
Search returns 0 or 1
Add to end of list.

Array based implementation

```
class List:
    allocate array arr of large enough size
    size = 0

    def append (element):        # O(1)
        arr[size] = element
        size++

    def search (element):        # O(n)
        for i = 0 to size-1:
            if arr[i] == element:
                return 1
        return 0
```

[Next page]

FABER-CASTELL
Date
Page No.

```
def delete(element):  # O(n)
    last = 0
    for i = 0 to size-1:
        if arr[i] != element:
            arr[last] = arr[i]
            last++
    size = last
```

	ArrayList	LinkedList
insert	~~O(n)~~ O(1)	~~O(1)~~ O(n)
delete	O(n)	~~O(1)~~ O(n)
search	O(n)	O(n)
size	O(1)	~~O(1)~~ O(n)

```
void LinkedList::print() {
    Node *curr = head ~~->next~~;
    while (curr) {
        curr->print();
        curr = curr->next;
    }
}
```

OR

```
for(Node* curr=head; curr;
    curr=curr->next) {
    curr->print();
}
```

```
void LinkedList::printRecursive(~~Node *nd~~) {   # invoke w/ printRecursive(head);
    if (nd) ~~->next~~ {
        nd->print();
        printRecursive(nd->next);
    }
}
```

↳ Interchanging the 2 lines prints in reverse order

$T(n) = O(1) + T(n-1)$ w/ $T(1) = O(1)$

$T(n) = O(n)$

FABER CASTELL

Date

Page No

```
void LinkedList::insert(datatype val) {
    Node * ptr = head;
    while (ptr->next) {
        ptr = ptr->next;
    }
    ptr->next = new Node;
    ptr->next->val = val;
    ptr->next->next = nullptr;
}
```

Special case of head == nullptr

```
if (ptr == nullptr) {
    head = new Node;
    head->val = val;
    head->next = nullptr;
    return;
}
```

```
void delete(datatype val) {
    for(Node ** pptr = &head; *pptr; ~~pptr = &((*pptr)->next)~~) {
        if ((*pptr)->val == val) { temp = (*pptr)->next;
            delete *pptr;
            *pptr = ~~(*pptr)->next;~~ temp;
        } else {
            pptr = &((*pptr)->next);
        }
    }
}
```

```
void LinkedList::delete(datatype val) {
    Node *prev = nullptr, *curr = head;
    while (curr) {
        if (curr->val == val) {
            if (prev) {
                prev->next = curr->next;
            } else {
                head = curr->next;
            }
            Node * temp = curr->next;
            delete curr;
            curr = temp;
        } else { prev = curr; curr = curr->next; }
    }
}
```

FABER-CASTELL
Date
Page No.

Doubly linked list:
head, tail stored

```
void insert (datatype val) {
    DNode *ptr = new DNode;
    ptr -> val = val;
    ptr -> prev = tail;
    ptr -> next = nullptr;
    if (tail) {
        tail -> next = ptr;
    }
    tail = ptr;
}
```

where,

```
class DNode {
    datatype val;
    DNode *prev, *next;
};
```

Q) String made of '{', '(', '[', ']', ')', '}'
check if balanced parentheses string
(i.e. a balanced parantheses string is a string which is either a concatenation of two ~~times~~ balanced parantheses strings or contains a balanced parantheses string inside a pair of matching brackets or is an empty string ~~pair of matching brackets~~):

```
bool checkBalancedParanthesesString (string s) {
    stack <char> st;
    for (char ch : s) {                    ; ← O(n)
        if (ch == '(') {
            st.push (')');
        } else if (ch == '{') {
            st.push ('}');
        } else if (ch == '[') {
            st.push (']');
        } else {if (st.empty() or st.top() != ch) {
            return false;
        } else {
            st.pop();
        }
    }
    return st.empty();
}
```

Stack ADT: Last In First Out
push, pop supported.
To implement using a singly linked list,
push ≡ inserting at front (at head)
pop ≡ deleting element ~~from~~ at head

Location		Prev	Next		
1000	N_1	null	2000	Head	1000
2000	N_2	1000	3000		
3000	N_3	2000	4000		
4000	N_4	3000	null	Tail	4000

Array - based Stack Implementation:

```
Allocate array arr
size := 0
def push(val):
    arr[size++] = val
def pop():
    return arr[--size]
def peek():
    return arr[size - 1]
def size():
    return size
def isempty():
    return size() == 0
```

Infix & Postfix Expressions:

((((2+(3×9))+8)-2) → Infix

2 3 9 × + 8 + 2 - → Postfix

InE: val base case

(InE1 op InE2) general case

PostE: val base case

PostE1 PostE2 op general case

Date
Page No.

```
string InFixToPostfix(string in){
    string post = "";
    stack<char> ops;
    for (char c : in) {
```

(7+9) x (2-4) = 7 9 + 2 4 - x = -32

7+(9x(2-4)) = 7 9 2 4 - x + = -11

(7+9)x2-4 = 7 9 + 2 x 4 - = 28

7+(9x2)-4 = 7 9 2 x + 4 - = 21

```
string in_to_post(string in) {      # needs brackets
    string post = "";               # around each operation
    stack<char> ops;

    for (char c:in) {
        if (c == '+' || c == '-' || c == '*' || c == '/') {
            post += " ";
            ops.push(c);
        } else if ('0' <= c && c <= '9') {
            post += c;
        } else if (c == ')') {
            post += " " + ops.pop();
            ops.pop();
        }
    }

    return post;
}
```

3+(7 * 5)+((6*2)+8) * 9)

```
def evaluate_post(string post):        # O(n)
    initialise a stack s

    for every token in post:
        if token is operator:
            b = s.pop()
            a = s.pop()
            s.push(a operator b)
        else:
            s.push(token)

    return s.pop()
```

Reverse a Linked List

```
def reverse(head):
    Node *prv = nullptr, *nxt = head->next;
    for (; nxt; prv = prv->next, nxt = nxt->next) {
        head->next = prev;
    Node *p = nullptr, *n = head->next;
    while (hea
```

[Next page]

FABER CASTELL

Date

Page No.

```
void List::reverse() {
    if (head==nullptr) return;   // redundant

    Node *curr=head, *prev=nullptr;   // can use head
    while (curr) {                    // instead of curr
        Node *temp=curr->next;
        curr->next=prev;
        prev=curr;
        curr=temp;
    }

    head=prev;
}
```

~~Node* void List::reverse(start=head){~~

```
void List::reverse() {
    head=reverseRecursive(head, nullptr);
}

Node* List::reverseRecursive(start, prev) {
    Node *tmp=start->next;
    start->next=prev;
    return tmp!=nullptr? reverseRecursive(tmp, start): start;
}
```

```
def in_to_post(string in):
    stack<char> ops; string postE;
    for symbol in E:
        if symbol is operand:
            postE += operand + " ";
        else if symbol is '(':
            ops.push('(');
        else if symbol is operator:
            ~~if symbol is '*' or '/':~~
            while (prec(symbol) <= prec(ops.top())):
                postE += ops.pop() + " ";
            ops.push(symbol)
        else if symbol is ')':
            while ops.top() != '(':
                postE += ops.pop() + " ";
            ops.pop();
    while ! ops.empty():
        postE += ops.pop() + " ";
    return postE;
```

Date
Page No.

Disjoint Set Problem

makeset(x), find(x), union(x,y)

```
class DisjointSets {
    map<int, int> parent, size;

    void makeset(int x) {  // O(1)
        parent[x] = x;
        size[x] = 1;
    }

    int find(int x) {  // O(α(n))
        if (parent.count(x)==0) return -1;  // or any other value
        if (parent[x] == x) return x;
        return parent[x] = find(parent[x]);
    }

    void union(int x, int y) {  // O(α(n))
        x = find(x);
        y = find(y);
        if (x != y) {
            if (size[x] < size[y]) swap(x, y);
            parent[y] = x;
            size[x] += size[y];
        }
    }
};
```

O(α(n)) × O(1)

FABER CASTELL
Date
Page No.

Queue

FIFO

enque() ≡ push_back(), deque() ≡ pop_front(), isempty()

List: Doubly Linked list, (or Singly w/ tail ptr)
enque() = insert at tail, dequeue() ≡ pop at head
isempty() ≡ head == nullptr

Array:
enque() ≡ push_back(), deque() ≡ pop_front
arr[end++] = val
return arr[start++]
isempty() ≡ start == end

~~class Queue {~~
~~vector int arr[] of sufficient size~~

class Queue {
ElementType arr[N];
int front, back, ~~size~~;

Queue() {
}

```
class Queue {
    ElementType arr[N];
    int front, back;

    Queue() : front(0), back(-1) {}

    bool isempty() {
        return back == -1;
    }

    void enqueue(ElementType val) {
        if (back >= 0 && (back+1)%N == front) return;
        ~~arr[front] = val;~~
        ~~back~~ back = (~~front~~ back + 1) % N;
        arr[back] = val;
    }

    ElementType dequeue() {
        if (back == -1) return 0 or -1;
        ~~back = (back+1) % N;~~
        ElementType ret_val = arr[~~back~~ front];
        front = (front+1) % N;
        if ((back+1)%N == front) {
            back = -1;
            front = 0;
        }
        return ret_val;
    }
};
```

queue of operator (initialised to n free operators)
queue of {users, time}

vector of struct {operator, user, time_left}
decrement time at every step
if 0, say user done & push operator to queue

For every request insert {op, front(), assg[front()]}

Queue from Stack:

2 stacks: in, out

```
def enqueue(val):        # O(1)
    in.push(val)

def dequeue():           # Amortised O(1)
    if out.isempty():
        while !in.isempty():
            out.push(in.pop)
    return out.pop()

def isempty():
    return in.isempty() && out.isempty()
```

Stack with

stacks st, mi

def push(val):
st push(va
if mi<st
mi
else
mi

def pop()
must po
return

def getmi
return

def isemp
retu

To do
store in
If mi

FABER CASTELL

Date

Page No.

Stack with getmin():

```
stacks: st, minst                    # O(n) space

def push(val):                       # O(1)
    st.push(val)
    if minst.isempty() || minst.top() >= val:
        minst.push(val)
    else:
        minst.push(minst.top())

def pop():                           # O(1)
    minst.pop()
    return st.pop()

def getmin():
    return minst.top()

def isempty():
    return st.isempty()
```

To do in O(1) space, maintain a current_min integer
store in stack the diff. from it.
If min. to be updated: corner case

Stack with getmin():

```
stack st
currmin := 0

def push(val):
    if (st.isempty() and st.top() < 0):
        st.currmin += st.top()

def push(val):
    if (st.empty() or val < currmin):
        st.push(val - currmin)

def push(val):
    st.push(val - currmin)
    if (val < currmin or st.size() == 1):
        currmin = val

def pop():
    if (st.top() < 0):
        curr_min -= st.top()
    return currmin + st.pop()

def top():
    return currmin + max(0, st.top())

def getmin():
    return currmin

def isempty():
    return st.isempty()
```

Trees

Hierarchical : Root points to subtrees (may be empty)
Unique path b/w any two points.
Length of path : number of edges in path
Depth of node : length of path from root
Depth of tree : maximum node depth in tree
Height of node : maximum length of path to any leaf in any of its subtrees.
Size of tree : number of nodes

$size(x) = 1 + \sum_{children} size(child)$ [size(null) = 0]
$depth(x) = 1 + depth(parent)$ [depth(root) = 0] or d(null) = -1
$height(x) = 1 + \max_{children}(height(child))$ [height(leaf) = 0] or h(null) = -1

Coding

36 characters : each character is a 6-bit string
Improvement: Shorter codes for frequent characters
Need for unambiguous encoding

Eg.		freq	code
	a	9	1000
	b	10	1001
	c	15	01
	d	4	101
	e	25	00
	i	30	11

Property: No code is a prefix of another code
Eg i.e prefix code

An ancestor is the parent or an ancestor of the parent.
An ancestor is any node from which a directed path to the node exists.

FABER CASTELL
Date
Page No.

Binary Trees

Every node has a left child and a right child, both of which can be null (no child).

```
struct BTreeNode {
    int val;
    BTreeNode *left, *right;
};
```

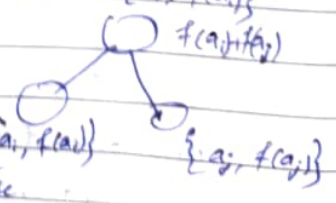
Generating Codes

Inputs : a_i - characters
$f(a_i)$ - frequency of occurence

Code s.t. no code is a prefix of another. (the output string is unambiguously decodable)
Objective: code with smallest output string
Sum of Products of lengths of code and frequencies is to be minimised.

If left denotes 0 & right denotes 1, and leaf nodes are assigned characters, we must minimise sum of products of depths of nodes & frequencies of the nodes.

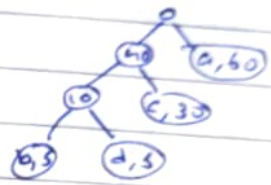
∴ We create nodes with values $\{a_i, f(a_i)\}$
We combine nodes to make (root) $f(a_i)+f(a_j)$
We keep combining the nodes with smallest values until we are left with one (root) node.



```
int n;
vector<int> a, f;
[input n, a, f]
priority_queue<int, vector<int>, greater<int>> pq;
for (int i=0; i<n; ++i) {
```

```
struct Node {
    int f;
    char c;
    Node *left, *right;
};
```

a	60	1
b	5	000
c	30	01
d	5	001




```
int n;
cin >> n;
vector<char> a(n);
vector<int> f(n);
for (int i=0; i<n; ++i) cin >> a[i] >> f[i];
```

```
priority_queue pq
for i = 1 to n:
    Create node with a[i], f[i]
    Insert to pq
for i = 1 to n-1:
    l = pq.pop()
    r = pq.pop()
    Create node with left child l & right child r,
    f = l.f + r.f
    Insert new node to pq
```

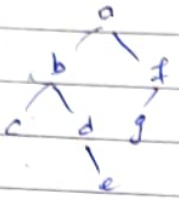
pq.top() is a tree representing the code

Full trees : Nodes either have 0 or 2 children

n inserts, n deletions

priority_queue : both operations in $\log n$

∴ Overall time complexity: $n \log n$



Inorder: cbdeagf

Pre: abcdefg

Post: cedbgfa