CSCI 270

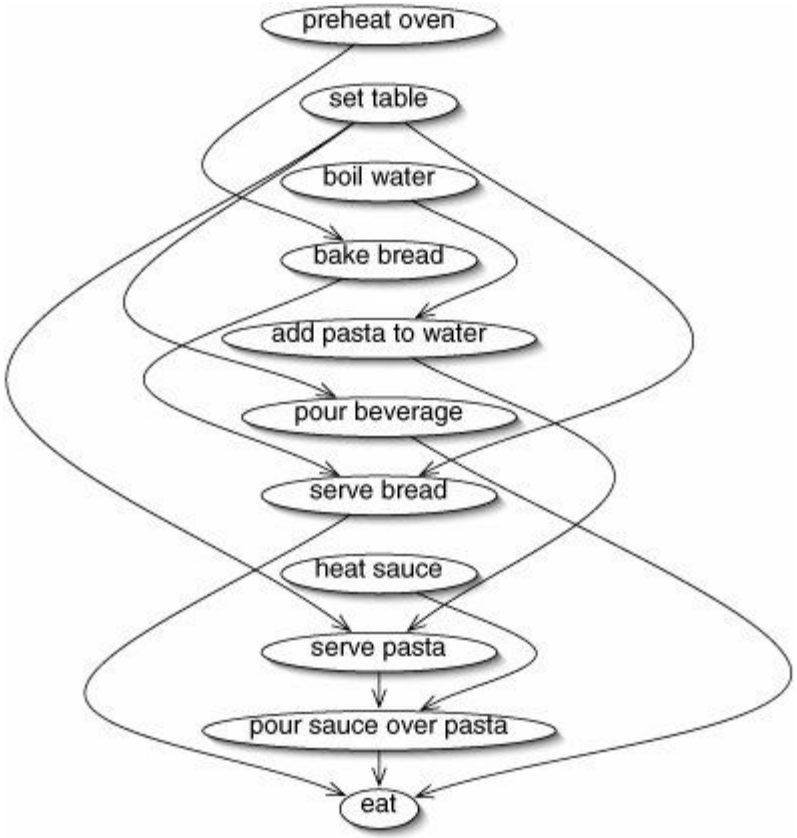
Discussion - Week 2

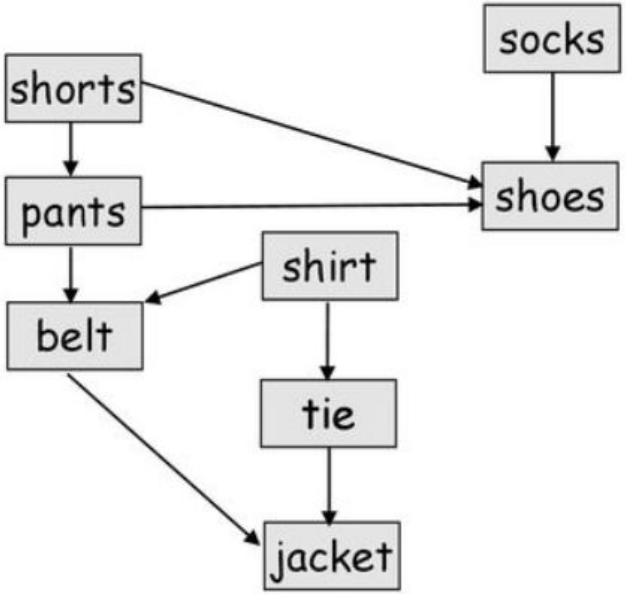
Topological Ordering

A topological sort or topological ordering of a Directed Acyclic Graph is a linear ordering of its vertices such that if there is a directed edge from any vertex *u* to *v*, then *u* must be before *v* in the ordering (NOT CONSECUTIVE NECESSARILY)

Example:

- Scheduling jobs based on dependencies
- ordering of formula cell evaluation when recomputing formula values in spreadsheets


preheat oven
set table
boil water
bake bread
add pasta to water
pour beverage
serve bread
heat sauce
serve pasta
pour sauce over pasta
eat

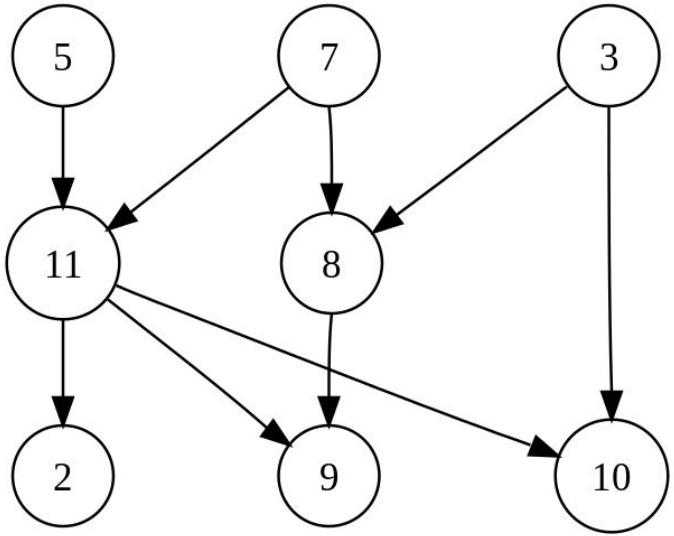

socks
shorts
shoes
pants
shirt
belt
tie
jacket

Ordering example

Different possible orders:

- 5, 7, 3, 11, 8, 2, 9, 10
- 3, 5, 7, 8, 11, 2, 9, 10
- 5, 7, 3, 8, 11, 10, 9, 2
- 7, 5, 11, 3, 10, 8, 9, 2
- 5, 7, 11, 2, 3, 8, 9, 10
- 3, 7, 8, 5, 11, 10, 2, 9

  …….



Algorithm

map = vertex -> its indegree

queue = all vertices with in-degree as 0

Topo_order = []

While (queue is not empty):

    Dequeue to get a vertex u

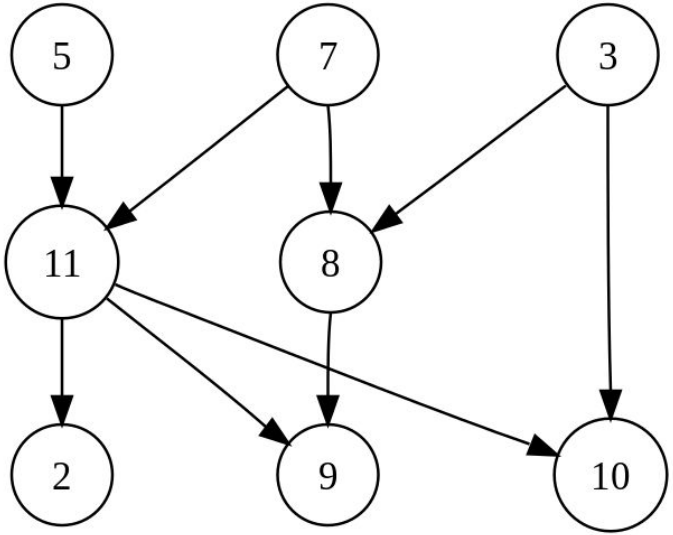
    topo_order.append(u)

    For all outgoing edges (u,v):

        map[v] = map[v]-1 // decrease in-degree of v by 1

        If map[v] == 0: // all incoming edges taken care of

            enqueue(queue, u)



Map: {5:0, 7:0, 3:0, 11:2, 8:2, 2:1, 9:2, 10:2}
Queue: 5,7,3
Topo_order: []

Algorithm

map = vertex -> its indegree

queue = all vertices with in-degree as 0

Topo_order = []

While (queue is not empty):

    Dequeue to get a vertex u

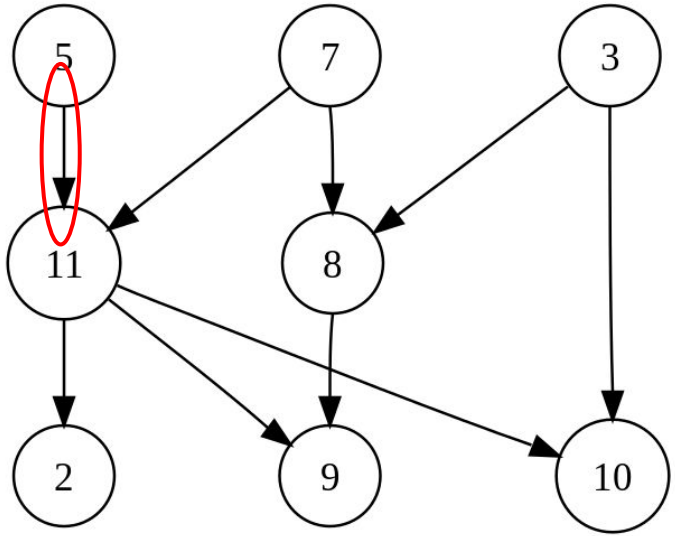
    topo_order.append(u)

    For all outgoing edges (u,v):

        map[v] = map[v]-1 // decrease in-degree of v by 1

        If map[v] == 0: // all incoming edges taken care of

            enqueue(queue, u)



Map: {5:0, 7:0, 3:0, **11:1**, 8:2, 2:1, 9:2, 10:2}
Queue: 7,3
Topo_order: 5,

Algorithm

map = vertex -> its indegree

queue = all vertices with in-degree as 0

Topo_order = []

While (queue is not empty):

    Dequeue to get a vertex u

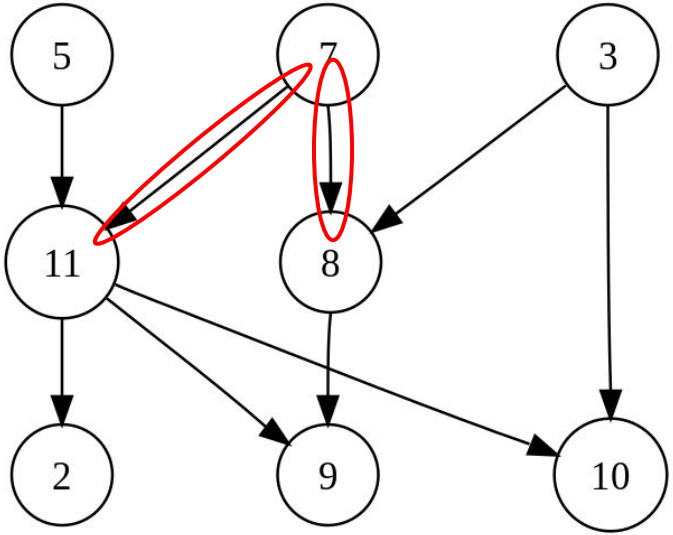
    topo_order.append(u)

    For all outgoing edges (u,v):

        map[v] = map[v]-1 // decrease in-degree of v by 1

        If map[v] == 0: // all incoming edges taken care of

            enqueue(queue, u)



Map: {5:0, 7:0, 3:0, **11:0**, **8:1**, 2:1, 9:2, 10:2}
Queue: 3,11
Topo_order: 5,7

Algorithm

map = vertex -> its indegree

queue = all vertices with in-degree as 0

Topo_order = []

While (queue is not empty):

    Dequeue to get a vertex u

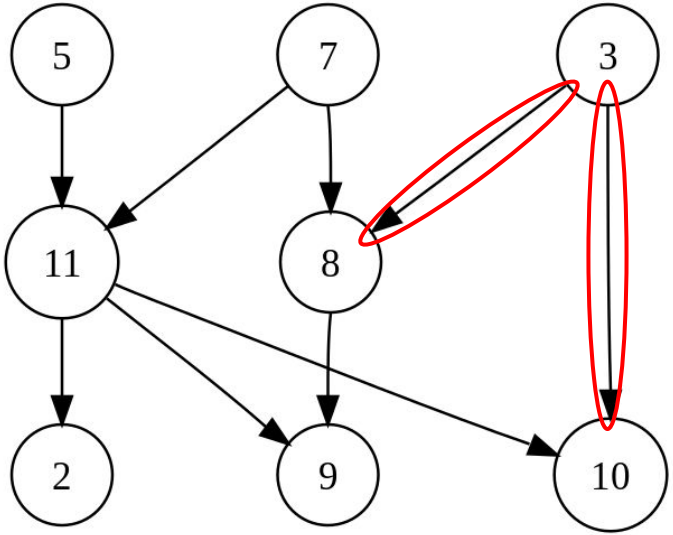
    topo_order.append(u)

    For all outgoing edges (u,v):

        map[v] = map[v]-1 // decrease in-degree of v by 1

        If map[v] == 0: // all incoming edges taken care of

            enqueue(queue, u)



Map: {5:0, 7:0, 3:0, 11:0, **8:0**, 2:1, 9:2, **10:1**}
Queue: 11,8
Topo_order: 5,7,3

Algorithm

map = vertex -> its indegree

queue = all vertices with in-degree as 0

Topo_order = []

While (queue is not empty):

    Dequeue to get a vertex u

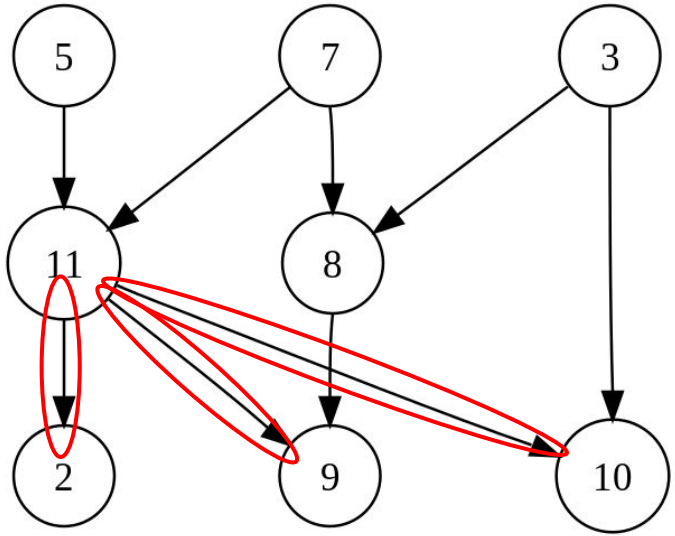
    topo_order.append(u)

    For all outgoing edges (u,v):

        map[v] = map[v]-1 // decrease in-degree of v by 1

        If map[v] == 0: // all incoming edges taken care of

            enqueue(queue, u)



Map: {5:0, 7:0, 3:0, 11:0, 8:0, **2:0**, **9:1**, **10:0**}
Queue: 8,2,10
Topo_order: 5,7,3,11

Algorithm

map = vertex -> its indegree

queue = all vertices with in-degree as 0

Topo_order = []

While (queue is not empty):

    Dequeue to get a vertex u

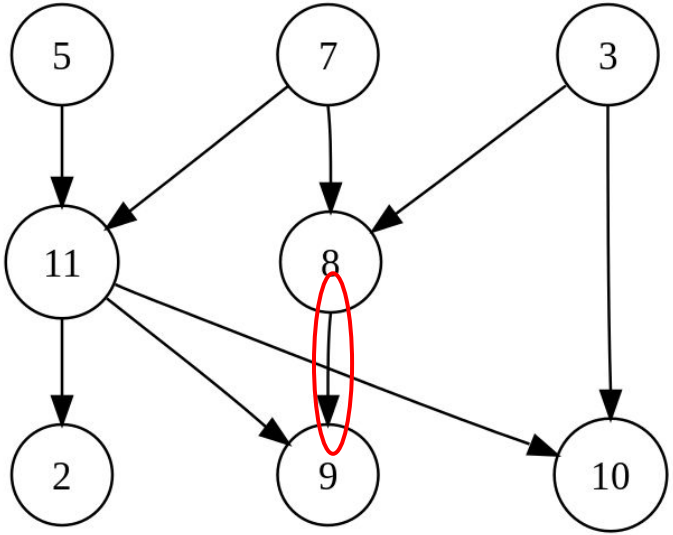
    topo_order.append(u)

    For all outgoing edges (u,v):

        map[v] = map[v]-1 // decrease in-degree of v by 1

        If map[v] == 0: // all incoming edges taken care of

            enqueue(queue, u)



Map: {5:0, 7:0, 3:0, 11:0, 8:0, 2:0, **9:0**, 10:0}
Queue: 2,10,9
Topo_order: 5,7,3,11,8

Algorithm

map = vertex -> its indegree

queue = all vertices with in-degree as 0

Topo_order = []

While (queue is not empty):

    Dequeue to get a vertex u

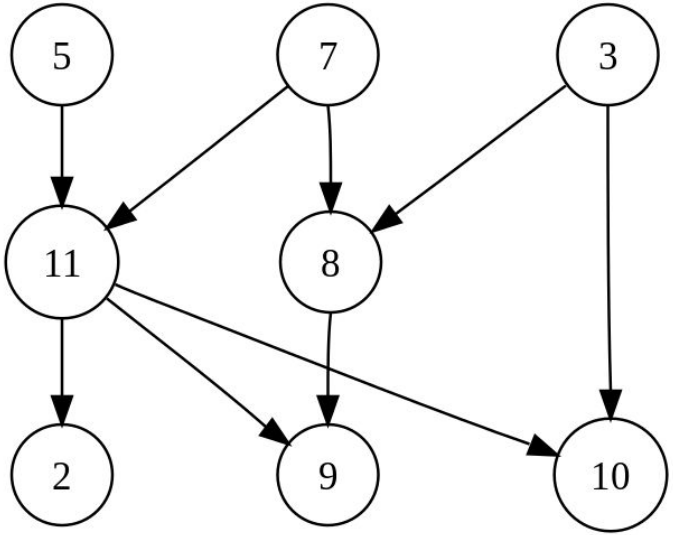
    topo_order.append(u)

    For all outgoing edges (u,v):

        map[v] = map[v]-1 // decrease in-degree of v by 1

        If map[v] == 0: // all incoming edges taken care of

            enqueue(queue, u)



Map: {5:0, 7:0, 3:0, 11:0, 8:0, 2:0, 9:0, 10:0}
Queue: 10,9
Topo_order: 5,7,3,11,8,2

Algorithm

map = vertex -> its indegree

queue = all vertices with in-degree as 0

Topo_order = []

While (queue is not empty):

    Dequeue to get a vertex u

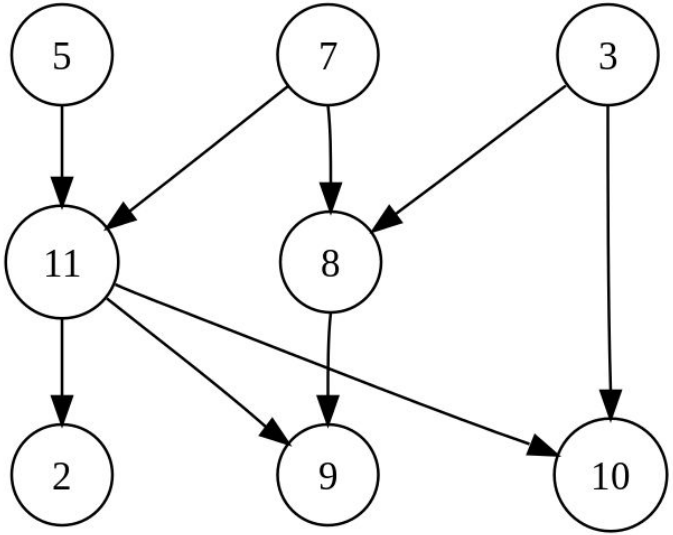
    topo_order.append(u)

    For all outgoing edges (u,v):

        map[v] = map[v]-1 // decrease in-degree of v by 1

        If map[v] == 0: // all incoming edges taken care of

            enqueue(queue, u)



Map: {5:0, 7:0, 3:0, 11:0, 8:0, 2:0, 9:0, 10:0}
Queue: 9
Topo_order: 5,7,3,11,8,2,10

Algorithm

map = vertex -> its indegree

queue = all vertices with in-degree as 0

Topo_order = []

While (queue is not empty):

    Dequeue to get a vertex u

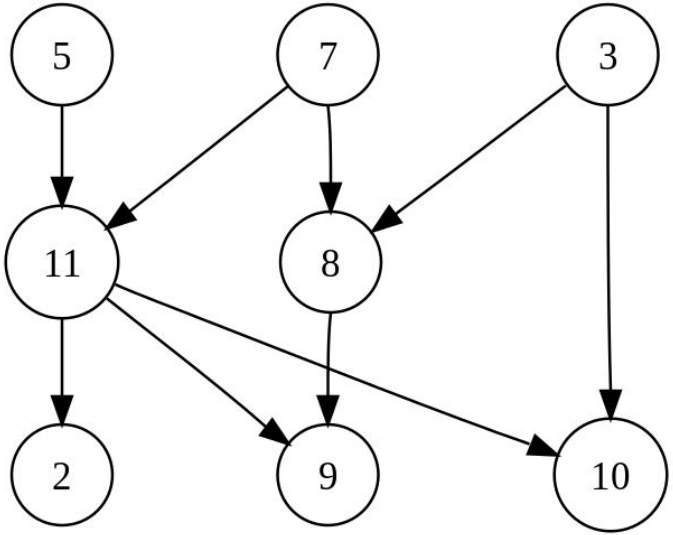
    topo_order.append(u)

    For all outgoing edges (u,v):

        map[v] = map[v]-1 // decrease in-degree of v by 1

        If map[v] == 0: // all incoming edges taken care of

            enqueue(queue, u)



Map: {5:0, 7:0, 3:0, 11:0, 8:0, 2:0, 9:0, 10:0}
Queue:
Topo_order: 5,7,3,11,8,2,10,9

Complexity Analysis

Creating map - O(E)

Enqueue-ing and dequeue-ing each vertex once - O(V)

Checking all the Edges - O(E)

Total = O(V+E)

Problem

You are given an integer n, which indicates that there are n courses labeled from 1 to n. You are also given prerequisite relationship between courses such that a course can be taken only when all its prerequisites are completed.

In one semester, you can take any number of courses as long as you have taken all the prerequisites in the previous semester for the courses you are taking.

Give an algorithm to find the minimum number of semesters needed to take all courses.

It is guaranteed that it is possible to take all the courses.

Solution

Create a graph with the vertices corresponding to courses and for every course u that is a prerequisite for course v there is an edge from u to v.

```
map = vertex -> its indegree
queue = all vertices with in-degree as 0
Total_sem, curr_courses = 0, queue.length()
While (queue is not empty):
        Dequeue to get a vertex u
        Decrement curr_courses by 1
        If curr_courses == 0:
                curr_courses = queue.length()
                Total_sem++
        For all outgoing edges (u,v):
                map[v] = map[v]-1 // decrease in-degree of the v by 1
                If map[v] == 0:
                        enqueue(queue, v)
return Total_sem
```

Question 1

We have a connected undirected graph $G=(V,E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

(Ch 3 Q 6 from textbook)

Solution

Proof by contradiction:
Assume there is an edge e = (u,v) in G that does not belong to T.
Since T is a DFS tree, one of u or v is the ancestor of the other. On the other hand, since T is a BFS Tree, u and v differ by 1 layer. Now since one of u or v is the ancestor of the other and they differ by 1 layer, therefore (u,v) should be in the BFS tree T. This contradicts the assumption. Therefore, G cannot contain any edges that do not belong to T

Question 2

Given positive non-decreasing functions f_1, f_2, g_1, g_2 such that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. Is it true that $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$? Is it true that $2^{f1(n)} = O(2^{g1(n)})$?

Solution:

True. $f_1(n) \cdot f_2(n) \leq c_1 \cdot g_1(n) \cdot c_2 \cdot g_2(n) = (c_1 \cdot c_2) \cdot (g_1(n) \cdot g_2(n))$

False. Does not work for $f_1(n) = 2n$ and $g_1(n) = n$

Question 3

Find the tight upper bound on the worst case run time of the following code segment.

```
while(n>0)
    find_max(L, n) // finds max in L[0…….n-1]
    n = n/4
```

Solution

The call to find_max takes linear time with respect to n

Adding up the cost to call find_max in each iteration:

1st call: cn

2nd call: cn/4

3rd call: cn/16

.

.

.

Summation of the series: cn + cn/4 + cn/16…..

= cn · (1 + ¼ + 1/16 + ….)

≤ 2cn because we know that (1 + ½ + ¼ ……) = 2

Therefore, the tight upper bound is θ(n)

Question 4

Find the tight **lower** bound on the best case run time of the following code segment.

```
string func(int n):
    if n == 0: return “a”
    string st = func(n-1)
    return st + st
```

Solution

Looking at the output:

n	Output
0	a
1	aa
2	aaaa
3	aaaaaaaa
…	
n	2^n

So, the tight lower bound is $\theta(2^n)$

Question 5

In a connected bipartite graph, is the bipartition unique? Justify your answer.

Solution:

This is True. We can prove this by contradiction. Let's say we have already found the bipartition X,Y where all edges in the graph go between X and Y. Now let's assume that we have another bipartition X', Y' different from X, Y. If X', Y' is different from X, Y, then there must be at least one node *i* that used to be in X and now is not in X' but has moved into Y'. We can run a BFS starting from node *i.* Say *i* is at level 1 in the BFS tree. All nodes at level 2 that must have belonged to Y should now be in X', and all nodes at level 3 which must have belonged to X should now be in Y', and so on. In other words, all node that used be in X are now in Y' and all nodes that were in Y are now in X'. So X', Y' is not different from X, Y.

Question 6

Arrange the following functions in increasing order of growth rate with g(n) following f(n) in your list if and only if f(n) = O(g(n))

$$\log n^n,\ n^2,\ n^{\log n},\ n \log \log n,\ 2^{\log n},\ \log^2 n,\ n^{\sqrt{2}}$$

Solution

First separate the functions into logarithmic, polynomial, and exponential

$$2^{\log n} = n,\ n^{\log n} = 2^{(\log n)^2},\ \log n^n = n \log n$$

Logarithmic: $\log^2 n$
Exponential: $n^{\log n}$
Polynomial: everything else

We know that logarithmic functions grow slower than polynomial functions and exponential grows faster than polynomial

We just need to order polynomial

$$n,\ n \log \log n,\ n \log n,\ n^{\sqrt{2}},\ n^2$$

Therefore

$$\log^2 n,\ n,\ n \log \log n,\ n \log n,\ n^{\sqrt{2}},\ n^2,\ n^{\log n}$$