COMP2230 Algorithms

Lecture 5

Professor Ljiljana Brankovic

Lecture Overview

- Binary Search
- Topological Sort
- Backtracking
- Games

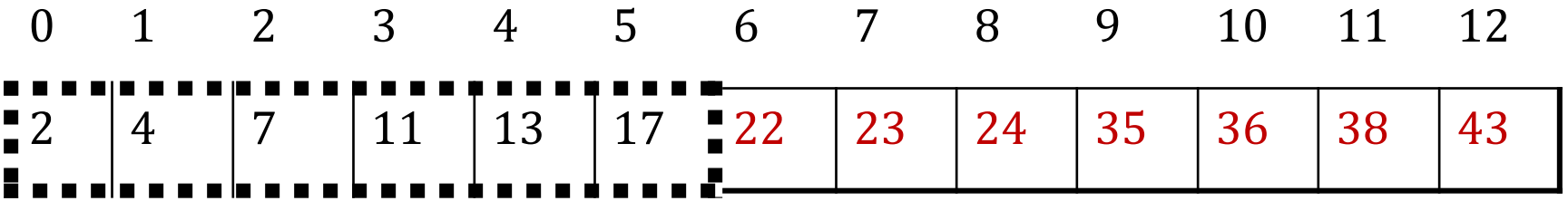
Based on the textbook and some slides by P. Moscato and Y. Lin.

Binary Search

Binary search is a very efficient algorithm for searching in a sorted array.

In fact, it is an optimal algorithm – no other algorithm that uses only comparisons is faster than binary search.

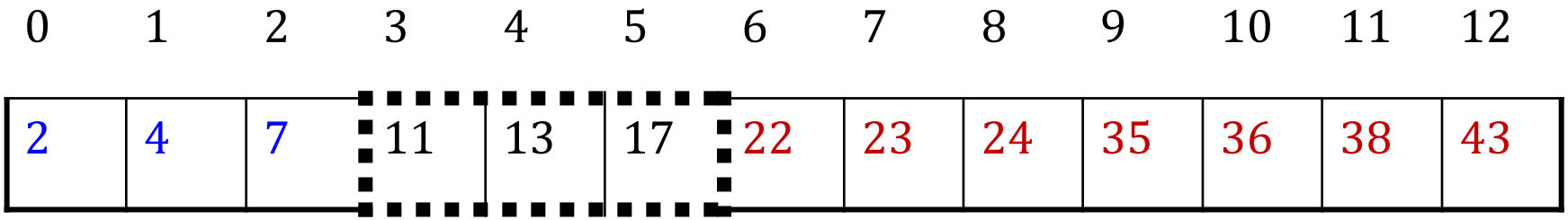
Example 1 - Binary Search



Find 11.

$$i = 0\,;\, j = 12\,;\qquad k = \frac{i+j}{2} = 6\,;\qquad 22 > 11 \;\rightarrow\; j = k - 1 = 5$$

Example 1 - Binary Search

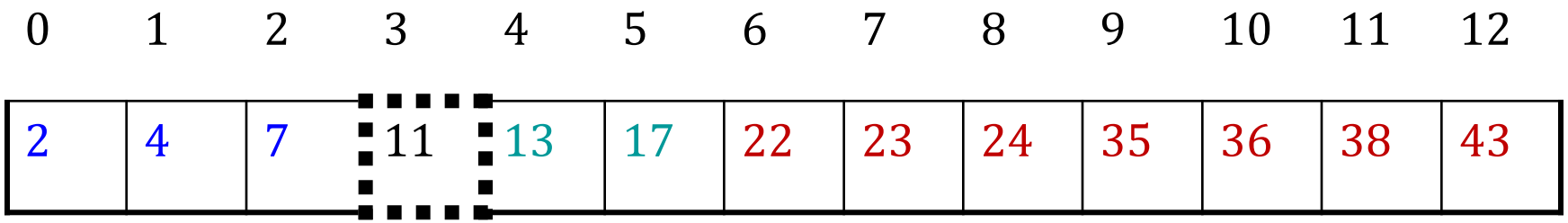


Find 11.

$i = 0;\ j = 12;\quad k = \frac{i+j}{2} = 6;\quad 22 > 11 \rightarrow\quad j = k - 1 = 5$

$i = 0;\ j = 5;\quad k = \frac{i+j}{2} = 2;\quad 7 < 11 \rightarrow\quad i = k + 1 = 3$

Example 1 - Binary Search



Find 11.

$i = 0;\ j = 12\ ;\quad k = \frac{i+j}{2} = 6\ ;\quad 22 > 11\ \rightarrow\quad j = k - 1 = 5$

$i = 0;\ j = \ 5\ ;\quad k = \frac{i+j}{2} = 2\ ;\quad 7 < 11\ \rightarrow\ i = k + 1 = 3$

$i = 3;\ j = \ 5\ ;\quad k = \frac{i+j}{2} = 4\ ;\quad 13 > 11\ \rightarrow\ j = k - 1 = 3$

Example 1 - Binary Search

0	1	2	3	4	5	6	7	8	9	10	11	12
2	4	7	11	13	17	22	23	24	35	36	38	43

Find 11.

$i = 0;\ j = 12;\quad k = \frac{i+j}{2} = 6;\quad 22 > 11 \rightarrow\quad j = k - 1 = 5$

$i = 0;\ j = 5;\quad k = \frac{i+j}{2} = 2;\quad 7 < 11 \rightarrow\quad i = k + 1 = 3$

$i = 3;\ j = 5;\quad k = \frac{i+j}{2} = 4;\quad 13 > 11 \rightarrow\quad j = k - 1 = 3$

$i = 3;\ j = 3;\quad k = \frac{i+j}{2} = 3;\quad 11 = 11$

Algorithm 4.1.1 Binary Search

This algorithm searches for the value key in the nondecreasing array *L[i],…,L[j]*. If key is found, the algorithm returns an index *k* such that *L[k]* equals key. If key is not found, the algorithm returns *-1*, which is assumed not to be a valid index.

```
Input Parameters: L,i,j,key
Output Parameters: None
bsearch(L,i,j,key) {
    while (i ≤ j) {
        k = (i + j)/2
        if (key == L[k]) // found
            return k
        if (key < L[k]) // search first part
            j = k - 1
        else // search second part
            i = k + 1
    }
    return -1 // not found
}
```

Correct or not?

```
bsearch(L,i,j,key) {
   while (i ≤ j) {
        k = (i + j)/2
        if (key == L[k])
                return k
        if (key < L[k])
                j = k
        else
                i = k
   }
   return -1
```

If correct, what is the worst-case time?

The algorithm is not correct. When the key is not in the array, the algorithm does not terminate.

Correct or not?

```
bsearch(L,i,j,key) {
   if (i>j)
         return -1
   k = (i + j)/2
   if (key == L[k])
        return k
   flag = bsearch(L,i,k-1,key)
   if (flag == -1 )
        return bsearch(L,k+1,j,key)
   else
        return flag
}
```

If correct, what is the worst-case time?

The algorithm is correct but it is not very efficient. It always searches through lower half of the array first and thus the worst-case time is $\Theta(n)$.

Correct or not?

```
bsearch(L,i,j,key) {
   if (i>j)
         return -1
   k = (i + j)/2
   if (key == L[k])
        return k
   if (key <L[k])
        return bsearch(L,i,k,key)
   else
        return bsearch(L,k+1,j,key)
}
```

If correct, what is the worst-case time?

The algorithm is not correct. It does not always terminate.

Binary Search: Complexity for successful search

Best-case : $C_{best}(n) = 1 = \Theta(1)$

Worst-case : $C_{worst}(n) = \lfloor \lg n \rfloor + 1 = \Theta(\lg n)$

Average-case : $C_{avg}(n) = \lfloor \lg n \rfloor = \Theta(\lg n)$

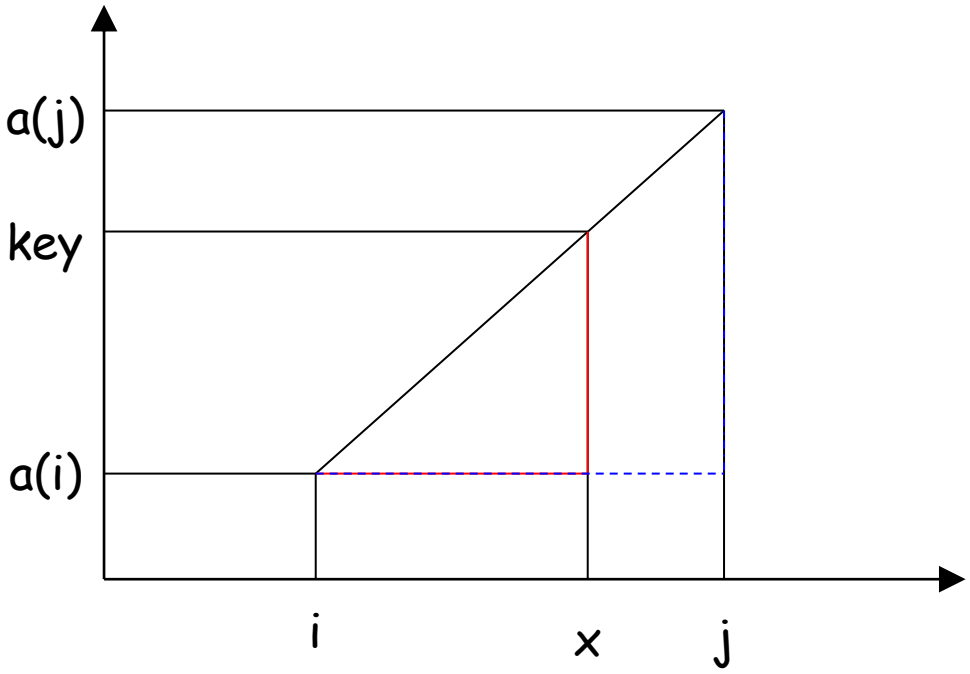
Prove the above!

Can we do better than binary search, that is, better than $\Theta(\lg n)$?

Interpolation search

- Binary search compares a search key with the middle value of the sorted array.
- Interpolation search takes into account the values of the search key and the smallest and largest element in the array, and estimates where the search key would be if it is in the array.
- Interpolation search assumes that values in the array increase linearly with the index.

Interpolation search



$$\frac{x-i}{key-a(i)} = \frac{j-i}{a(j)-a(i)}$$

$$x = i + \frac{(key-a(i))(j-i)}{a(j)-a(i)}$$

Interpolation search

The worst case complexity for interpolation search is $\Theta(n)$.

However, in an array with random keys, the average-case complexity is $\Theta(\lg \lg n)$.

Hashing

Hashing distributes the values of the array evenly among elements of the *hash* array.

This is done by computing a hash function.

For example, if the elements of the array are integers, the hash function can be

$$h(K) = K \bmod m.$$

The hash values will be integers between 0 and $m - 1$, inclusive.

Hashing

Let n be the size of the original array, and m the size of the hash array.

Whenever $m < n$ we will certainly have *collisions*, that is, two or more elements of the original array being hashed into the same cell of the hash array.

Hashing

One way to deal with collisions is to have a linked list for each cell of the hash array that gets more than one element of the original array. Then for a "good" hash function, that is, the function that distributes the values evenly among elements of the hash array, the average number of comparisons for a successful search will be $1 + \frac{n}{2m}$.

The drawback of hashing is the extra space required for the hash array.

Topological Sort

Example 2: Given a list of courses and prerequisites, give a topological sort, that is, a list of courses in order in which they can be taken to satisfy the prerequisites.

Topological Sort

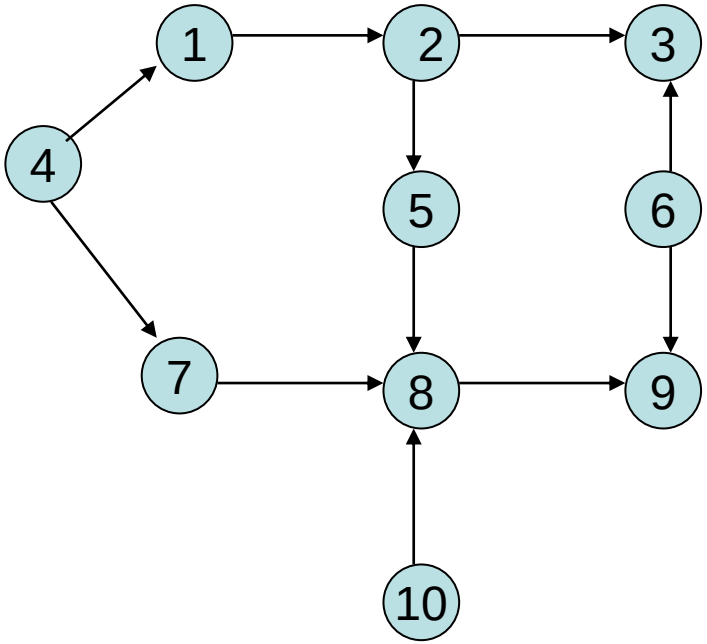
	Course	Prerequisite
1	MATH130	MATH120
2	MATH140	MATH130
3	MATH200	MATH140, PHYS130
4	MATH120	None
5	COMPSCI150	MATH140
6	PHYS130	None
7	COMPSCI100	MATH120
8	COMPSCI200	COMPSCI100, COMPSCI150, ENG110
9	COMPSCI240	COMPSCI200, PHYS130
10	ENG110	none

Topological Sort

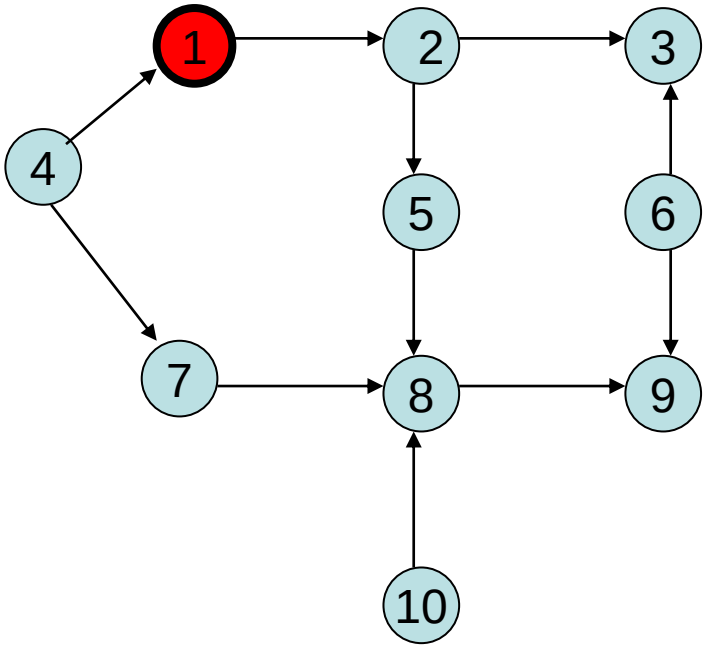
In order for topological sort to exist, there must not be a cycle in the prerequisites!

We can model this problem as a directed graph – it must not have any directed cycle – *directed acyclic graph* (dag)

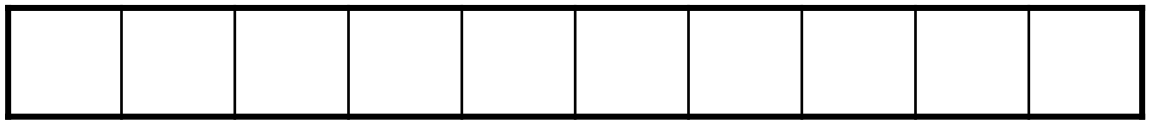
Topological Sort

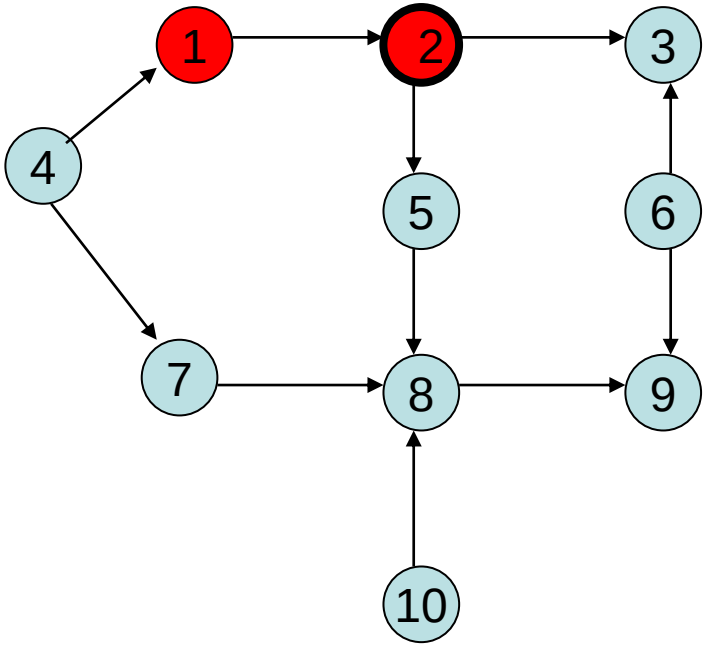


Topological Sort

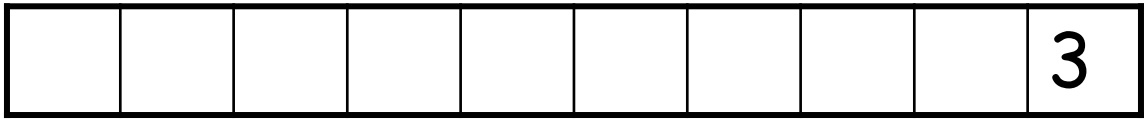


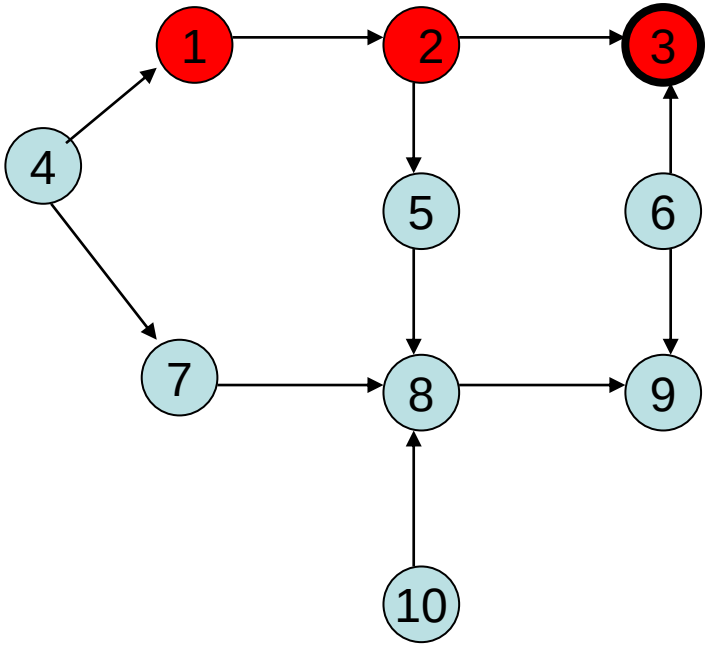
Topological Sort



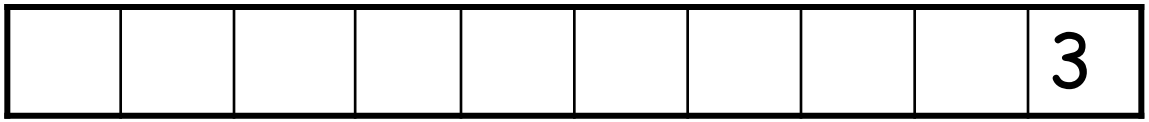


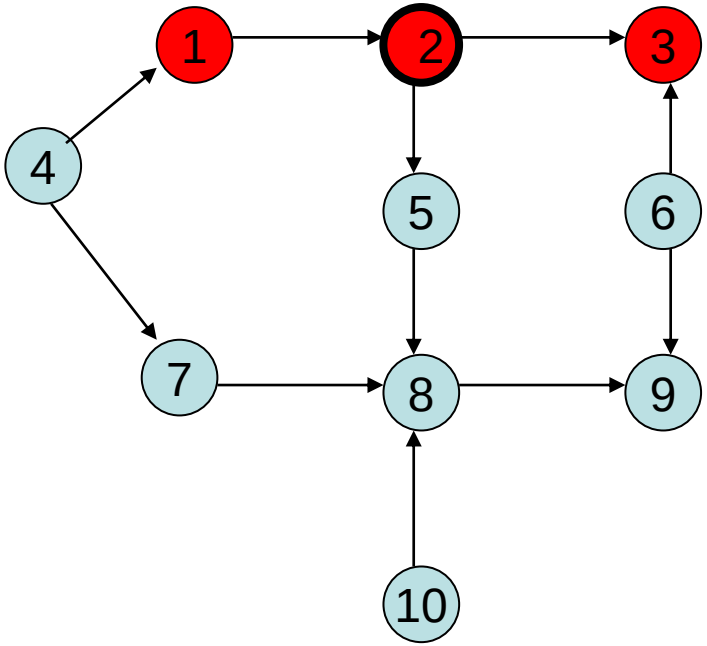
Topological Sort



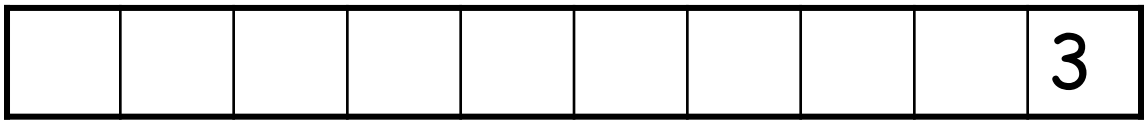


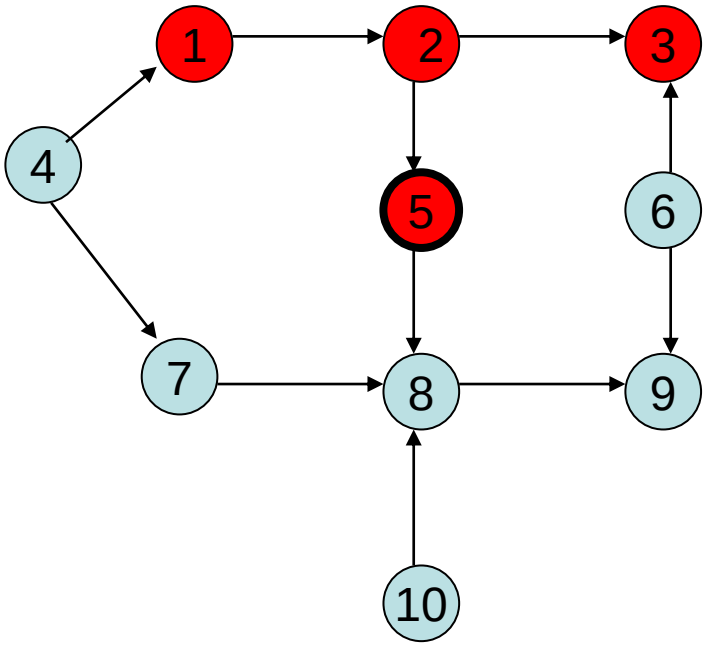
Topological Sort



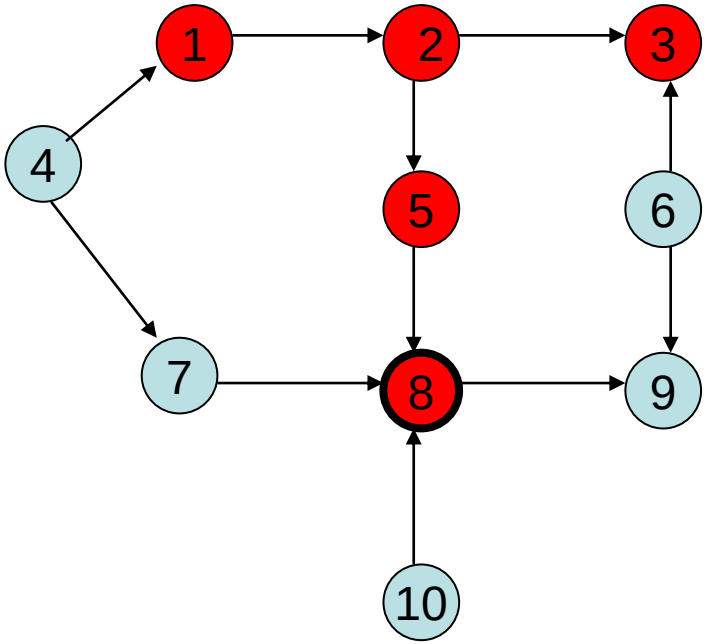


Topological Sort

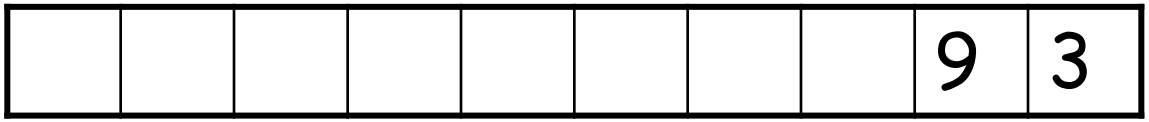


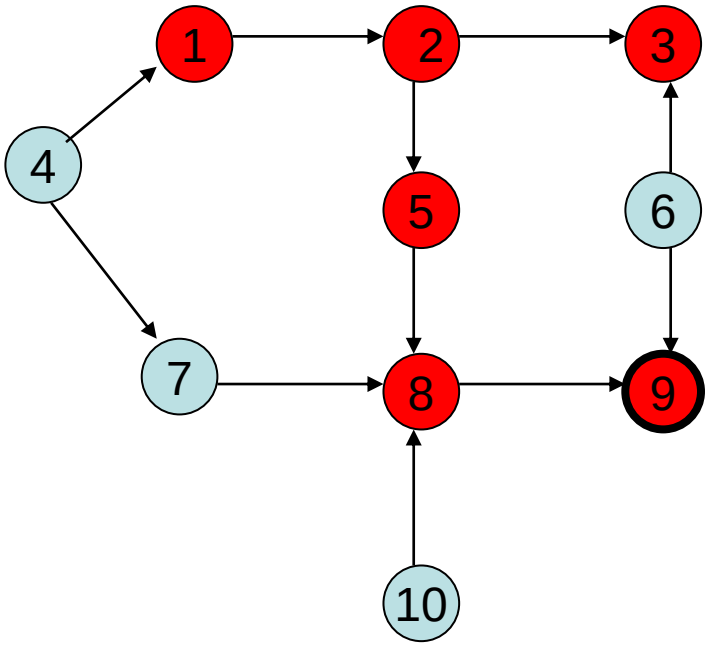


Topological Sort



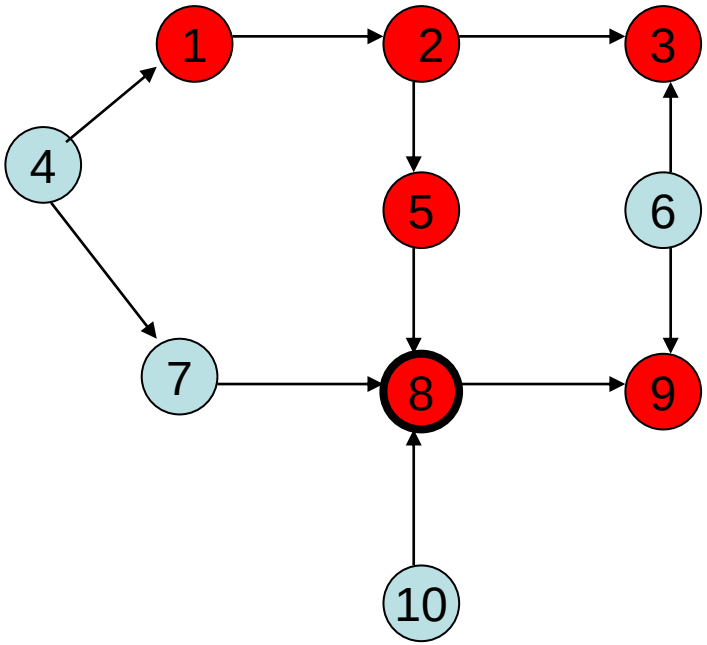
Topological Sort





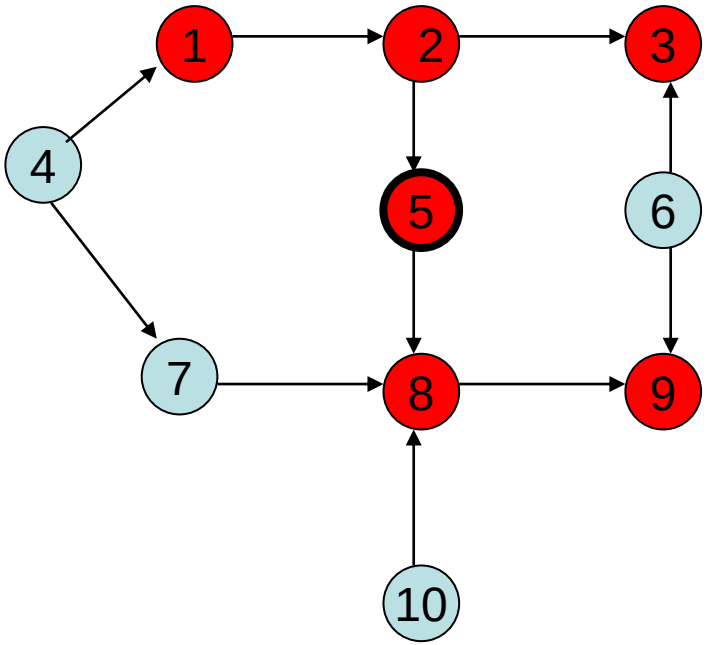
Topological Sort

							8	9	3



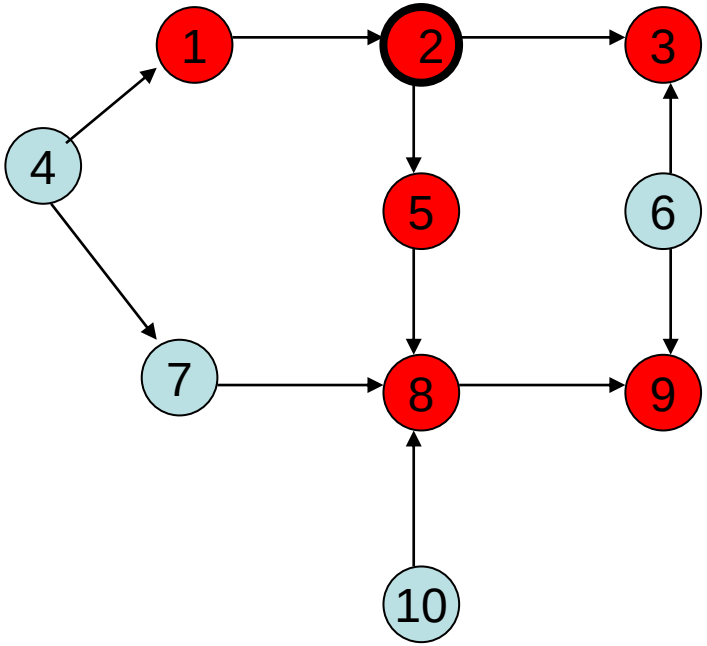
Topological Sort

						5	8	9	3



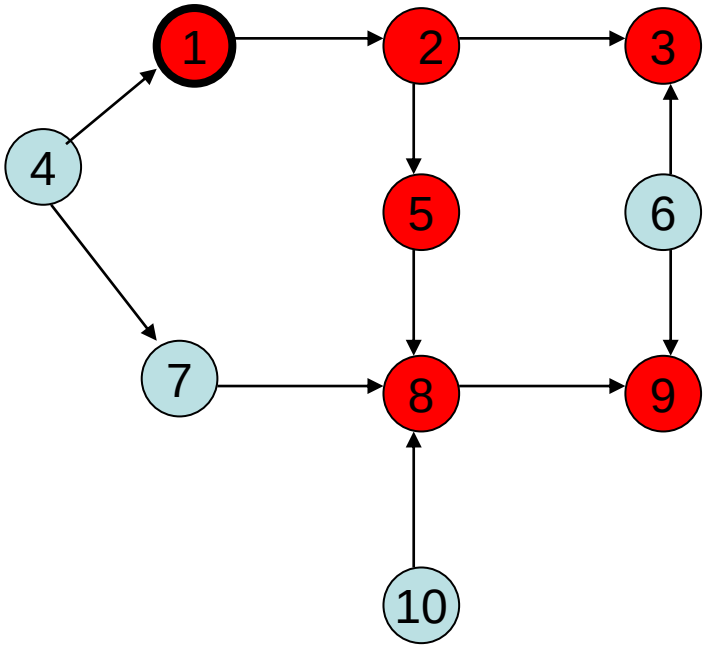
Topological Sort

					2	5	8	9	3



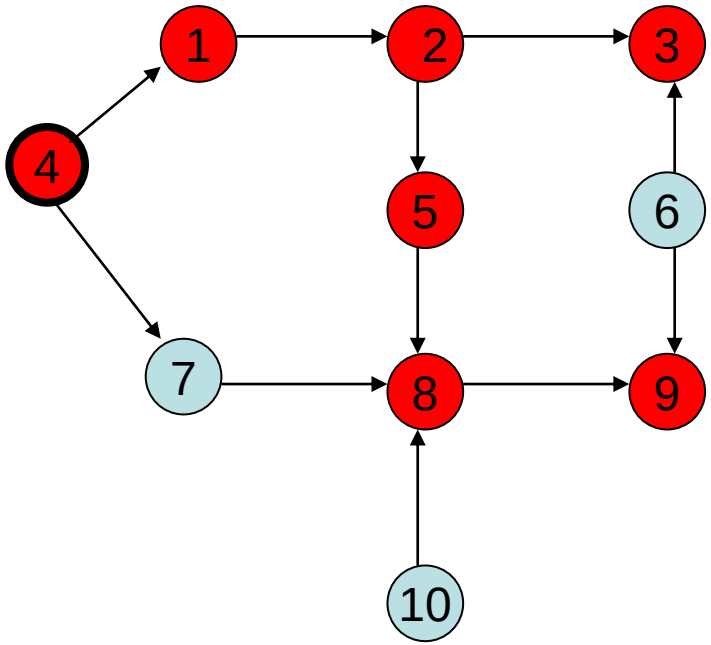
Topological Sort

				1	2	5	8	9	3



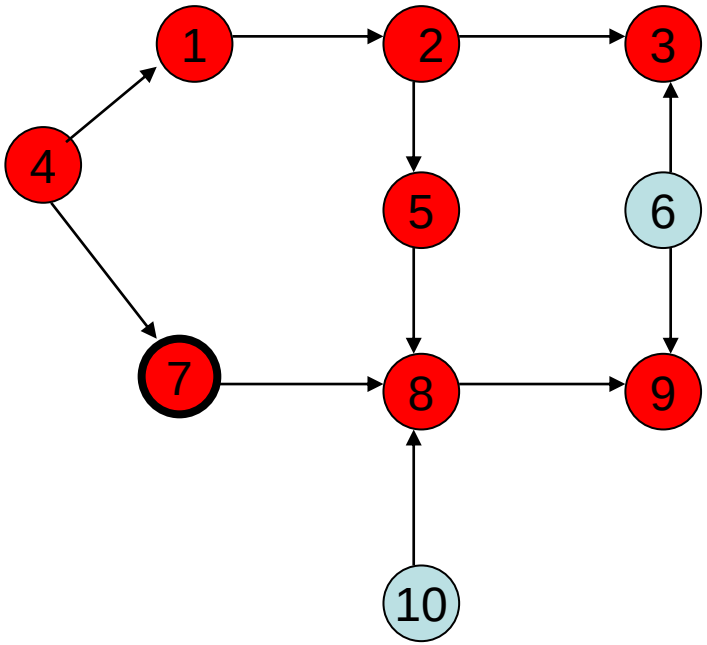
Topological Sort

				1	2	5	8	9	3



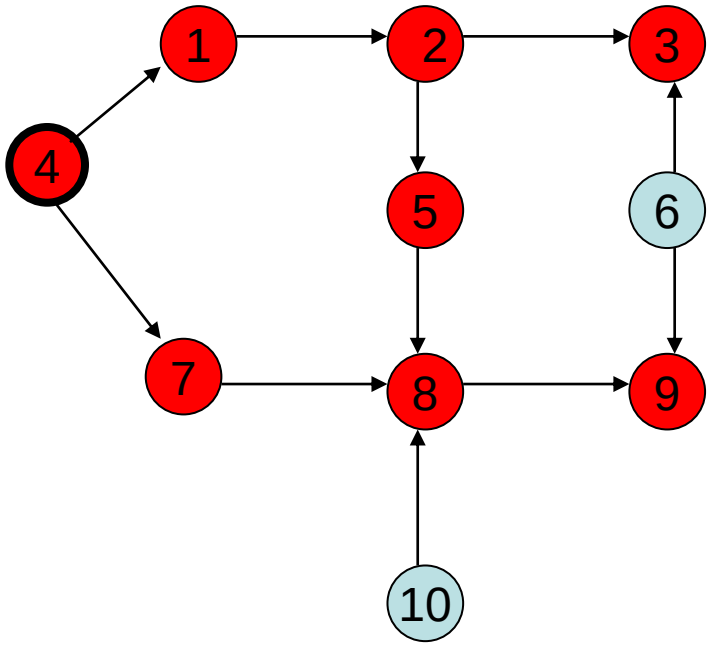
Topological Sort

			7	1	2	5	8	9	3



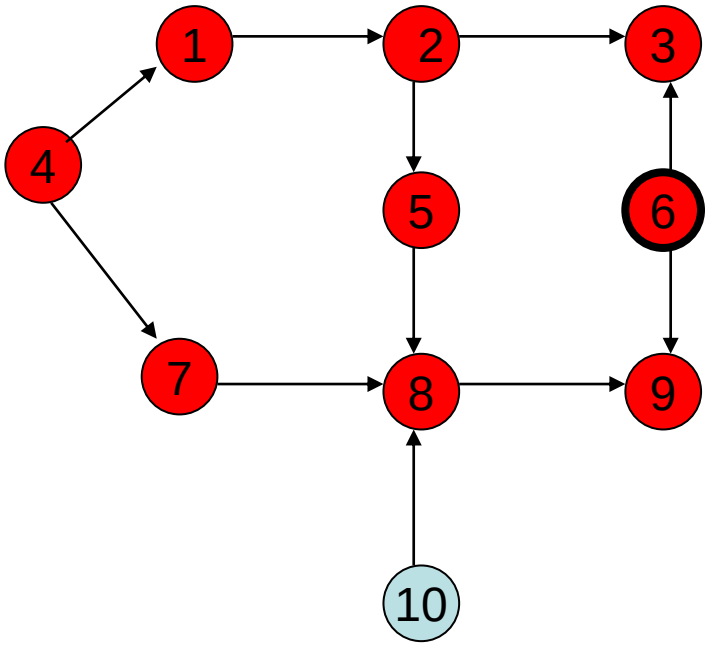
Topological Sort

		4	7	1	2	5	8	9	3



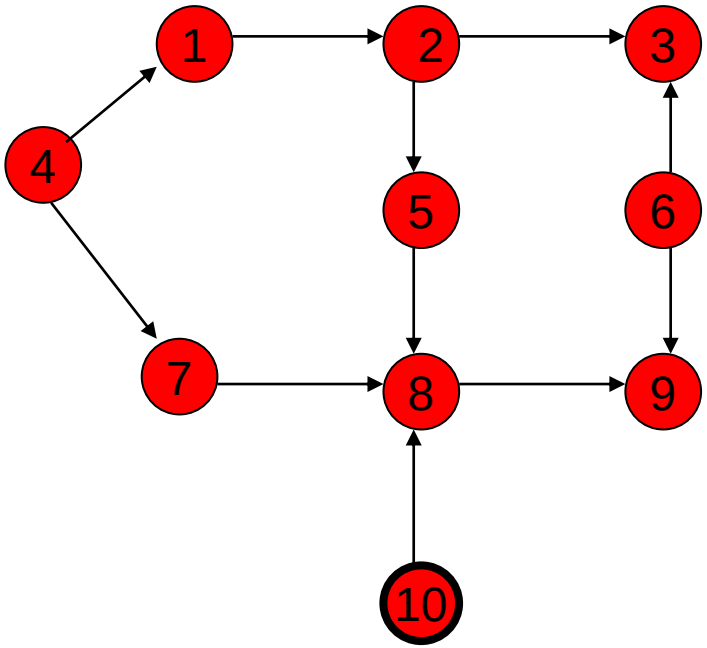
Topological Sort

	6	4	7	1	2	5	8	9	3



Topological Sort

10	6	4	7	1	2	5	8	9	3



Algorithm 4.4.1 Topological Sort

This algorithm computes a topological sort of a directed acyclic graph with vertices *1, … , n*. The vertices in the topological sort are stored in the array *ts*.

The graph is represented using adjacency lists; adj[*i*] is a reference to the first node in a linked list of nodes representing the vertices adjacent to vertex *i*. Each node has members *ver*, the vertex adjacent to *i*, and *next*, the next node in the linked list or null, for the last node in the linked list.

To track visited vertices, the algorithm uses an array *visit*; visit[*i*] is set to true if vertex *i* has been visited or to false if vertex *i* has not been visited.

```
Input Parameters: adj
Output Parameters: ts
top_sort(adj,ts)  {
    n = adj.last
    // k is the index in ts where the next vertex is to be
    // stored in topological sort. k is assumed to be global.
    k = n
    for i = 1 to n
        visit[i] = false
    for i = 1 to n
        if (!visit[i])
            top_sort_recurs(adj,i,ts)
}
top_sort_recurs(adj,start,ts) {
    visit[start] = true
    trav = adj[start]
    while (trav != null) {
        v = trav.ver
        if (!visit[v])
            top_sort_recurs(adj,v,ts)
        trav = trav.next
    }
    ts[k] = start
    k = k - 1
}
```

Complexity of Topological Sort

Suppose the graph has m edges and n vertices.

for loop in *top_sort*	$\Theta(n)$
top_sort_recurs	$\Theta(m)$
Total	$\Theta(m+n)$

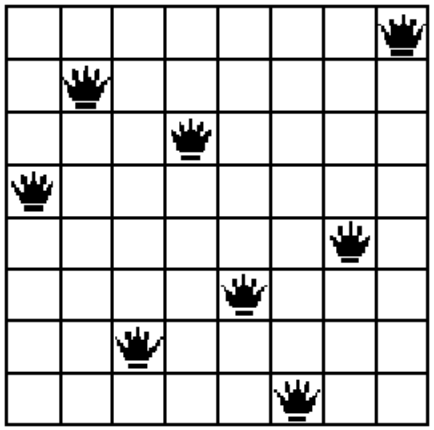
Backtracking

Backtracking

Backtracking is basically a depth-first search applied to a dynamically generated tree.

Example 3: The n-queens problem is to place n queens on an $n \times n$ board so that no two queens are in the same row, column, or diagonal.

8 x 8 board



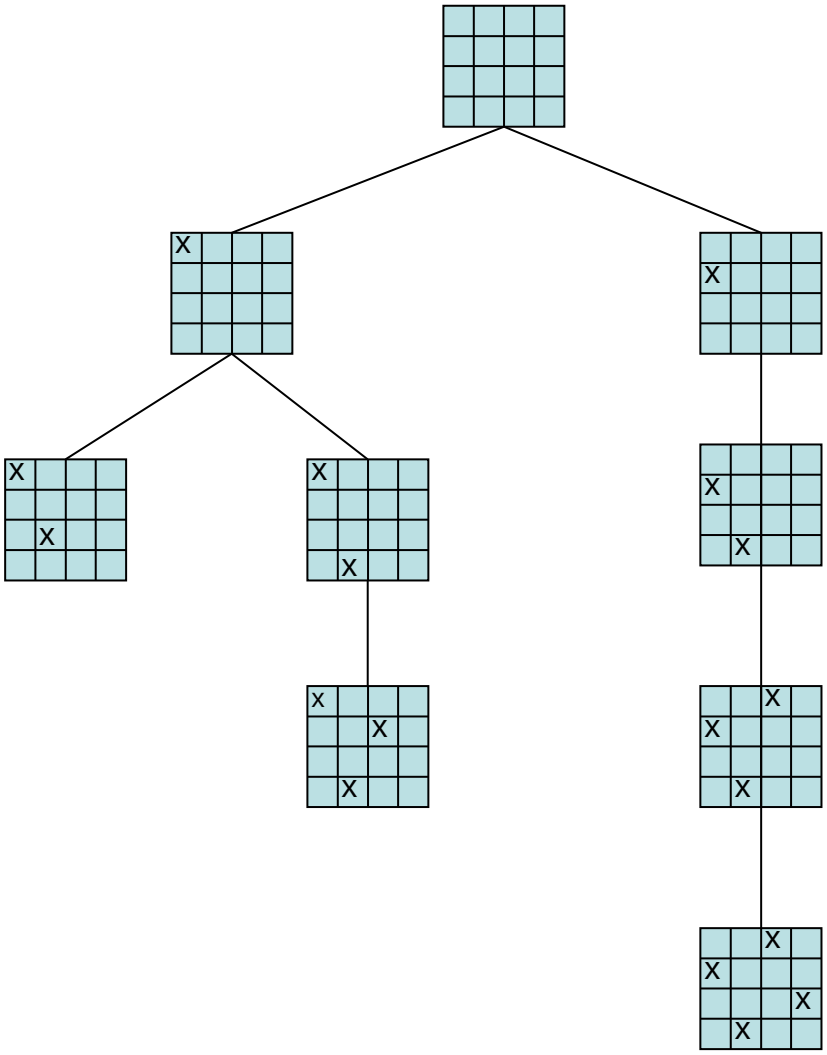
Algorithm n-Queens, Initial Version

Using backtracking, this algorithm outputs all solutions to this problem.

We place queens successively in the columns beginning in the left column and working from top to bottom.

When it is impossible to place a queen in a column, we return to the previous column and move its queen down.

Example 4



Algorithm n-Queens, Initial Version

The value of `row[k]` is the row where the queen in column `k` is placed.

The algorithm begins when `n_queens` calls `rn_queens(1,n)`. When `rn_queens(k,n)` is called, queens have been properly placed in columns `1` through `k - 1`, and `rn_queens(k,n)` tries to place a queen in column `k`.

If it is successful and `k` equals `n`, it prints a solution.

If it is successful and `k` does not equal `n`, it calls

`rn_queens(k + 1,n)`.

If it is not successful, it backtracks by returning to its caller

`rn_queens(k - 1,n)`.

```
n_queens(n) {
    rn_queens(1,n)
}

rn_queens(k,n) {
    for row[k] = 1 to n
        if (position_ok(k,n))
            if (k == n) {
            for i = 1 to n
                    print(row[i] + " ")
                println()
            }
            else
            rn_queens(k + 1,n)
}

position_ok(k,n)
    for i = 1 to k - 1
        // abs is absolute value
        if (row[k] == row[i] || abs(row[k] - row[i]) == k - i)
            return false
    return true
}
```

Algorithm 4.5.2 Solving the *n*-Queens Problem Using Backtracking

The value of `row_used[r]` is true if a queen occupies row `r` and false otherwise.

The value of `ddiag_used[d]` is true if a queen occupies *ddiag* diagonal `d` and false otherwise. According to the numbering system used, the queen in column `k`, row `r`, is in *ddiag* diagonal `n - k + r`.

The value of `udiag_used[d]` is true if a queen occupies *udiag* diagonal `d` and false otherwise. According to the numbering system used, the queen in column `k`, row `r`, is in *udiag* `k + r - 1`.

The function `position_ok(k,n)` assumes that queens have been placed in columns `1` through `k - 1`. It returns true if the queen in column `k` does not conflict with the queens in columns `1` through `k- 1` or false if it does conflict.

```
Input Parameter: n
Output Parameters: None
n_queens(n) {
    for i = 1 to n
        row_used[i] = false
    for i = 1 to 2 * n - 1
        ddiag_used[i] = udiag_used[i] = false
    rn_queens(1,n)
}

...
```

```
...
// When rn_queens(k,n) is called, queens have been
// properly placed in columns 1 through k - 1.
rn_queens(k,n) {
    for row[k] = 1 to n
        if (position_ok(k,n))
            row_used[row[k]] = true
            ddiag_used[n - k + row[k]] = true
            udiag_used[k + row[k] - 1] = true
            if (k == n) {
                // Output a solution. Stop if only one
                // solution is desired.
            for i = 1 to n
                    print(row[i] + " ")
                println()
            }
            else
            rn_queens(k + 1,n)
            row_used[row[k]] = false
            ddiag_used[n - k + row[k]] = false
            udiag_used[k + row[k] - 1] = false
}
...
```

```
...

// position_ok(k,n) returns true if the queen in column k
// does not conflict with the queens in columns 1
// through k - 1 or false if it does conflict.
position_ok(k,n)
    return !(row_used[row[k]]
        || ddiag_used[n - k + row[k]]
        || udiag_used[k + row[k] - 1 ])

}
```

Complexity of Backtracking for n-queens

Suppose we have $n \times n$ board.

- for loops in *n_queens* $\Theta(n)$
- *rn_queens*

$$n(1 + n + n(n-1) + n(n-1)(n-2) + \cdots + \mathrm{n}(\mathrm{n}-1)(\mathrm{n}-2) \ldots 2)$$
$$= \mathrm{n} \times n!\left(\frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}\right) = \mathrm{O}(n \times n!)$$

Total $\mathrm{O}(n \times n!)$

Graphs and Games

A special game: **NIM**

- Two players, *A* and *B*

- Pile of n matchsticks on the table

- Player A starts, taking k matchsticks, where $1 \leq k < n$
- Player on next turn may take j matchsticks, where $j \leq n - k$ and $1 \leq j \leq 2k$, and so on

- Player to take last matchstick wins

Model *NIM* as a Directed Graph?

- We may have a lot of options
- We want to model the whole game as a graph:
  - What are the nodes?
  - What are edges?
  - Why directed?

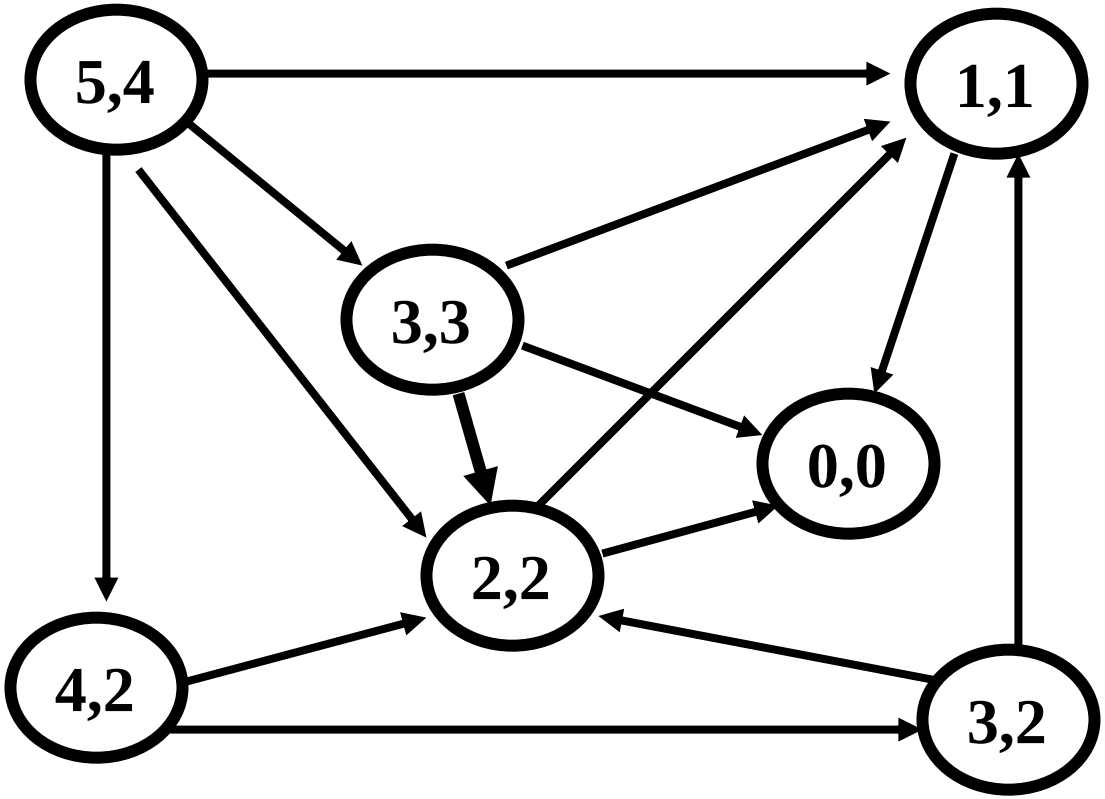
Modelling (cont)

- We may have a lot of moves
- Want to model the whole game (over time) as a directed graph:
  - What are the nodes?
  - What are edges?
  - Why directed?

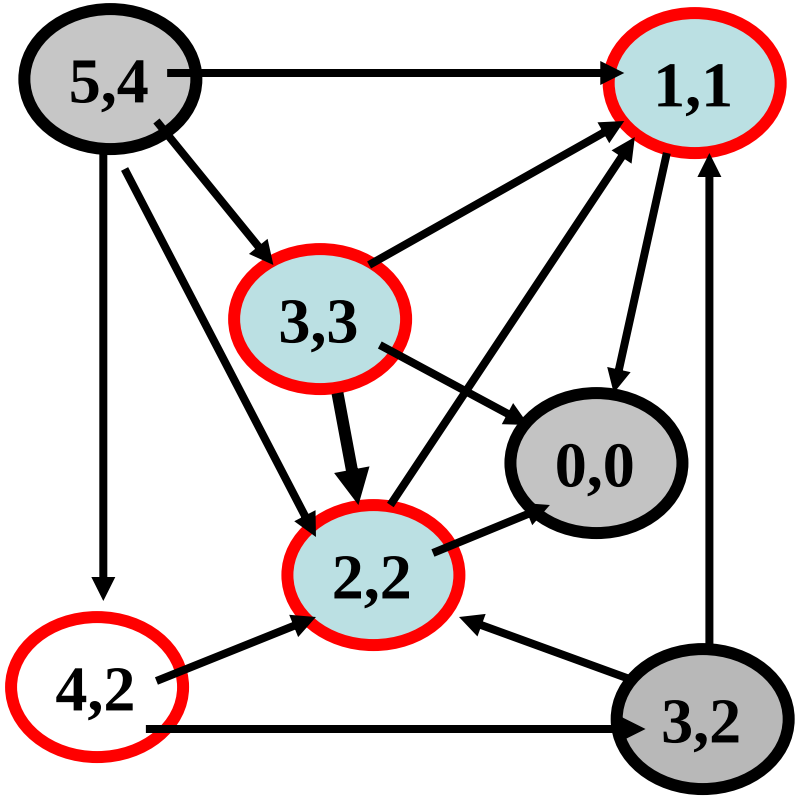
- Each node represents current state of the game.
  - ordered pair (i, j)
  - i = the number of sticks on table
  - j = maximum we can remove
- Each valid move is a directed edge
- Start: $(n, n-1)$
- Node $(0, 0)$ is a losing position
  - Why?

Model (cont)

- Graph representing the game of *NIM*



Winning and Losing



- Node $(0,0)$ is a losing position
- Nodes $(1,1)$, $(2,2)$ and $(3,3)$ are all winning positions because the player can choose to send opponent to $(0,0)$
- Node $(3,2)$ is losing because player has to send opponent to a winning position

Winning and Losing (general)

Winning position

- a move exists, which puts your opponent in a losing position
- A winning node may have many edges to other winning nodes
- Just needs one edge to a losing node.

Losing position

- Each move (if any) puts your opponent in a winning position

Is every position either winning or losing in *NIM*?

Determining a winning position

- Imagine, you want to write the world's best *NIM* playing computer program....
- Need an algorithm to determine whether a given node (i, j) is winning.
  - Usually the starting position
- All you have is the node (i, j) and the rules.

More general games

- two players, alternating turns
- symmetric: same rules for each
- deterministic: no chance involved (eg. dice)
- cannot last forever... what does that say about graph?
- No position allows infinite number of legal moves
- Terminal positions - positions allowing no legal moves

- Graph:
  - nodes = game states
  - directed edges = legal moves
  - Labels: win, lose, draw can be assigned to nodes

Labelling strategy

1. Label any terminal positions
   - often, if you can't move you lose
   - sometimes if you can't move it's a draw (eg. chess)

2. A non-terminal is winning if at least one edge leads to a losing position

3. A non-terminal is losing if all edges lead to winning positions

4. Any other non-terminal is a draw.
   - One of it's successors must also be a draw. Why?

Big Games

- Labeling always works.... in principle.

- Consider chess
  - state information is very large
  - Graph may be ***extremely*** big
  - May only want to explore a small section of the graph
  - May not want to store whole graph