Recitation 9 (HW8)

Online: Xinyi Zhao xz2833@nyu.edu

GCASL 461: Yifan Jin yj2063@nyu.edu

New York University

Basic Algorithms (CSCI-UA.0310-005)

Problem 1

Problem 1 (20 points)

Recall the activity selection problem discussed in the lecture. We developed a greedy approach which in every step picks the interval with the earliest finish point among the remaining ones. Modify it in order to develop a different greedy algorithm picking the first interval based on another property. Show that your greedy algorithm actually works (You can assume that the greedy approach discussed in the lecture gives an optimal answer and may use it to prove the correctness of your algorithm).

Problem 1

Activity Selection Problem

Input: $T_1 = [s_1, f_1)$ — n tasks

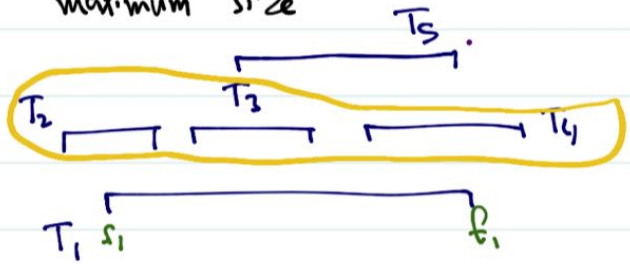
$T_2 = [s_2, f_2)$

$\vdots$

$T_n = [s_n, f_n)$

Output: find a subset of pairwise non-intersecting tasks of maximum size

Example

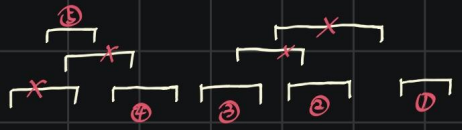


Output = 3 choose $\{T_2, T_3, T_4\}$

Problem 1

Idea : the optimal first interval is the one with the maximum starting time

Example :



output = 5

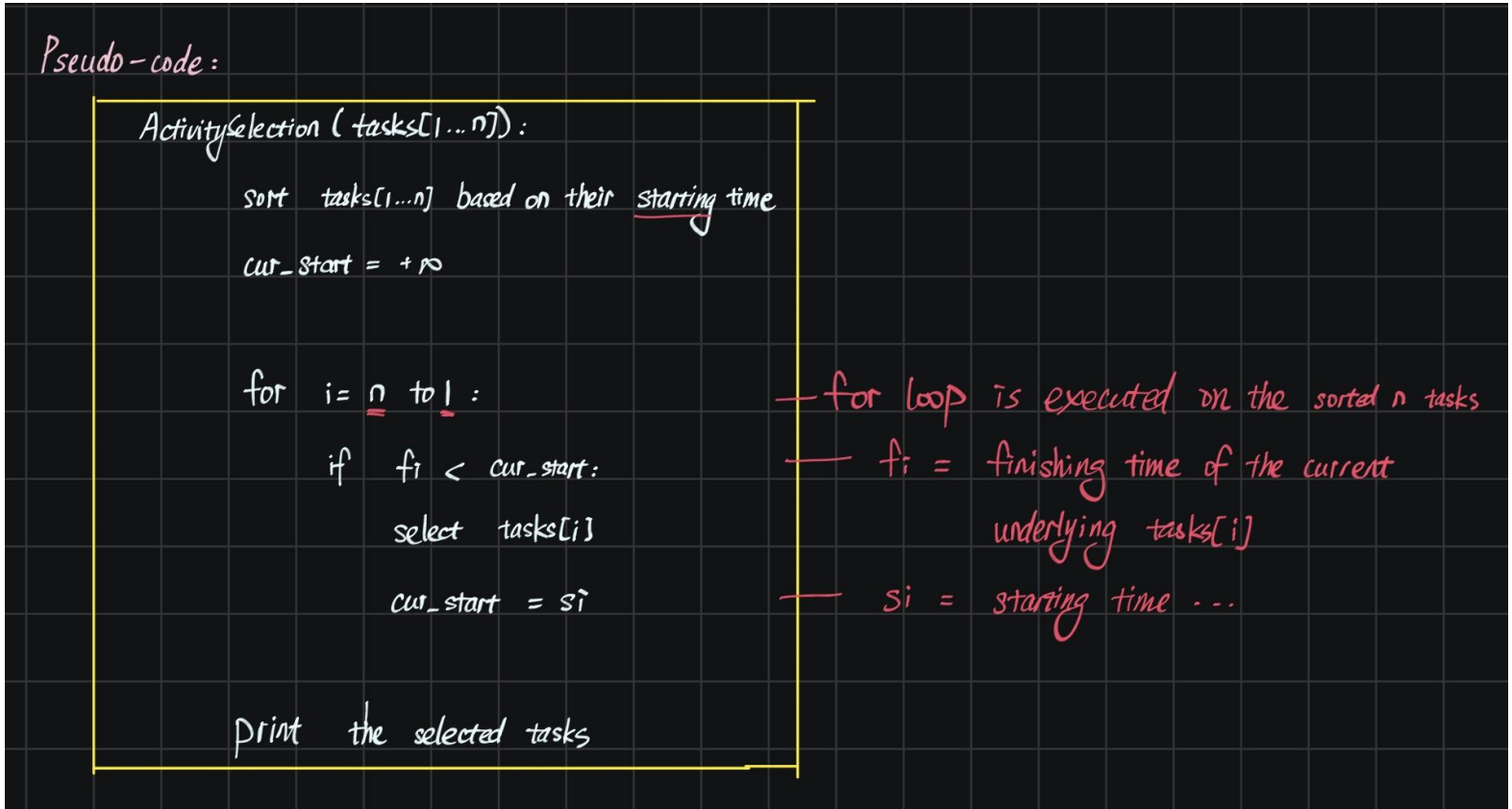
Greedy Approach :

(1) sort the tasks based on their starting time

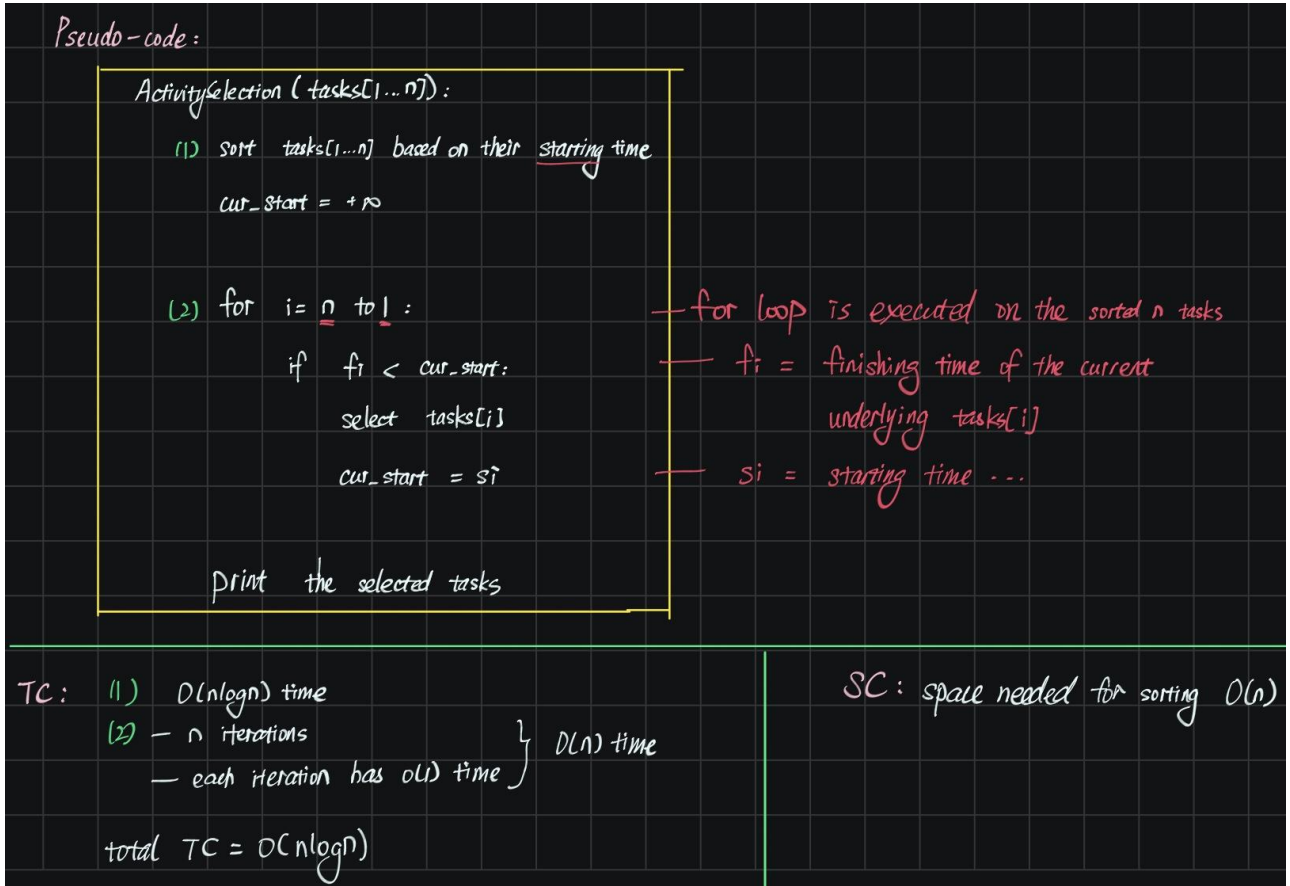
(2) pick the interval with the latest starting time among all the remaining intervals and discard all the intervals it intersects

(3) repeat part (2) until no interval is left

Problem 1


Pseudo-code:
ActivitySelection (tasks[1...n]) :
sort tasks[1...n] based on their starting time
cur_start = +∞
for i = n to 1 :
if fi < cur_start:
select tasks[i]
cur_start = si
print the selected tasks
— for loop is executed on the sorted n tasks
— fi = finishing time of the current underlying tasks[i]
— si = starting time ...

Problem 1


Pseudo-code:
ActivitySelection (tasks[1...n]):
(1) sort tasks[1...n] based on their starting time
cur_start = +∞
(2) for i= n to 1 :
if fi < cur_start:
select tasks[i]
cur_start = si
print the selected tasks
— for loop is executed on the sorted n tasks
— fi = finishing time of the current underlying tasks[i]
— si = starting time ...
TC: (1) O(nlogn) time
(2) — n iterations
— each iteration has o(1) time
O(n) time
total TC = O(nlogn)
SC: space needed for sorting O(n)

Problem 1

Proof of correctness:

Lemma At each iteration, taking the interval with the max starting point among the remaining intervals (that don't intersect the previously chosen interval) gives an optimal non-overlapping subset of intervals.

Problem 1

Proof Suppose there is an optimal subset of intervals (S) whose last interval (J) (the interval of S with max starting point) is Not the interval with max starting point among all the remaining intervals (I).

- J doesn't intersect any other interval of S
- I has a starting point larger than the starting point of J

⇒ The only interval from S that I may intersect is J.

Construct a new optimal subset S':

$$S' = (S \setminus \{J\}) \cup \{I\}$$

S' is an optimal subset of intervals since:

- S' has the same number of intervals of S
- Intervals in S' are pairwise non-overlapping.

Thus, we have an optimal subset of intervals S' that includes I. This finishes the proof.

Problem 2

Problem 2 (15 points)

Recall that if the input graph $G = (V, E)$ is given by an array of adjacency lists, then the running time of the BFS algorithm on G is $\Theta(|V| + |E|)$. Now suppose G is given using an adjacency matrix. Rewrite the pseudo-code for the BFS algorithm on G using its adjacency matrix. Find the time and space complexities using Θ expressions.

Problem 2

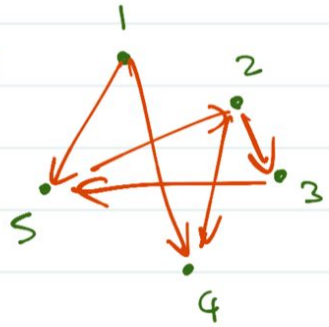
Two ways of representing graphs as input:

- Adjacency lists

(i) adjacency lists: an array of lists

$V[1 \dots n]$: $V[i]$ is a linked list consisting of the vertices to which there is an edge from vertex i

Example



$V[1 \dots 5]$

$V[1] = 4 \to 5$: linked list

$V[2] = 3 \to 4$

$V[3] = 5$

$V[4] = \phi$

$V[5] = 2$

- efficient for storage: $O(|V|+|E|)$
- slow for deciding adjacencies: time to answer $i \to j$: $O(n)$

Problem 2

Two ways of representing graphs as input:

- Adjacency matrix

(ii) adjacency matrix: array of arrays (2D-array)

$V[1..5][1..5] =$

	1	2	3	4	5
1	0	0	0	1	1
2	0	0	1	1	0
3			⋮		
4					
5					

n×n

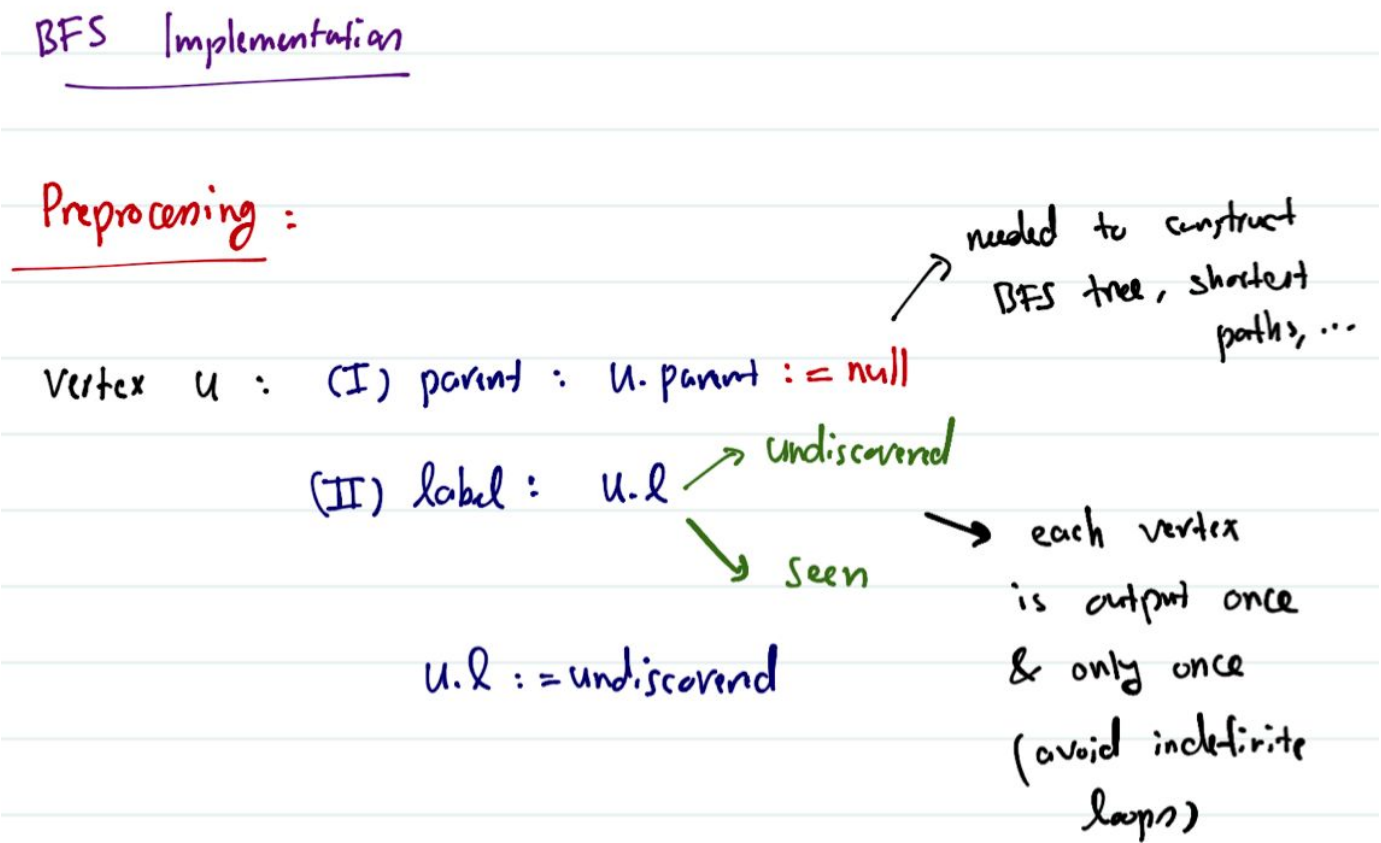
$$V[i][j] = \begin{cases} 1 & \text{if there is an edge } i \to j \\ 0 & \text{O.W.} \end{cases}$$

- efficient for deciding adjacencies:

  time needed to check $i \xrightarrow{\text{edge}} j$: $O(1)$

- storage efficiency is low: $\theta(n^2) = \theta(|V|^2)$

Problem 2


BFS Implementation
Preprocessing :
Vertex u : (I) parent : u.parent := null
needed to construct BFS tree, shortest paths, ...
(II) label : u.l
undiscovered
seen
u.l := undiscovered
each vertex is output once & only once (avoid indefinite loops)

Problem 2

BFS(G, S) with adjacency lists

Implementation

```
BFS (G, s)

    queue Q := {s}                    Q: seen list

    s.l = seen

    while Q is non-empty
        u = Q.front()
        print ("u")
        Q.pop()
        for each vertex v in V[u]     edge u → v
            if (v.l == undiscovered)
                Q.push(v)
                v.l = seen
                v.parent = u
```

Problem 2

BFS(G, S) with adjacency matrix

Implementation (adjacency matrix)

```
BFS(G, S):
    queue Q := {s}
    s.l = seen
    while Q is non-empty:
        u = Q.front()
        print("u")
        Q.pop()
        for v = 1 to n:
            if (V[u][v] == 1) and (v.l == undiscovered):
                Q.push(v)
                v.l = seen
                v.parent = u
```

edge $u \to v$

Problem 2

```
BFS(G):
    for each vertex s in V(G):
        if (s.il == undiscovered):
            BFS(G, s)
```

TC of BFS(G):

- Each vertex in ~~for~~-loop of BFS(G) is visited once: $\Theta(|V|)$
- TC of aux calls to BFS(G, s): $\Theta(|V|)$

$\Rightarrow$ TC of BFS(G) = $\Theta(|V|) \cdot \Theta(|V|) = \Theta(|V|^2)$

SC of BFS(G): queue Q

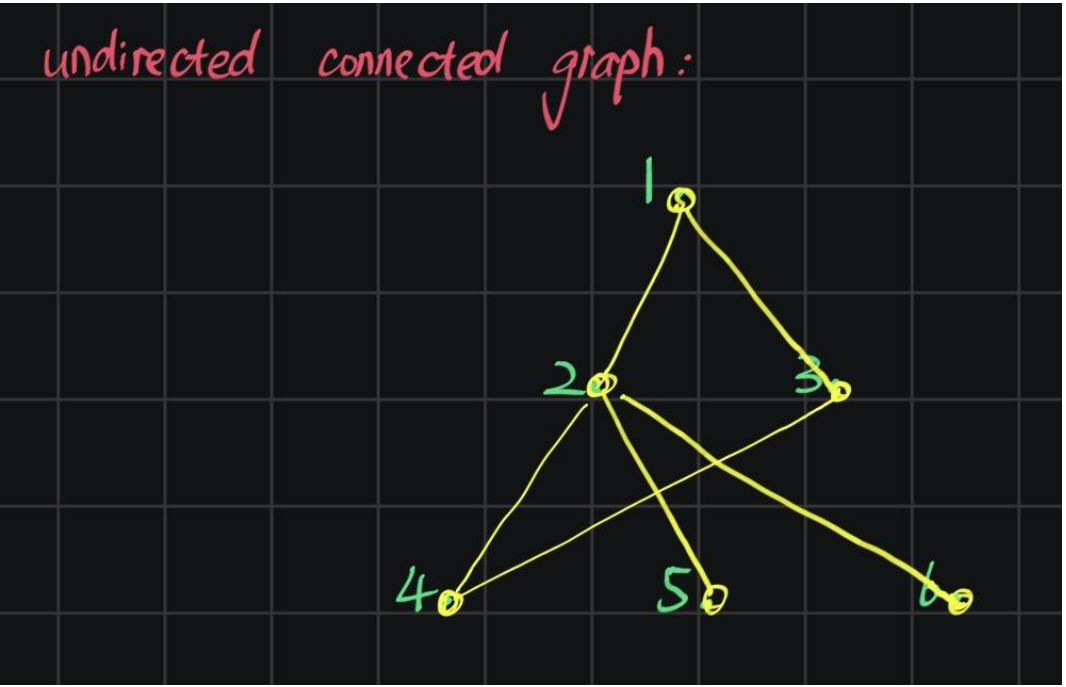
SC = $\Theta(|V|)$

Problem 3

Problem 3 (15 points)

Given an undirected graph $G = (V, E)$, develop an $O(|V| + |E|)$-time algorithm to check whether G is connected. Write the pseudo-code of your algorithm and justify why your algorithm satisfies the required running time.

Problem 3



What can we get if we run BFS(G,1)?

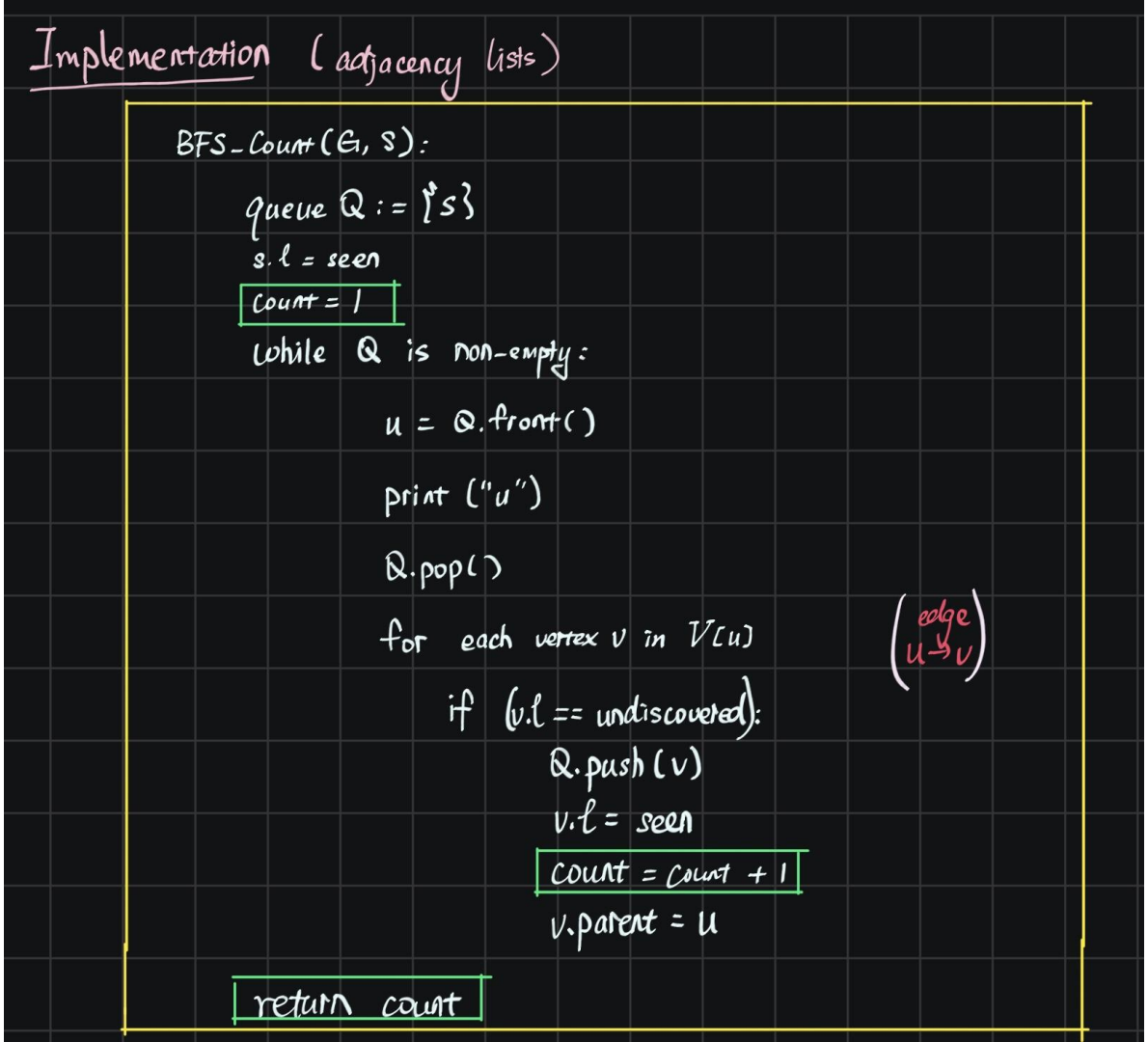
=>

1, 2, 3, 4, 5, 6

=> it will give us all the vertices connected to “1”

=> the number of vertices connected to “1” is equal to the number of all vertices in graph G

Problem 3


Implementation (adjacency lists)
BFS_Count(G, s):
queue Q := {s}
s.l = seen
count = 1
while Q is non-empty:
u = Q.front()
print ("u")
Q.pop()
for each vertex v in V[u]
(edge u→v)
if (v.l == undiscovered):
Q.push(v)
v.l = seen
count = count + 1
v.parent = u
return count

Problem 3

```
check_connected(G):
    n = |V|
    s = pick any vertex in G
    count = BFS_count(G, s)
    if n == count:
        return true
    else:
        return false
```

TC of BFS_count(G, s): $TC = \Theta(|V|+|E|)$

SC of BFS_count(G, s): $SC = \Theta(|V|)$

Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

Basic idea: check each vertex

All edges coming to it from all other vertices

0	**1**	**1**	**1**
0	**0**	**0**	**0**
1	**1**	**0**	**0**
0	**1**	**1**	**0**

-> the corresponding column is full of 1 except itself

No edges going out from it -> the corresponding row is full of 0.

Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

Basic idea: check each vertex

All edges coming to it from all other vertices

-> the corresponding column is full of 1 except itself

No edges going out from it -> the corresponding row is full of 0.

0	1	1	1
0	0	0	0
1	1	0	0
0	1	1	0

Each check is O(n)
Total running time: O(n^2)

Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

Optimization: only check one possible vertex

How?

Hint: All edges coming to it from all other vertices, No edges going out from it

Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

Step 1:

Since **all edges coming to it from all other vertices,** starting from arbitrary vertices, the goal vertex (call it **vertex X**) must be directly reachable(if exists).

In other words, vertex X must exists among the vertices with a directed edge from arbitrary starting vertex.

0	**1**	**1**	**1**
0	**0**	**0**	**0**
1	**1**	**0**	**0**
0	**1**	**1**	**0**

Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

Step 1:

Only need to find vertex X among these vertices

Since **all edges coming to it from all other vertices,** starting from arbitrary vertices, the goal vertex (call it **vertex X**) must be directly reachable(if exists).

In other words, vertex X must exists among the vertices with a directed edge from arbitrary starting vertex.

0	**1**	**1**	**1**
0	**0**	**0**	**0**
1	**1**	**0**	**0**
0	**1**	**1**	**0**

Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

Step 2: How to find the possible vertex X?

Hint: **All edges coming to it from all other vertices, No edges going out from it**

Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

Step 2: How to find the possible vertex X?

Hint: **All edges coming to it from all other vertices, No edges going out from it**

For any pair of vertices from step 1,

Eliminate the nodes which have a edge going out

Eliminate the nodes which don't have a edge coming in.

Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

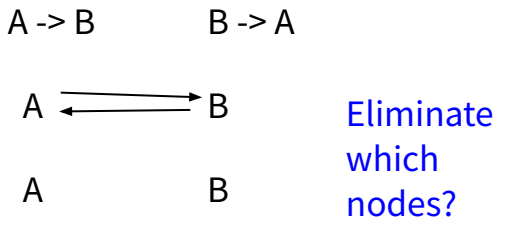
Step 2: How to find the possible vertex X?

Hint: **All edges coming to it from all other vertices, No edges going out from it**

For any pair of vertices from step 1,

Eliminate the nodes which have a edge going out

Eliminate the nodes which don't have a edge coming in.



Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

Step 2: How to find the possible vertex X?

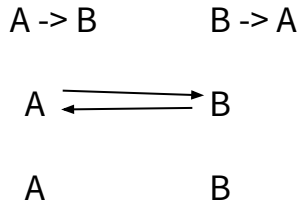
Hint: **All edges coming to it from all other vertices, No edges going out from it**

For any pair of vertices from step 1,

Eliminate the nodes which have a edge going out

Eliminate the nodes which don't have a edge coming in.

O(1) for each elimination



Eliminate which nodes?

Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

Step 3:

After step 2, there is at most one vertex remaining.

(All other vertices satisfy the condition of elimination)

Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

Step 3:

After step 2, there is at most one vertex remaining.

(All other vertices satisfy the condition of elimination)

If there remains one vertex, check it

If there is no vertex remaining, report 'No Such Vertex'.

Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

Step 1: Starting from arbitrary vertices, record all directly reachable vertices

Step 2: For any pair of vertices from step 1,

Eliminate the nodes which have a edge going out

Eliminate the nodes which don't have a edge coming in.

Step 3: check the possible one remaining vertex

Problem 4

Problem 4 (25 points)

Let G be a directed graph on n vertices which is given as input by the adjacency matrix $V[1 \ldots n][1 \ldots n]$. Develop an $O(n)$-time algorithm to check if there is any vertex in G that has edges coming to it from all other vertices of G but no edges going out from it. Justify why your algorithm runs in $O(n)$ time.

Step 1: Starting from arbitrary vertices, record all directly reachable vertices

O(n),
at most n-1
vertices selected

Step 2: For any pair of vertices from step 1,

Eliminate the nodes which have a edge going out

Eliminate the nodes which don't have a edge coming in.

Each time eliminates 1 or 2 vertices
Thus at most eliminates N-1 times.
Each elimination is O(1)
Totally O(n)

Step 3: check the possible one remaining vertex

O(n)

Problem 5

Problem 5 (13+12 points)

We want to develop a divide and conquer-based algorithm to find the convex hull of n given points.

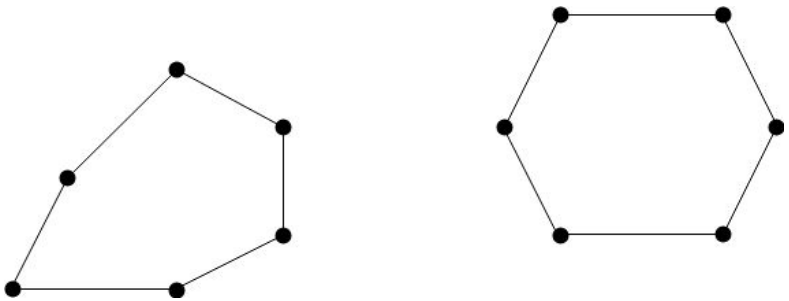
(a) First, given two convex polygons P and Q which have n points in total, develop an $O(n)$-time algorithm to find the convex hull of their union.
You have to fully explain your algorithm. You do NOT need to write the pseudo-code of your algorithm.

(b) Given n points in the plane, use part (a) to develop a divide and conquer-based algorithm to find their convex hull. Your algorithm must work in $O(n \log n)$ time.
Write the pseudo-code of your algorithm and justify why it satisfies the required running time.

Problem 5

(a) First, given two convex polygons P and Q which have n points in total, develop an $O(n)$-time algorithm to find the convex hull of their union.
You have to fully explain your algorithm. You do NOT need to write the pseudo-code of your algorithm.

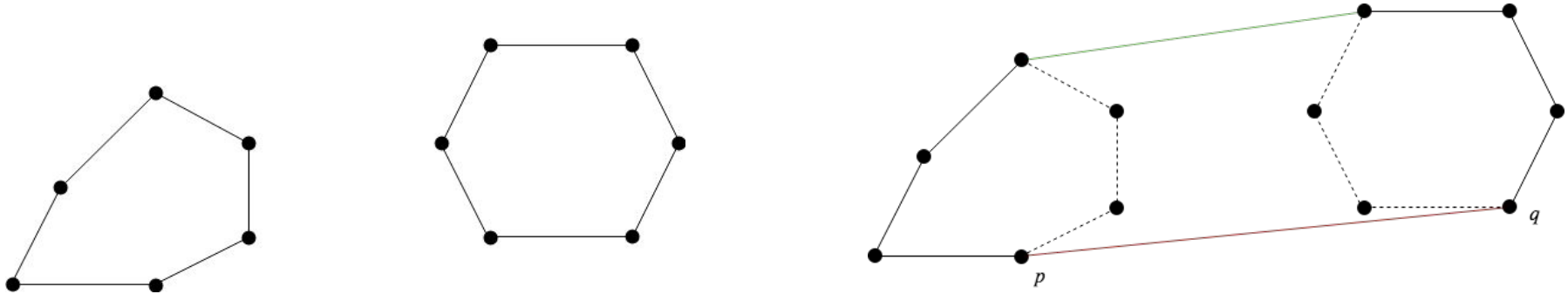
For simplicity, we assume two convex polygons are non-overlapped.



Problem 5

(a) First, given two convex polygons P and Q which have n points in total, develop an $O(n)$-time algorithm to find the convex hull of their union.
You have to fully explain your algorithm. You do NOT need to write the pseudo-code of your algorithm.

For simplicity, we assume two convex polygons are non-overlapped.



It's easy to infer that we only need to add two edges and delete several edges.

Problem 5

(a) First, given two convex polygons P and Q which have n points in total, develop an $O(n)$-time algorithm to find the convex hull of their union.
You have to fully explain your algorithm. You do NOT need to write the pseudo-code of your algorithm.

For simplicity, we assume two convex polygons are non-overlapped.

Reference:
https://algorithmtutor.com/Computational-Geometry/An-efficient-way-of-merging-two-convex-hull/

Problem 5

(b) Given n points in the plane, use part (a) to develop a divide and conquer-based algorithm to find their convex hull. Your algorithm must work in $O(n \log n)$ time.
Write the pseudo-code of your algorithm and justify why it satisfies the required running time.

Problem 5

(b) Given n points in the plane, use part (a) to develop a divide and conquer-based algorithm to find their convex hull. Your algorithm must work in $O(n \log n)$ time.
Write the pseudo-code of your algorithm and justify why it satisfies the required running time.

Just like the Merge-Sort function (the most classic Divide and Conquer algorithm):

```
1 Find-Convex-Hull(A):
2     if(length(A) <= 4):
3         construct convex hull
4         return;
5     Find-Convex-Hull(left of A)
6     Find-Convex-Hull(right of A)
7     Merge-Convex-Hull(A)
```

```
MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)
```

Problem 5

(b) Given n points in the plane, use part (a) to develop a divide and conquer-based algorithm to find their convex hull. Your algorithm must work in $O(n \log n)$ time.
Write the pseudo-code of your algorithm and justify why it satisfies the required running time.

Just like the Merge-Sort function (the most classic Divide and Conquer algorithm):

```
1 Find-Convex-Hull(A):
2     if(length(A) <= 4):        O(1)
3         construct convex hull
4         return;
5     Find-Convex-Hull(left of A)
6     Find-Convex-Hull(right of A)
7     Merge-Convex-Hull(A)
```

Codes are in the link in part(a), Running time: O(n)

```
MergeSort(A, p, r):
    if p > r
        return
    q = (p+r)/2
    mergeSort(A, p, q)
    mergeSort(A, q+1, r)
    merge(A, p, q, r)
```

Problem 5

(b) Given n points in the plane, use part (a) to develop a divide and conquer-based algorithm to find their convex hull. Your algorithm must work in $O(n \log n)$ time.
Write the pseudo-code of your algorithm and justify why it satisfies the required running time.

One question: How to guarantee that each time the two convex hull are non-overlapped, as we assume in part(a)?

Problem 5

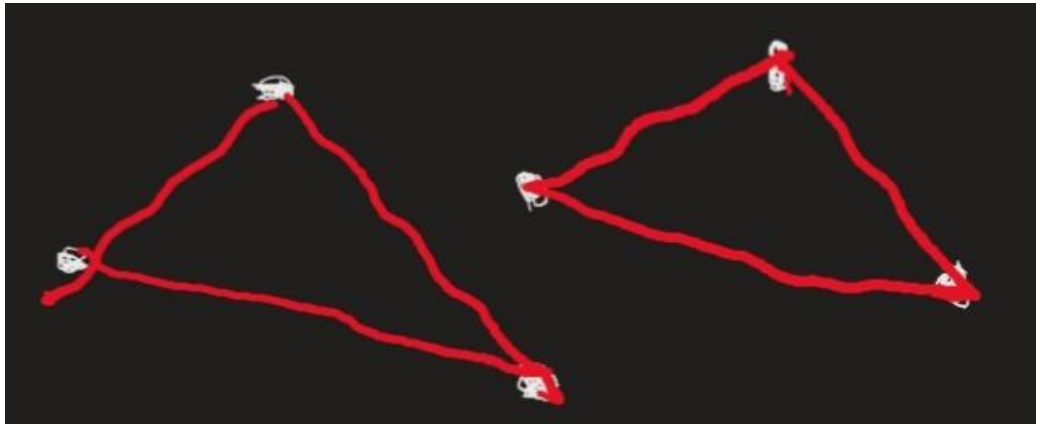
(b) Given n points in the plane, use part (a) to develop a divide and conquer-based algorithm to find their convex hull. Your algorithm must work in $O(n \log n)$ time.
Write the pseudo-code of your algorithm and justify why it satisfies the required running time.

One question: How to guarantee that each time the two convex hull are non-overlapped, as we assume in part(a)?

Initialization:

Sort all nodes based on X index

(and on Y index if tied)



Problem 5

(b) Given n points in the plane, use part (a) to develop a divide and conquer-based algorithm to find their convex hull. Your algorithm must work in $O(n \log n)$ time.
Write the pseudo-code of your algorithm and justify why it satisfies the required running time.

Overall Algorithm:

1. Sort A based on X index (and Y index if tied)
2.

```
Find-Convex-Hull(A):
    if(length(A) <= 4):
        construct convex hull
        return;
    Find-Convex-Hull(left of A)
    Find-Convex-Hull(right of A)
    Merge-Convex-Hull(A)
```

Invocation call:

Find-Convex-Hull(A)

Problem 5

(b) Given n points in the plane, use part (a) to develop a divide and conquer-based algorithm to find their convex hull. Your algorithm must work in $O(n \log n)$ time.
Write the pseudo-code of your algorithm and justify why it satisfies the required running time.

Overall Algorithm:

1. Sort A based on X index (and Y index if tied) O(n logn)
2.

```
1 Find-Convex-Hull(A):
2     if(length(A) <= 4):
3         construct convex hull
4         return;
5     Find-Convex-Hull(left of A)
6     Find-Convex-Hull(right of A)
7     Merge-Convex-Hull(A)
```

Invocation call:

Find-Convex-Hull(A)

$T(n) = 2\,T(n/2) + O(n)$
$T(n) = O(n \log n)$

Q & A

Thank you