4/13/21

Introduction

Algorithms are the ideas behind computer programs.

To be interesting, an algorithm has to solve a general, specified problem.

Must be correct, ideally efficient.

To describe algorithms,

Algorithms can be proven correct only through induction, or otherwise proven incorrect through contradiction (an example input where it fails).

4/14/21

Asymptotic Notation

Problem of the day the Knapsack problem:

Given a set of integers S and given a target number T, find a subset of S that adds up to exactly T.

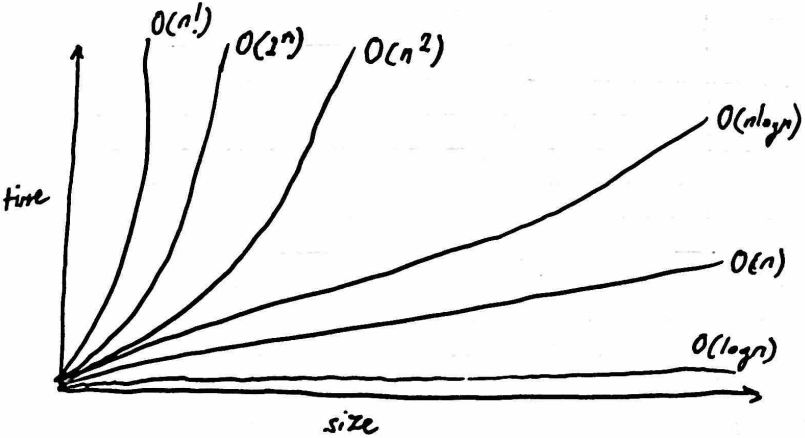
4/15/21

Program Analysis

Estimating the running time of an algorithm is usually easy given a precise enough description of the algorithm.

We care about the growth-coefficient of the complexity function. Constants that don't change based on the size of 'n' are safe to ignore.

Rule of thumb - nested loop = $O(n^2)$



Asymptotic dominance rankings:

$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < n!$

Logarithms? Simply an inverse exponential function.

Saying $b^x = y$ is the same as:

$x = \log_b y$

Logarithms reflect how many times we can double something until we can get to n.

Or, how many times to halve some n to get to 1.

eg In binary search we throw away half of n after each compalison.

How many times can we half n before getting to the result (1)? → log n.

eg How tall a binary tree do we need till we have n leaves?

How many times can we double 1 till we get to n? → log n.

4/16/21

Elementary Data Structures

There are two aspects to any data structure:

1.) Its abstract operations

2.) The implementation of these operations

That these are different implementations of the same abstract operations enables us to optimize performance in different circumstances.

Contiguous vs. Linked Data Structures

Data structures can be broadly grouped by how they are stored in memory:

1.) Contiguously allocated structures are composed of single slabs of memory → arrays, heaps, hash tables.

2.) Linked structures are composed of distinct chunks of memory joined by pointers.
→ linked lists, trees, graphs.

Arrays

An array is a fixed-size structure of records such that each element can be efficiently located by its index.

- Constant time (O(1)) access given an index.
- Physical continuity helps exploit high speed cache memory.

Simple arrays have fixed size. Dynamic arrays automatically reallocate its size n to 2n each time we run out of space.
→ $\log_2 n$ reallocations!

cf. slices in Go.

Linked Lists

```
type List struct {
    val  int
    next *List
}
```

Lend themselves well to recursive algorithms because their structure is recursive.

- Search → O(n) linear.
- Insert → O(1) if prepend, O(n) if append.
- Delete → O(n)

Insertions/deletions are simpler vs. arrays. With large records, moving pointers is easier and faster than moving the items themselves.

Doubly-linked lists maintain pointers in both directions.

Stacks and Queues

Stacks are last-in, first out (LIFO)

- Push inserts at top of stack.
- Pop removes and returns top item.

Queues are first-in, first out (FIFO)

- Enqueue inserts item at back of queue.
- Dequeue removes and returns front item.

Stacks are more easily represented as an array with push/pop incrementing/decrementing a counter.

Queues work well as linked list working on either end of the list.

All operations can be done in O(1) constant time.

Dictionary/Dynamic Set Operations

This is the interface for a kind of key/value store

"search" - get(K) retrieves value at key K.
"insert" - set(x) sets the value x in the set
- delete(x) removes the value x in the set
- min, max returns element w/smallest or largest key.
- predecessor successor returns next smallest or next largest key in a totally ordered set.

There are a variety of implementations for these dictionary operations, all have different time bound trade offs for each of these operations.

4/19/21

Dictionaries

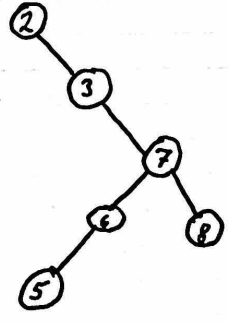
Let's dig into a concrete example of how using different implementations of a dictionary using linked lists affect operation efficiency.

Operation	Singly sorted	Singly unsorted
Search	O(n)	O(n)
Insert	O(n)	O(1)
Delete	O(n)	O(n)
Successor	O(1)	O(n)
Predecessor	O(n)	O(n)
Min	O(1)	O(n)
Max	O(1)	O(n)

Binary Search Tree

A BST efficiently supports all dictionary operations.

In a BST, each node has at most 2 child nodes, where for node x, left < x and right > x.



Searching a binary search tree takes $O(h)$, where h is the height of the tree.

Minimum is the left-most, and maximum is the right-most node. $O(h)$ to find either.

In order traversal (left → center → right) will take $O(n)$ because we must visit every node, duh.

For all dictionary operations, $O(h)$ is the time complexity. That said,

- if the BST is perfectly balanced, height = $\log n$
- severely unbalanced BSTs may have linear height, height = n

In practice (eg random insertion orders) BSTs on average have $\log n$ height.

When we talk about "balanced" trees, we mean trees whose height is $O(\log n)$.

No data structure can be better than $\Theta(\log n)$ on all dictionary operations.

Extra care must be taken on insertion/deletion to keep the BST balanced.

Red-Black trees, B-trees, splay trees — all examples.

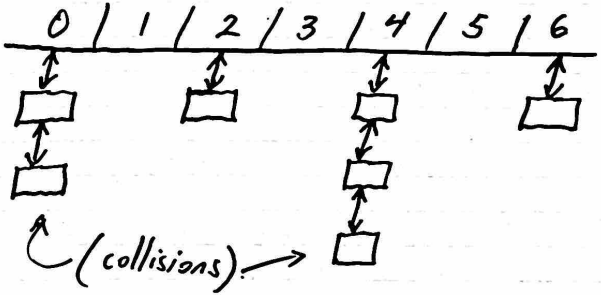
4/19/21

Hashing

A hash function is a mathematical function that maps keys to integers.

Hash tables are a very practical way to implement a dictionary, providing $\Theta(1)$ look-ups if you know the key.

If keys are mapped to the same integer bucket, you can append them to (short) lists:



A good hash function

1.) is easy to evaluate,

2.) tends to evenly distribute keys from 0...n buckets to minimize collisions.

Modular arithmetic is often used:

$h(K) = K \bmod M$, where M is a large prime.

Performance on dictionary operations

For most operations,

$O(1)$ expected, $O(n)$ worst case.

Pragmatically hash tables are often the best choice, though worst-case time is unpredictable. BSTs still provide better worst-case bounds.

Other uses for hashing - substring matching

Substring pattern matching - does text t contain the substring pattern p?

Brute force = $O(nm)$

where $n = |t|$ and $m = |p|$

overlaying the pattern p over every position in the text and comparing characters.

Instead we could compute a hash function on both p and the m-character substring starting from $t[i]$.

This seems to leave us with an $O(nm)$ algorithm again, where it takes $O(m)$ time to hash n substrings.

However, we can devise a hash function that allows to hash the ith position, and then derive $(i+1)$st position's hash in constant time. So this hashing substring pattern matching runs in

$O(n)$ time

where $n = |t|$.

Hashing as representation

Custom hashing functions can be used to bucket items by a canonical representation.

Proximity-preserving hashing techniques:

$h(\text{"o"}, \text{"d"}, \text{"g"}) \rightarrow$ god, dog

put similar keys in the same bucket.

4/20/21

Heapsort / Priority Queues

Sorting is important because once a set of items is sorted, many other problems become easy.

Applications of sorting

- Search pre-processing, eg ~~finite~~ binary search.
- Given n numbers, find the pair closest to each other.

- Is a set of numbers unique? Sort and check for adjacent pairs.
- Given a set of n items, which element occurs the largest number of times?

  Sort and do a linear scan of adjacent elements.

  The number of instances of some K can also be found with a binary search over a sorted set.
- What is the K^{th} largest item? Trivial if sorted, look at the item at index K.

Priority Queues

Priority Queues (PQs) are data structures that provide extra flexibility over sorting.

PQs allow you to insert without resorting everything with each new arrival.

1.) insert(x) - inserts item x into the queue.

2.) FindMin or FindMax - returns a pointer to the item whose key K is either the min or max.

3.) DeleteMin or DeleteMax — removes item with the min or max key.

Each of the P.Q. operations can be supported using heaps or balanced binary trees in $O(\log n)$.

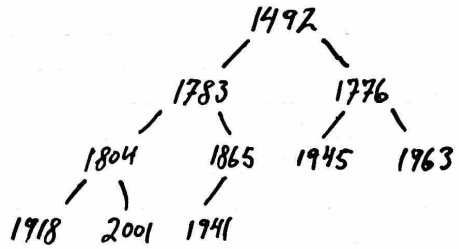
Binary Heaps

A binary heap is defined to be a binary tree with a key in each node such that

1.) All leaves are on at most two adjacent levels.

2.) All leaves on the lowest level occur to the left, and all levels except the lowest one are completely filled.

3.) The key in root is [either ≥ or ≤ than] all its children, and the left and right subtrees are again binary heaps.



1	1492
2	1783
3	1776
4	1804
5	1865
6	1945
7	1963
8	1918
9	2001
10	19481

A heap-labeled tree with its implicit representation as an array. In this example all parents are ≤, so it's a "min" heap.

NB | Heaps maintain a partial order that is weaker than sorted order, yet stronger than random order, so it is efficient to maintain and still easy to retrieve the minimum element, which is the root node (or max).

Heaps can be represented implicitly using an array, where

- for a node at index K
  - left child → 2K
  - right child → 2K + 1
- The parent of K is at K/2.

This array representation is not as flexible to modification as a pointer-based tree however.

Constructing Heaps

Heaps can be built by inserting new elements into the left-most open spot.

If the new element is greater than its parent, swap their positions and recur (or less than, depending on whether we want the root node to be min or max).

Since all but the last level is always filled, the height $h = \log n$.

Binary heaps are an excellent option for implementing priority queues.

Operation	Average	Worst
Insert	O(1)	O(log n)
FindMin	O(1)	O(1)
DeleteMin	O(log n)	O(log n)

4/21/21

Mergesort / Quicksort

Mergesort is a tidy recursive sorting algorithm that works by splitting the elements into two groups, sorting each group recursively, then merging the sorted groups.

```
func mergeSort(list) {
    if len(list) <= 1 {
        return list
    } else {
        middle = len(list) / 2

        left = mergeSort(list[0:middle])
        right = mergeSort(list[middle:])

    return merge(left, right)
    }
}
```

```
Func merge(left, right, merged)
    if len(left) == 0 && len(right) == 0 {
        return merged
    } else if len(left) == 0 {
        return merged + left
    } else if len(right) == 0 {
        return merged + right
    } else if left[0] < = right[0]
        merged = merged + left[0]
        return merge(left[1:], right, merged)
    } else if right[0] < left[0]
        merged = merged + left[0]
        return merge(left, right[1:], merged)
    }
}
```

Mergesort takes linear time to merge, and log time to "divide and conquer" the list, therefor

$$O(n \log n)$$

Quicksort is another divide and conquer algorithm:

1.) If the list has less than two elements, return immediately.

2.) Otherwise pick a value, called a pivot, in the list.

3.) Partition the list: reorder the elements, ~~so~~ while determing a point of division, so that all elements with values less than the pivot come before the division, while elements greater than the pivot come after.

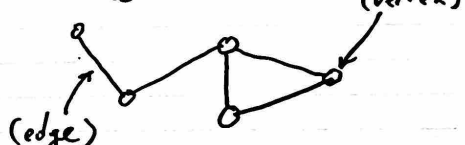
4.) Recursively apply the sort to the sublist to the left and the sublist to the right of the division point.

Runs in $O(n \log n)$ time typically.

4/22/21

Graph Data Structures

A graph $G = (V, E)$ is defined by a set of vertices V, and a set of edges E consisting of ordered or unordered pairs of vertices from V.

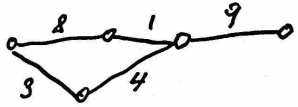


Directed vs. Undirected Graphs

Edges may have a direction:

Weighted vs. Unweighted

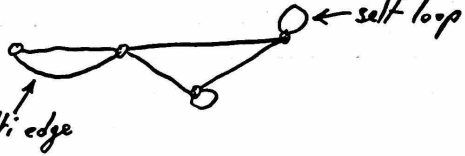
In weighted graphs, each edge or vertex is assigned a numerical weight:



Simple vs. Non-simple

Some types of edges make a graph more complex, or "non-simple":

- self-loop edges (x, x)
- multi-edge, same edge (x, y) happens more than once.



Sparse vs. Dense

Sparse graphs have few edges (linear relative to vertices) whereas dense graphs have many edges (quadratic relative to vertices).

Cyclic vs. Acyclic

An acyclic graph has no cycles. Trees are acyclic undirected graphs.

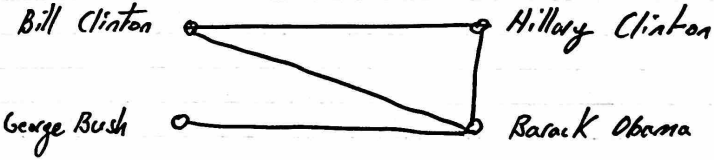
Directed acyclic graphs are called DAGs, and for edge (x, y) imply that x must come before y.

Embedded vs. Topological

A graph is embedded if the vertices and edges have assigned geometric positions. A topological graph doesn't care how it's arranged in space, just relationships.

Example: Friendship Graph

Consider a graph where vertices are people, and edges are friendships.



o Donald Trump

- If I am your friend, does that mean you are my friend?

ie is the graph undirected? Edge (x, y) implies (y, x).

- Am I linked by some chain of friends to a celebrity?

In other words, what is there a path from one vertex to another? What's the shortest path?

A graph is connected if there is a path between any two vertices.

A directed graph is strongly connected if there is a directed path between any two vertices.

- Who has the most friends?

The degree of a vertex is the number of edges adjacent to it.

- What is the largest clique?

A graph clique is a complete subgraph where each vertex pair has an edge between them.

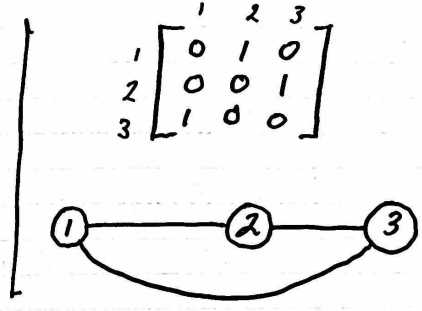
Data Structures for Graphs

1.) Adjacency Matrix

For a graph $G = (V, E)$ containing n vertices and m edges,

Represent G using an $n \times n$ matrix M, where

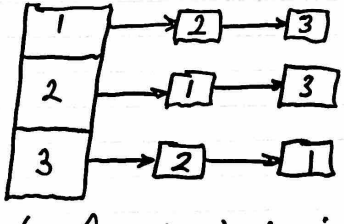
→ element $M[i, j]$ is 1 if (i, j) is an edge of G, and 0 if it isn't.



This representation may waste space if we have many vertices (n) but few edges (m)

2.) Adjacency List

Represents the graph as a $(n \times 1)$ array of pointers, where the i^{th} element points to a linked list of the edges incident on i.



To test if (i, j) is in the graph, we search the i^{th} element for j.

Tradeoffs between Adjacency Matrices & Lists

Operation	Winner
Faster to test (x,y)?	matrices
Faster to find vertex degree?	lists
Memory use? Small	lists $(m+n)$ vs. n^2
Memory use? Big	matrices (small win)
Edge insertion/deletion	matrices $O(1)$
Faster to traverse	lists $m+n$ vs. n^2

In practice lists are better for most problems.

4/23/21

Breadth-First Traversal

Graph traversal must be

1.) efficient - each edge is visited at most twice

2.) correct - every vertex is visited.

Marking Vertices

The key idea is that we must mark each vertex when we first visit it, and keep track of what we have not yet completely explored. Each vertex has one of three states:

1.) undiscovered - initial state.

2.) discovered - found the vertex, but haven't yet explored all its edges.

3.) processed - after we have discovered and explored all its incident edges.

To completely explore a vertex, we look at each edge going out of it. For each edge which goes to an undiscovered vertex, we mark it discovered and add it to the list of work to do.

Bread-First Traversal (BFT)

Breadth first search is appropriate for find shortest paths.

The algorithm relies on three data structures:

- "discovered" boolean array
- "processed" boolean array.
- a FIFO queue of discovered nodes to be processed, so the oldest vertices are processed first.

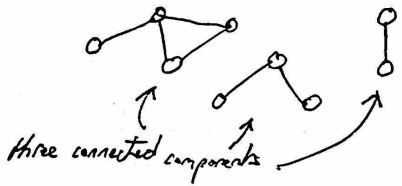
In BFT/BFS vertices are discovered in order of increasing distance from the root.

NB | This means the BFS path from the root to any node will be the shortest path!

While traversing with BFS we can build a parents array of vertices. We can then recurse from vertex x back to the root.

Connected Components

The connected components of an undirected graph are the separate "pieces" of the graph such that there is no connection between the pieces:



Anything we discover as part of BFS must be a connected component. We then repeat the search from any undiscovered/unconnected vertex to find other components.

4/26/21

Depth-First Search

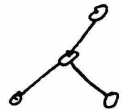
DFS is trivial to implement ~~recursively~~ recursively. Starting at root, recursively DFS each child vertex.

DFS is a form of back-tracking. Both involve exhaustively searching all possibilities by advancing if possible, and backing up as soon as there is no unexplored possibility for further advancement.

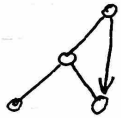
Edge Classification

Every graph edge is either:

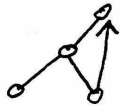
1.) A Tree Edge.



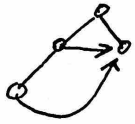
2.) A forward edge to a descendant.



3.) A back edge to an ancestor.



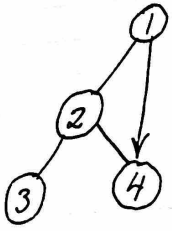
4.) A cross edge to a different node.



A key reason why DFS is so important is that it guarantees a particular edge ordering: in a DFS of an undirected graph, every edge is either a tree edge or a back edge.

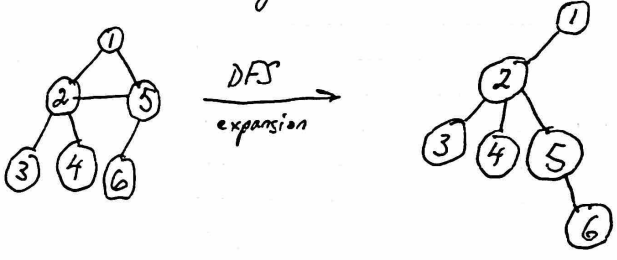
NB.

Here's an example: Suppose we have this graph with forward edge (1, 4):



Nonetheless, we will discover the back edge (4,1) before discovering (1,4) when using DFS.

Similarly, cross edges will be expanded as tree edges:



Why is this property of DFS useful?

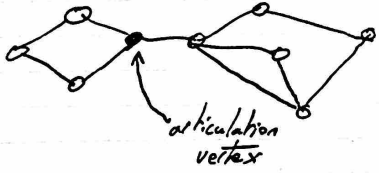
Finding Cycles

Back edges are the key to finding cycles in undirected graphs.

Any back edge going from x to an ancestor y indicates a cycle.

Finding articulation vertices

An articulation vertex is a vertex of a connected graph whose deletion disconnects the graph.

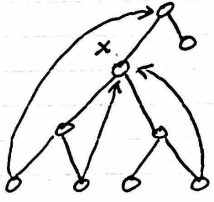


In a DFS tree, any vertex v (other than root) is an articulation vertex if

1.) v is not a leaf,

2.) Some subtree of v has ~~not~~ no back edge incident with an ancestor of v.

eg this DFS expansion:



"x" is an articulation because the right tree does not have a back edge to an ancestor of x.

Is it a Directed Acyclic Graph?

A directed graph is a DAG if and only if no back edges are encountered during a DFS.

4/28/21

Minimum Spanning Trees I

A tree is a connected graph with no cycles.

A spanning tree is a subgraph of some graph G which connects all vertices, and is a tree.

A minimum spanning tree of an edge-weighted graph G is a spanning tree whose edges sum to the minimum weight.

Minimum spanning trees (MSTs) are important because

- It is a problem where the greedy algorithm always gives the best answer. Greedy algorithms make the decision of what to do by selecting the best local option from available choices.
- Have many useful applications:
  - How to connect all vertices with, say, the minimum amount of wire?
  - Find natural vertex clusters.

Finding the MST: Prim's algorithm

1.) Starting from one vertex, grow the MST one edge at a time,

2.) Pick the smallest ("cheapest") edge which does not create a cycle.

Pseudocode:

```
PrimMST(G):
    Select arbitrary starting vertex
    While (there are still non-tree vertices)
        Select min weight edge between tree/non-tree
        Add the edge and vertex to tree Tprim.
```

Prim's algorithm can be implemented using a priority queue (see notes from 4/20/21).

1.) Add first node to queue

2.) Pull min node from queue, add to MST.

3.) Iterate over neighbors, if not already in MST, add to the priority queue.

4.) Continue until queue is empty.

Complexity is $O(E + V \cdot \log(V))$

where E = edges and V = vertices.

Kruskal's Algorithm

Another algorithm to find the MST, this is based on sorting the edges based on weight.

After that, pluck each edge in min → max order, skipping it if it creates a cycle.

The trick is detecting cycles efficiently. This is usually tracked using a disjoint set data structure. Using the disjoint set, we can merge two nodes into a single component. Then we can check if two nodes have been merged before, detecting the cycle.

Complexity is $O(E \cdot \log(V))$

Where E = edges and V = vertices.

5/2/21

Shortest Path

Finding the shortest path between two nodes in a graph has many different applications:

- Transportation logistics
- Motion planning / pathfinding
- Network diameters.

Remember T9 text prediction on old phones? That used a graph where the edges linked neighboring words as vertices. Edge weight was based on grammatical probability.

In an unweighted graph, shortest path can be found using breadth first search (see notes for 4/23/21).

It's more complicated with a weighted graph, because visiting more nodes may still sum to a lower weight. There can be an exponential number of shortest paths, so we cannot find all shortest paths efficiently.

Dijkstra's Algorithm

The principle behind this algorithm is that

> $(s \ldots x \ldots t)$ is the shortest path from s to t,
>
> $(s \ldots x)$ had better be the shortest path from s to x.

This suggests a dynamic programming-like strategy, very similar to Prim's algorithm (see 4/28/21 notes).

5/4/21

Backtracking I

Backtracking is the key to implementing exhaustive search programs. It is a method for iterating through all possible configurations of a search space.

Abstractly, we model our solution as a vector:

> $a = (a_1, a_2, a_3, \ldots a_n)$

where each element a_i is selected from a finite ordered set S_i.

At each step in the backtracking algorithm, we

start from a given partial solution, say, $(a, \ldots a_k)$, and try to extend it by adding an element.

After extending it, we test whether what we have so far is a complete solution.

If not, the critical question is whether the current partial solution *could* be extended to a complete solution.

- if so, recur and continue.
- if not, delete the latest element ("backtrack") from a and try another possibility if it exists.

Recursive Backtracking

```
Backtrack(a, K)

if a is a solution, return a
else {
    K = K+1
    compute S_K
    while S_K ≠ nil do
        a_K = an element in S_K
        S_K = S_K - a_K
        Backtrack(a, K)
}
```

Abstract backtrack function in pseudocode.

Backtracking as a form of DFS

Backtracking is really just *depth-first search* on an implicit graph of configurations.

- Backtracking allows us to iterate through all subsets and permutations of a set.
- Backtracking ensures correctness by enumerating all possibilities.
- For backtracking to be *efficient* we must prune dead branches of the search space whenever possible.

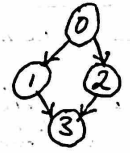
Example: DAG Paths

Given a directed acyclic graph (DAG) of n nodes labeled from 0 to $n-1$, find all possible paths from node 0 to node $n-1$.

The graph is modeled as follows: graph[i] is a list of all nodes you can visit from node i.

input: [[1,2], [3], [3], []]

output: [[0,1,3], [0,2,3]]



0→1→3 and 0→2→3

Backtracking solution in Go:

```
func findAllPaths(graph [][]int) [][]int {
    var paths [][]int
    n := len(graph)

    backtrack := func(node int, curPath []int){
        // Stop condition, end of the graph:
        if node == n-1 {
            paths = append(paths, curPath)
            return
        }
        lenCurPath := len(curPath)
        // Explore graph children:
        for _, child := range graph[node] {
            tmp := make([]int, lenCurPath, lenCurPath+1)
            _ := copy(tmp, curpath)
            tmp = append(tmp, child)
            backtrack(child, tmp)
        }
    }

    backtrack(0, []int{0})
    return paths
}
```

5/7/21

Dynamic Programming

Dynamic programming is a technique for efficiently computing recurrences by storing partial results.

eg. recursive n^{th} Fibonacci with memoization.

The trick to dynamic programming is to see that a naive recursive algorithm repeatedly computes the same subproblems over again, so storing/caching the answers in a table instead of recomputing leads to an efficient algorithm

5/12/21

Introduction to NP-Completeness

In computational complexity theory, we attempt to classify problems by how hard they are. There are many classifications; here are some important ones:

1.) P. Problems that can be solved in polynomial time; eg most common algorithms.

2.) NP. "Non-deterministic polynomial time", where "nondeterministic" just means we're talking about guessing a solution. A problem is NP if you can quickly (polynomial time) check if a solution is correct. But you're just verifying a guess!

The most famous open question in this field is

Does $P = NP$?

In other words, if it's always easy to check a solution (NP), should it also be easy to find a solution (P)?

NP-Completeness

The theory of NP-Completeness defines NP-complete problems as the hardest problems in P.

Even though we don't know whether there is any problem in NP that is not in P (NP=P?), we can at least point to an NP-complete problem and say that if there are any hard problems in NP, this is one of them.

So if we believe P and NP are unequal, and we can prove that a problem is NP-complete, then we can justifiably believe that the problem does not have a fast algorithm.

NP-Complete problems are problems whose status is unknown:

- No polynomial time algorithm has been found!
- Nor has it been proven that such an algorithm doesn't exist because we still don't know if P = NP.

NP-complete problems are the hardest problems in the NP set.

Assuming P != NP: