CoE 163

Computing Architectures and Algorithms

02a: Asymptotic Analysis

MAXIMIZING ALGORITHMS

When formulating algorithms, “efficiency” and “maximization” are the next important things after “correctness”

- Time (set-up, run, save)
- space (memory, storage)

TIME EFFICIENCY

We can measure runtime of an algorithm in terms of ”basic computer steps”

- Each computer is different
- Platform-independent measure
- Expressed as a function of the size of the input n

CONSIDER...

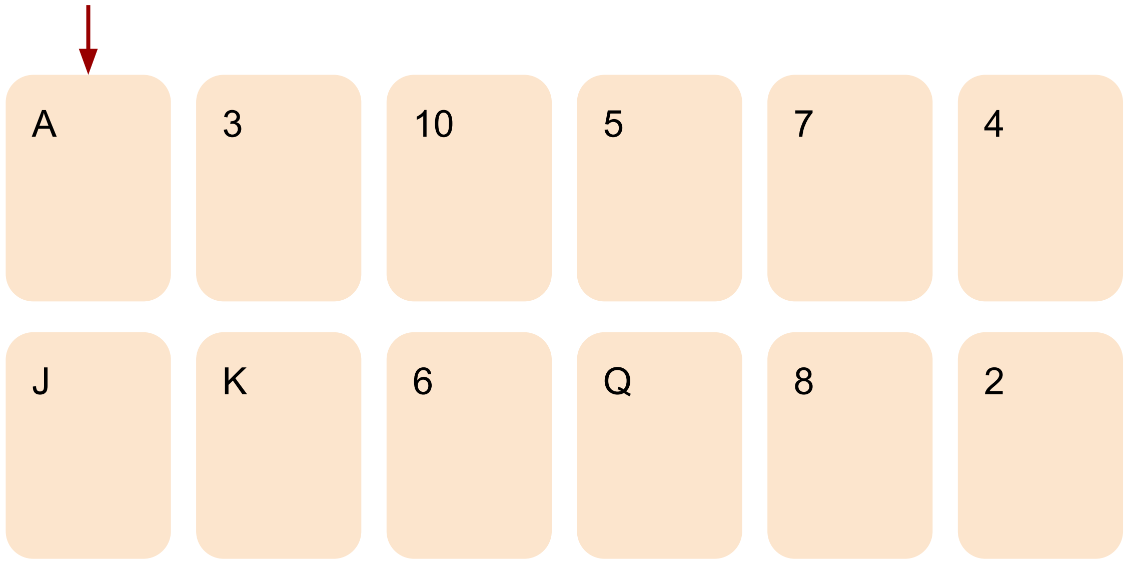
You are playing Tong-its and have a starting hand of 12 cards that you would like to sort by value.

SORTING BY HAND

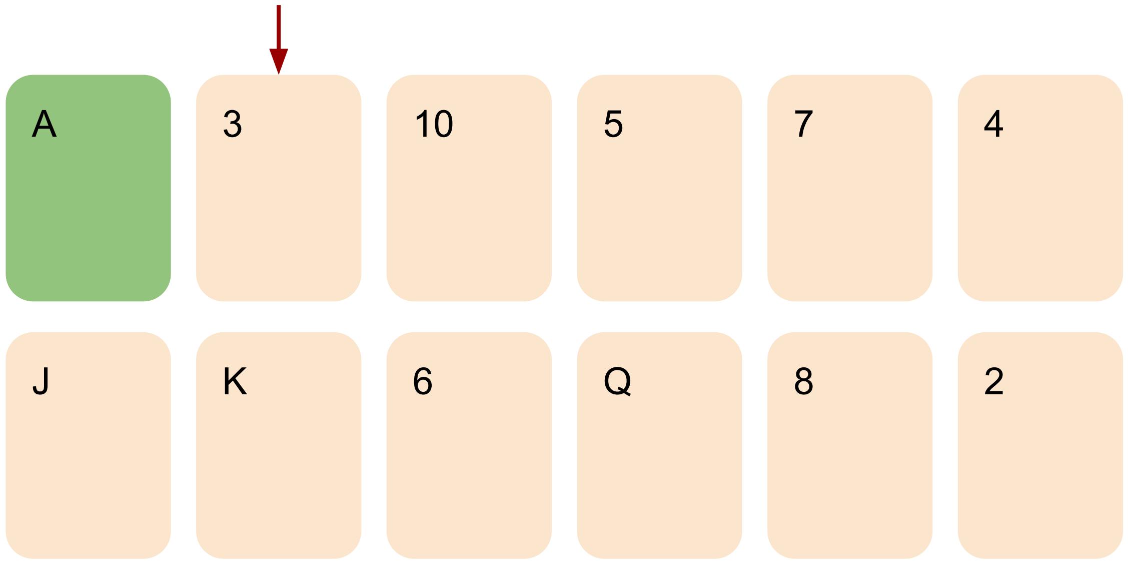
Consider a usual method where people scan all their cards, get the leftmost unsorted card, and insert it in the appropriate place on the sorted list.

Do this repetitively while tracking the boundary of the sorted and unsorted cards.

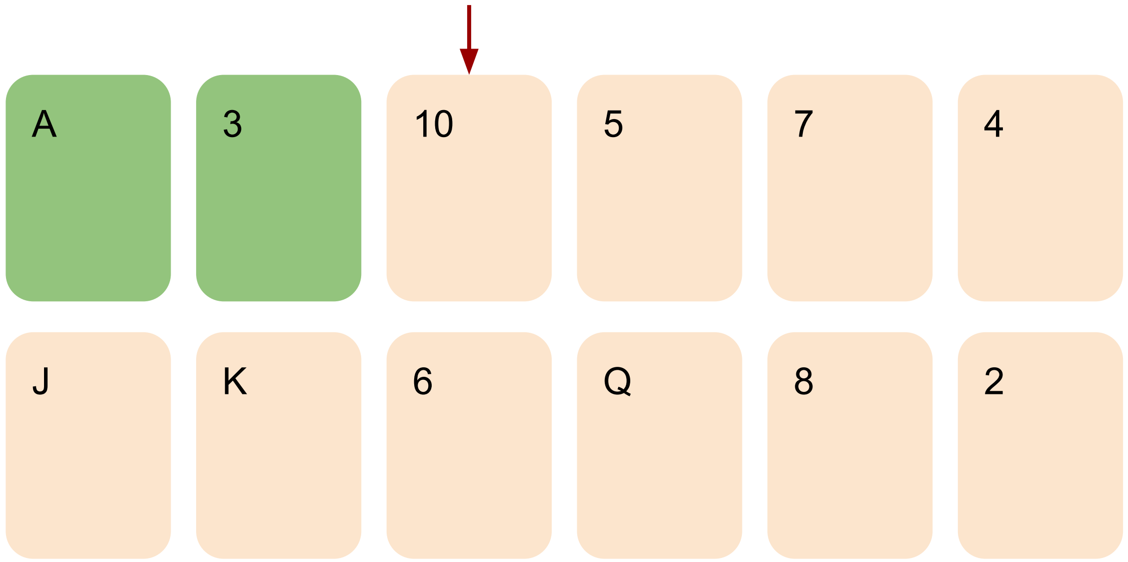
SORTING BY HAND



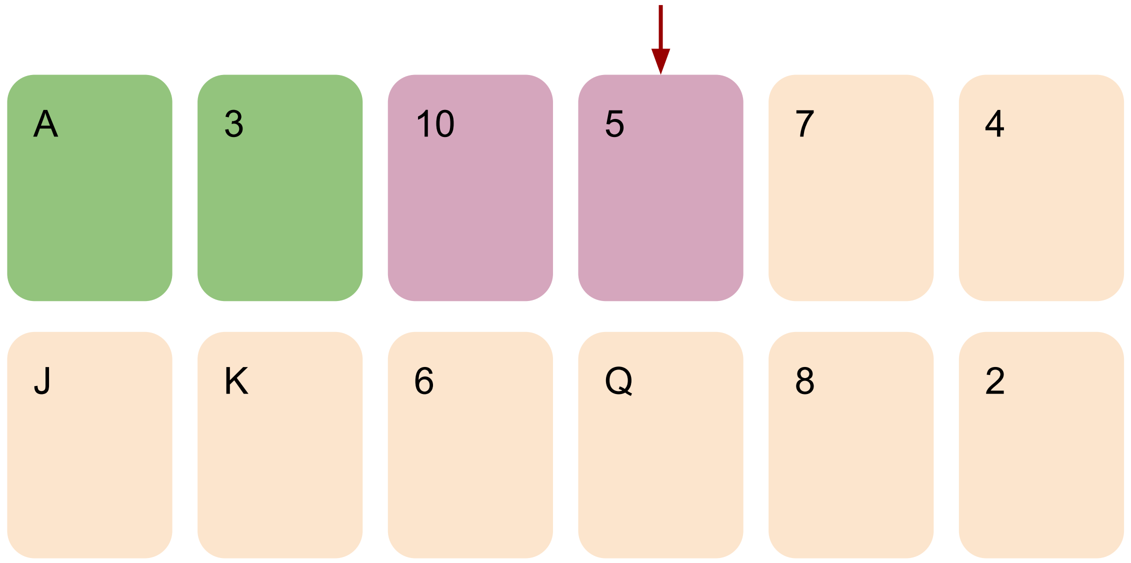
SORTING BY HAND



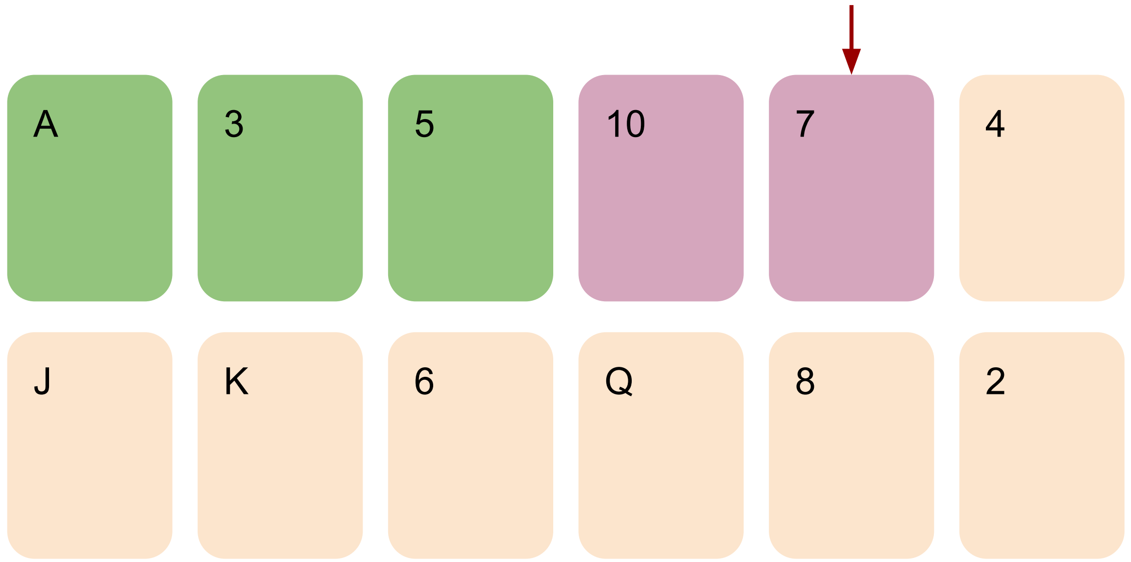
SORTING BY HAND



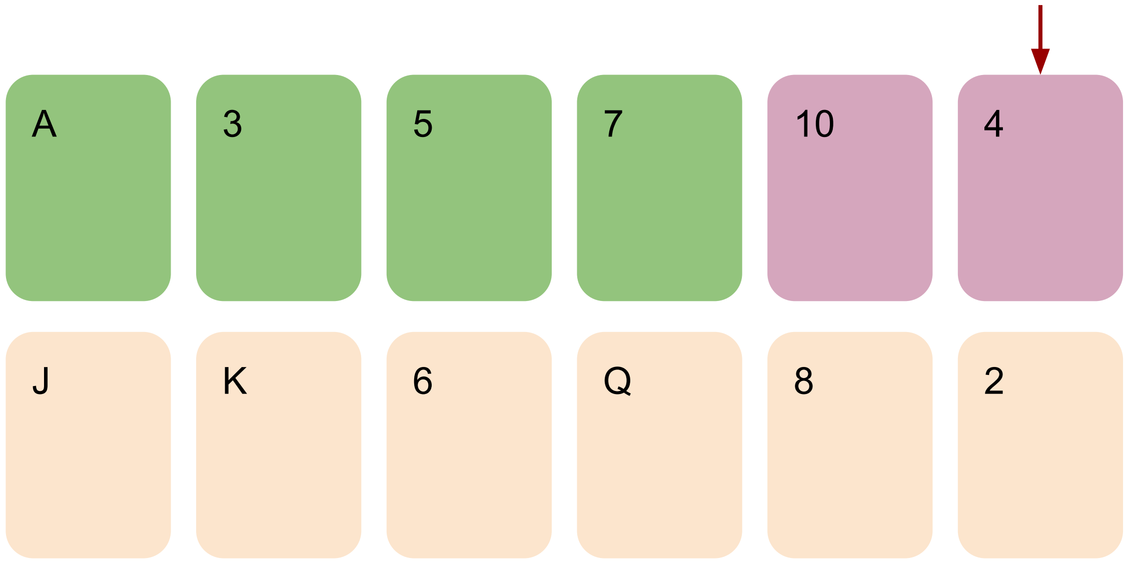
SORTING BY HAND



SORTING BY HAND



SORTING BY HAND



SOLVING THE PROBLEM

- How do we solve this problem as humans?
- How do we translate our solution into computer code?
- How do we decompose this problem if it is too big?
- How do we make it fast enough for our purposes?

PROBLEM DECOMPOSITION

- Save the 12 cards in an array
- Implicitly divide the array into two parts - sorted and unsorted
- When a lowest-valued element is picked from the unsorted set, put it at the end of the sorted set by swapping adjacent elements

PYTHON CODE

```
import random

hand_cards = [random.randint(1, 13) for x in range(12)]

# Unsorted hand_cards
print(hand_cards)

for i in range(len(hand_cards)):
     j = i
     while j > 0 and hand_cards[j] < hand_cards[j - 1]:
          hand_cards[j], hand_cards[j - 1] = \
               hand_cards[j - 1], hand_cards[j]

          j = j - 1

# Sorted hand_cards
print(hand_cards)
```

ALGORITHM RUNTIME

How do we measure the runtime of this algorithm in terms of “basic computer steps”?

- Assume certain operations cost one step
- Think in terms of data input size n
- Loop counting and guessing of loop invariants are key skills

ALGORITHM RUNTIME

```
import random

hand_cards = [random.randint(1, 13) for x in range(12)]

# Unsorted hand_cards
print(hand_cards)

for i in range(len(hand_cards)):
     j = i
     while j > 0 and hand_cards[j] < hand_cards[j - 1]:
          hand_cards[j], hand_cards[j - 1] = \
               hand_cards[j - 1], hand_cards[j]

          j = j - 1

# Sorted hand_cards
print(hand_cards)
```

Pseudocode of interest

ALGORITHM RUNTIME

- Let n = len(hand_cards)
- Assume swap, assignment, and math operations cost 1
- Loop analysis
    - Outer loop iterates n times
    - Inner loop iterates n - j times at the worse

```
for i in range(len(hand_cards)):
    j = i
    while j > 0 and hand_cards[j] < hand_cards[j - 1]:
        hand_cards[j], hand_cards[j - 1] = \
            hand_cards[j - 1], hand_cards[j]

        j = j - 1
```

ALGORITHM RUNTIME

If T(n) is the time it takes to run the algorithm, then,

$$T(n) = n * (n - j)$$

What if we can simplify this further, as j is variable?

BIG-O NOTATION

In the runtime equation, we can leave out the lower-order terms and the coefficients

$$T(n) = 5n^2 + n - 1$$

$$T(n) = \cancel{5}n^2 + \cancel{n} - \cancel{1}$$

We can conveniently express this as

$$T(n) = O(n^2)$$

BIG-O NOTATION: FORMAL

$$\exists f(n), g(n) : \mathbf{Z} \mapsto \mathbf{R}$$

$$f(n) = O(g(n))$$
$$\iff$$
$$f(n) \leq cg(n), c \in \mathbf{R} > 0$$

Interpretation: f(n) grows no faster than g(n), or “f(n) ≤ g(n)”

BIG-O NOTATION

$$T(n) = n * (n - j)$$

… can then be simplified in terms of the Big-O notation to …

$$T(n) = O(n^2)$$

This is the worst-case time complexity of our card-sorting algorithm.

OTHER “O” NOTATIONS

The opposite of the Big-O is the Big-Omega notation, where “g(n) ≥ f(n)”.

$$f(n) = O(g(n)) \equiv g(n) = \Omega(f(n))$$

If you want to note that “f(n) = g(n)”, then we use the Big-Theta notation.

$$f(n) = \Theta(g(n)) \equiv f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

The whole family of these are formally called Bachmann-Landau notations

BIG-O NOTATION: RULES

- Remove multiplicative constants
- Remove lower-order terms in a polynomial equation
- Higher exponent values grow faster
- Exponentials grow faster than polynomials
- Polynomials grow faster than logarithms

BIG-O NOTATION: RULES

- Many computer operations can be assumed to be of constant time (or O(1))
  - Math, relational, logical operations
  - Assignment
  - Printing/scanning strings
- Loops, recursions, and conditionals in loops need to be investigated

BIG-O NOTATION: EXAMPLES

Linearithmic:

$$O(n\ lg(n)) = 3n\ lg(7n) + lg(29n^3)$$

Polynomial:

$$O(n^2) = 2n^2 + 3n$$

Exponential:

$$O(2^n) = 5 * 2^n + n^{100}$$

Factorial:

$$O(n!) = n! + nlg(n) + 1$$

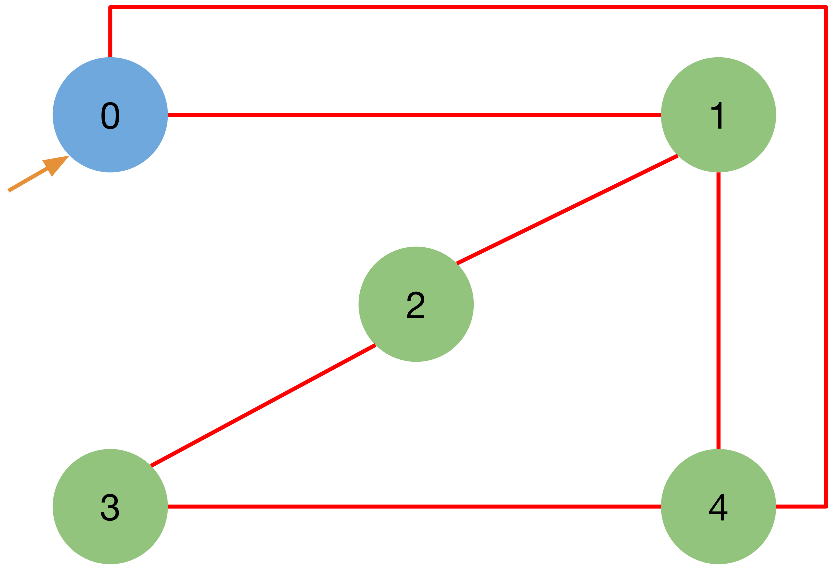
BIG-O NOTATION: WHY?

- The notation provides us with a platform-independent measure of runtime
- Runtime among different algorithms are easy to compare

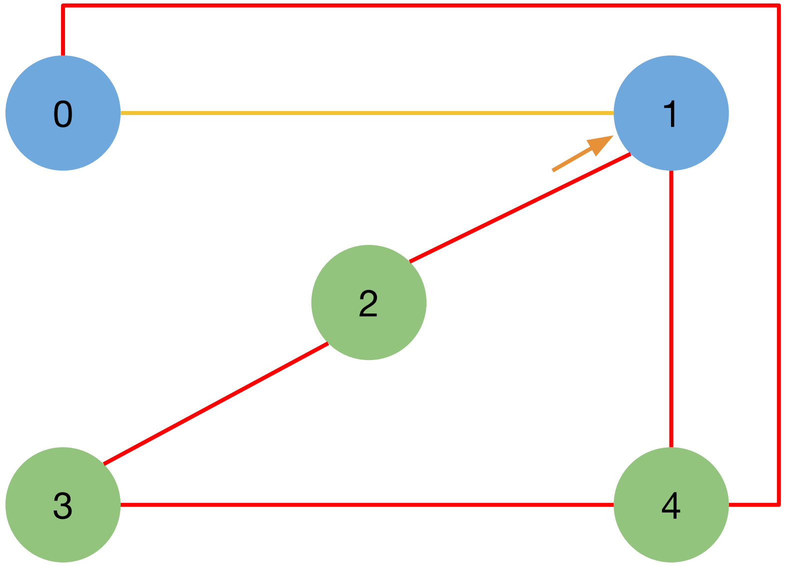
CONSIDER…

… finding the shortest path from an origin to destination using depth-first search (DFS).

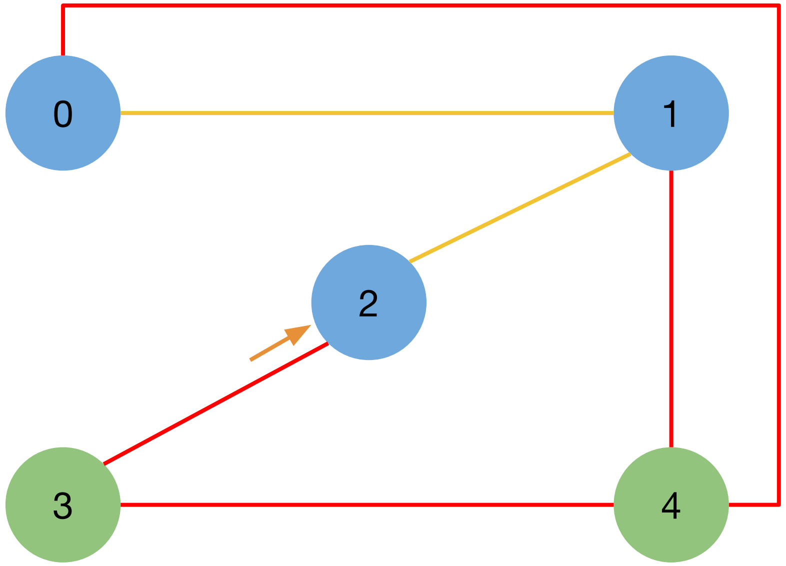
GRAPH TRAVERSAL



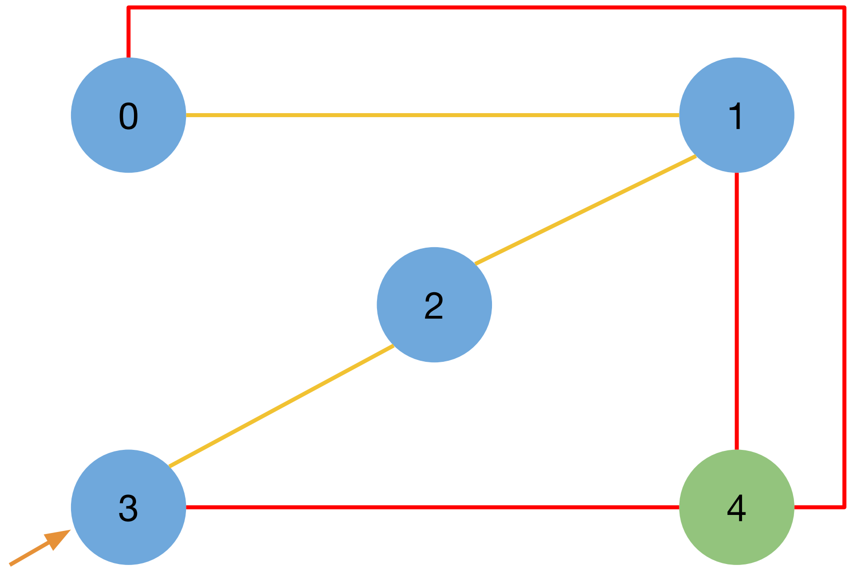
GRAPH TRAVERSAL



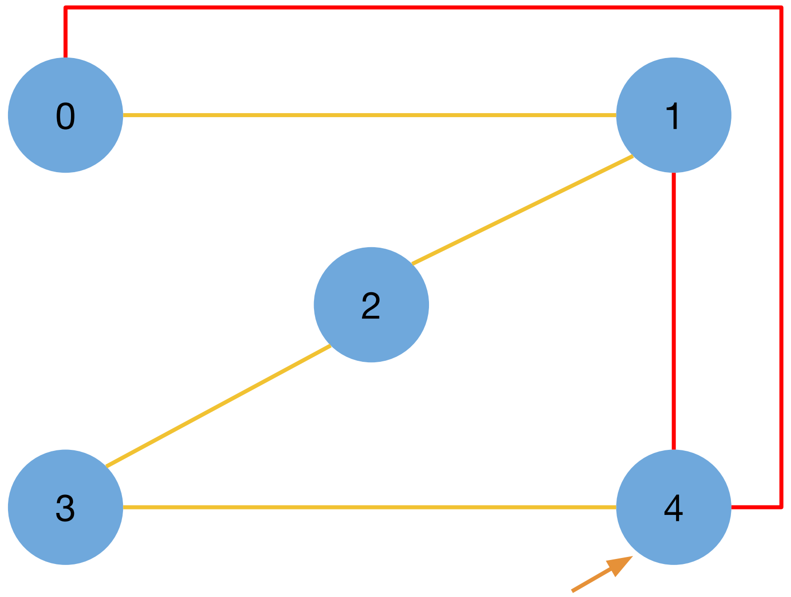
GRAPH TRAVERSAL



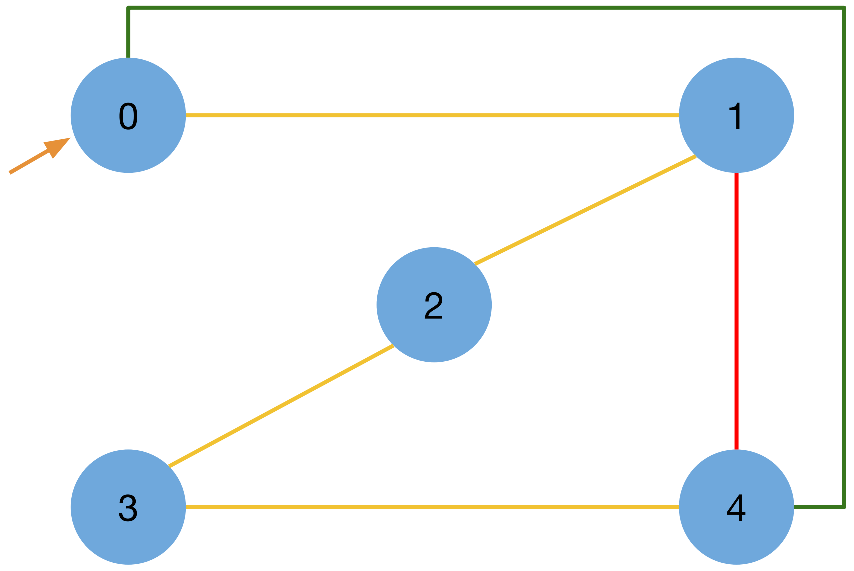
GRAPH TRAVERSAL



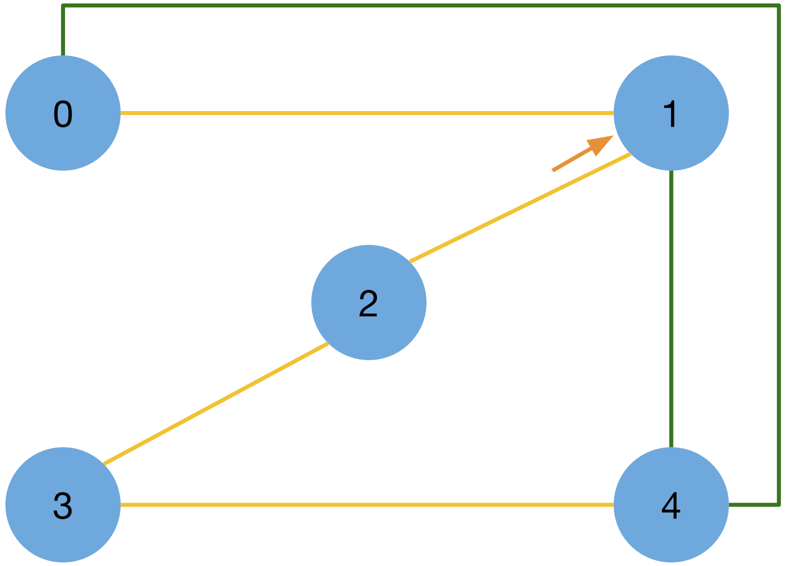
GRAPH TRAVERSAL



GRAPH TRAVERSAL



GRAPH TRAVERSAL



PYTHON CODE

```
# Assume 5 nodes
road_net = {0: [1, 4], 1: [0, 2], 2: [1, 3], 3: [2, 4], 4:
[0, 1, 3]}
visit_dist = [-1 for x in range(len(road_net))]

def dfs(this_dist, current_node):
     if visit_dist[current_node] >= 0:
          return

     print(current_node)
     visit_dist[current_node] = this_dist

     for each_nbr in road_net[current_node]:
          dfs(this_dist + 1, each_nbr)

start_node = 0
dfs(0, start_node)
```

TIME COMPLEXITY

```
# Assume 5 nodes
road_net = {0: [1, 4], 1: [0, 2], 2: [1, 3], 3: [2, 4], 4:
[0, 1, 3]}
visit_dist = [-1 for x in range(len(road_net))]

def dfs(this_dist, current_node):
     if visit_dist[current_node] >= 0:
          return

     print(current_node)
     visit_dist[current_node] = this_dist

     for each_nbr in road_net[current_node]:
          dfs(this_dist + 1, each_nbr)

start_node = 0
dfs(0, start_node)
```

Pseudocode of interest

TIME COMPLEXITY

- Let n be the number of nodes
- Recursion analysis
  - A node “visits” at most n_neighbor other nodes
    - May visit more than once
  - Every neighbor connection (edge) is checked

```
def dfs(this_dist, current_node):
    if visit_dist[current_node] >= 0:
        return

    print(current_node)
    visit_dist[current_node] = this_dist

    for each_nbr in road_net[current_node]:
        dfs(this_dist + 1, each_nbr)
```

goes through if just visited

loop n_{nbr} times

TIME COMPLEXITY

From the observations, the worst-case time complexity of DFS is

$$T(n) = O(|V| + |E|)$$

where |V| is the number of nodes and |E| the number of edges.

TIPS

- Make sure your algorithm is correct
  - Review EEE 121
  - Practice math proof techniques
  - Use Floyd-Hoare logic
- Practice keeping track of loops and recursions

RESOURCES

- Algorithms by Dasgupta, Papadimitriou, and Vazirani
- Your EEE 121 resources

CoE 163

Computing Architectures and Algorithms

02a: Asymptotic Analysis