What makes a good algorithm?

1. Time efficiency
2. Space efficiency
3. Correct on all inputs
4. Readable/easy to understand

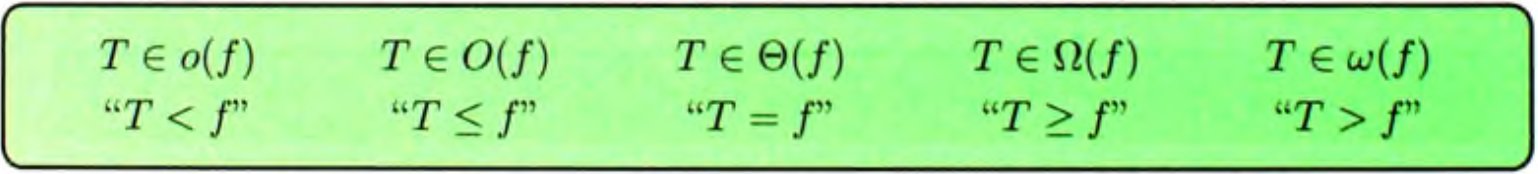
What is good runtime?

Small: O(n2), O(n), O(n log n)

**Unit Time:** O(1)

**Runtime between two functions:**

Given



**Cost:** The highest cost of the contents of the algorithm (AKA Big O notation)

**Omega:** lower bound

**Reduction**: If a problem is not solvable in linear time and you think you can solve a problem that would then allow you to solve the problem we agreed was impossible, then we have a contradiction

**Heisenberg Principle:** By observing a system you will change it

Call Stack:



# of recursive calls will be less than the number of recursive nodes in a complete binary tree of depth n-1

For a binary tree, at depth n - 1 we will have 22 nodes because it is a geometric series so:

for r = 2 and k = n - 1

Generally: f(n) <= 2n ← exponential in n

Improvement: stop recomputing and element redundancy in the call stack

*Try storing and referencing previously-computed numbers*

**Memorization:** compute number only once and store it

[Bit complexity](https://www.primidi.com/what_is_bit_complexity)

Ex: for the fibonacci sequence, create a table with n as the keys and the corresponding number in the sequence as the value, then reference them for the (n-1) and (n-2) lookup, reducing the runtime from O(2n) to O(n). Because we do n operations, we get (n-2+1)\*n = n2-n = n2

Claim: f(n) >= 2n \* ½ for n >= 6

Proof. This is a proof by induction

Base Case: n = 6 → 26/2 = 8 → 23 = 8

Step:



## Binary Computing/calculations

When you add two n-bit numbers., the result can be at most n+1 bits

**Bbinary multiplication:**

Its…its just normal multiplication….don’t forget that a carry 1 toggels the bit it’s carried to

You’re shifting n times so the runtime is O(n2)

**More precise version:**

function mult2(x, y) {

if (x == 0 || y == 0) return 0;

if (y == 1) return x;

z = mult2(x, (y / 2));

if (y % 2 == 0) return (2 \* z);

else return (2 \* z) + x;

}

*Time Complexity:*

Each call to mult2 works on (y/2) with n-1 bits → O(n)

You will make at most O(n) calls and O(n) \* O(n) = O(n2) time complexity

*Correcteness:*

*Case y is even:* re-write as y = 2k for some k → z = (2k)/2 = x \* k

2\*z = = 2 \* x \* y -> 2k = x\*y

*Case y is odd:* re-write as 2k+1 → y/2 = (2k+1)/2 = k + ½ = k

Z = xk → 2z = 2xk + x = x\*y

Q.E.D

mul3(x, y): reducing the number of bits by ½

x = xL | xR → x = 2^{2n}\*xL + xR → x \* y = (2^{n/2}xL + xR) \* (2^{n/2}\*yL + yR) →

x \* y = 2nxL \* yL + 2^{n/2}(xLyR + xRyL) + xR \* yR

Written Recursively:

T(n) = 4T(n/2) + O(n)

= 4(4T(n/8) + O(n/2)) + O(n)

= 42 (4T(n/4)) + 4O(n/2) + O(n)

= 42(4T(n/8) + O(n/4)) + 4 \* O(n/2) + O(n)

→ O(n) = O(n) = O(n) = O(n)

→ ← geometric series

**Gauss’ Trick:**

← 4 points of multiplication

| | |

## Modular Arithmetic:

← always between 0 and n - 1

→

Given some where, y-1 is the multiplicative inverse of

Def: the multiplicative inverse of y is a number between 0 and n - 1 such that

## Graph Algorithms:

**Graphs:** a mathematical way of representing relationships, made of nodes connected by edges

**Directed Graph:** the way we move down an edge matters

**Undirected Graph:** the way we move down an edge doesn’t matter

G = (V, E) where V = nodes and E = edges

* Captures pairwise relationships between objects
* Graph size parameters n = |V|, m = |E|

Def: A **path** in an undirected graph G = (V, E) is a sequence of P nodes v1 , v2 , …, vk with the property that each consecutive pair vi , vi+1 is joined by an edge in E

Def: A path is **simple** if all nodes are distinct

Def: A **graph** is **simple** if there are no multiple edges between two nodes and no self-loops

*Note: a single node CAN BE A PATH*

Def: A **cycle** is a path which starts and ends on the same node, and is thus not a simple path

*Note: there MUST be more than one node in order for a path to be a cycle*

Def: A **simple cycle** is a cycle in which all nodes don’t repeat (you don’t go through nodes more than once). AKA there are k-1 distinct nodes in a simple cycle

*Note: In a graph where n = |V|, you can have at most edges*

*Note: in terms of n and m, the largest simple path is min(n-1, m) and the largest simple cycle is min(n, m)*

Def: **connectivity**

## Chapter 2

### 2.1 - Multiplication (review this again)

The **divide-and-conquer strategy** solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of

problem

1. Recursively solving these subproblems
2. Appropriately combining their answers

**Product of Two Complex Numbers:**

**Runtimes and Trees:**

If a problem’s size is halved each recursion, then the tree has hight log2(n)

For a divide and conquer algorithm, the tree should have a depth of k = logb(n)

At k = 0, the ABOVE tree has runtime O(n) and at k = log2(n), the tree has a runtime of O(

if even and

*If the problem size is halved at each iteration, then it has a depth of O(log2(n))*

Eventually, we write a recurrence formula for the running time of an algorithm:

T(n) = T(n/2) + mult(n/2)

= T(n/2) + (n/2)2

= [T(n/4) + O(n/4)2] + (n2/4)

= T(n/4) + O(n2/16) + O(n2/4)

= n2/4 + n2/16 + n2/64

= n2/4(1+ (¼) + (1/16) +.....) ← geometric series <= 2

T(n) <= (n2/4) \* 2 <= (n2/2) → O(n2)

Fib3(n): O(n2) → O(n2log(n)) → O(n2)

*But we can do faster matrix multiplication O(n1.6) ≈ O(???????)*

Fib4(n): where the labdas are the eigen values of matrix M



If M =

M \* v = \* v for any value of that solves the equation is an eigen value of M, and corresponding v is the eigen vector

*Note To Self*

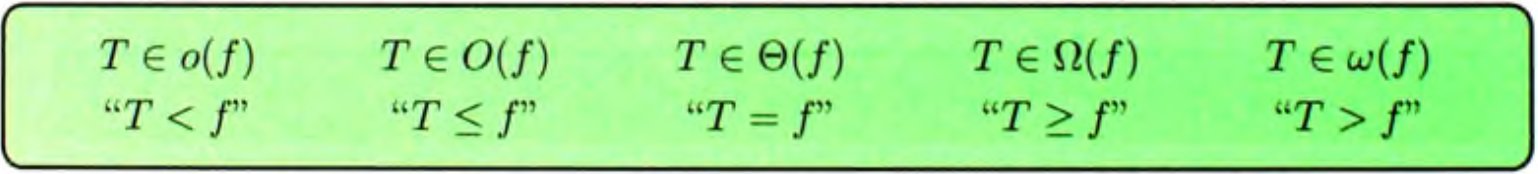
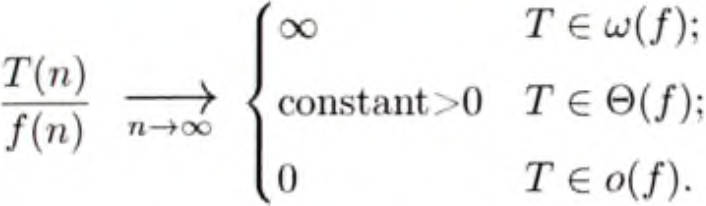
We write the characteristic equation | M - | = 0 and solve it\

**Asymptotic Time Analysis with Big O notation:**



We say that f(n) = O(g(n)) if there exists a constant c > 0 such that for all n > n0 f(n) <= C \* g(n)

*Note: we do NOT care about what happens before n0 in this case* *AKA there will be a section where 1000\*n2 will be greater than n3, and we might choose to ignore that?*



*Note: for the Constant > 0 case, it means that T(n) and f(n) have the same growth rate but can have different constants*

## GCD

**Prime Factorization:**

Repeatedly split each number into two separate numbers. When you reach a prime number, circle it and stop. For GCD, pick the largest number from each “tree”

**Euclid’s Algorithm:**

and do that repeatedly until it becomes obvious what the GCD is

**Euclid’s Algorithm:**

function Euclid(a, b):

if b = 0: return a

return Euclid(b, a mod b)

**Linear Combination:**

**Congruence:**

x is equivalent to y

is defined as x and y have the same remainder when divided by N

⇒ x-y is divisible by N and N|(n-y) is the solution

**Division(x, y):**

function divide(x, y)

if x = 0: return (q, r) = (0, 0)

(q, r) = divide(⌊x/2⌋, y)

q = 2 · q, r = 2 · r

if x is odd: r = r + 1

if r ≥ y: r = r − y, q = q + 1

return (q, r)

**Time Analysis:**

1. n recursive calls each time x is halved ⇒ 1 bit less due to right-shift
2. The cost of each call:

* 2 multiplictions
* A check if x is even
* Possible additio by 1
* Possible subtraction by y
* Total cost: n

**Total Cost:** O(n\*n) = O(n2)

**Correctness:**

????????

→

So and

So

**Modular Exponentiation (pseudocode):**

function **modexp**(x, y, N)

if y = 0: return 1

z = modexp(x, ⌊y/2⌋, N)

if y is even: return z2 mod N

else:

return x \* z2 mod N

**Modular Exponentiation:**

def **modexp**(x, y, N):

if (y == 0): return 1;

z = modexp(x, floor(y/2), N);

if (y % 2 == 0):

return (z \* z) % N;

else:

return (x \* z \* z) % N;

def **primality**(N):

r = modexp(random.randint(N), N-1, N);

if (r != 1): return True;

return False;

**Addition in modular Arithmetic:**

(A + B) mod C = (A mod C + B mod C) mod C

**Proof for Addition Arithmetic:**

We will prove that (A + B) mod C = (A mod C + B mod C) mod C

We must show that LHS=RHS

From the quotient remainder theorem we can write A and B as:

A = C \* Q1 + R1 where 0 ≤ R1 < C and Q1 is some integer. A mod C = R1

B = C \* Q2 + R2 where 0 ≤ R2 < C and Q2 is some integer. B mod C = R2

(A + B) = C \* (Q1 + Q2) + R1+R2

LHS = (A + B) mod C

LHS = (C \* (Q1 + Q2) + R1+ R2) mod C

We can eliminate the multiples of C when we take the mod C

LHS = (R1 + R2) mod C

RHS = (A mod C + B mod C) mod C

RHS = (R1 + R2) mod C

LHS=RHS= (R1 + R2) mod C

**Subtraction for modular arithmetic:**

(A - B) mod C = (A mod C - B mod C) mod C

**Modular Multiplicative Inverse:**

Find the inverse of a, b:

1. Euclid’s Algo: if d != 1, there is no multiplicative inverse

*A modular multiplicative inverse of an integer* a *is an integer* x *such that*

ex)

Solving Methods (video I got these from is [**here**](https://www.youtube.com/watch?v=mzEvIN8BuQ8&t=29s)):

*Plug and chug:* so try

* ❌
* ❌
* …..
* ✅

*Use the Extended Euclidian Aglorithm:*

1. can be re-written in quotient-remainder theorem:
2. ← because the gcd is 1, we can solve for the multiplicative inverse
3. Use Linear Combination
   1. ← from here we see that x = 3, y = -1, a = 5, b = 14

*Divide and Plug:*

1. Solve:
2. Check:

**Bezout’s Identity:** for

Euclid’s Algorithm - gcd(m, n) = gcd(rem(n, m), m)

Euclid’s Lemma - If d|m\*n AND gcd(m, n) = 1, then d|n

Relative Primes: mx + ny = 1

**Fermat’s Last Theorem:**

It is impossible for a cube to be the sum of two cubes

**Fermat’s Little Theorem:**

General Case:

*if p is a* ***prime number****, then for any integer a, the number aP – a is an integer multiple of p*

Special Case:

*If a is not divisible by p, Fermat’s little theorem is equivalent to the statement that a p-1-1 is an integer multiple of p*

**Euler’s Theorem:**

If n and a are **coprime** positive integers, and φ(n) is the Euler’s Totient function, then a raised to the power φ(n) is congruent to 1 mod(n)

AKA

**Euler's Totient Function:**

counts the positive integers up to a given integer n that are relatively prime to n

ex) the totatives of n = 9 are the six numbers 1, 2, 4, 5, 7 and 8. They are all relatively prime to 9, but the other three numbers in this range, 3, 6, and 9 are not, since gcd(9, 3) = gcd(9, 6) = 3 and gcd(9, 9) = 9.

Therefore, φ(9) = 6

*Note: Euler's totient function is a multiplicative function, meaning that if two numbers m and n are relatively prime, then φ(mn) = φ(m)φ(n)*

**Runtime (**[**reference**](https://en.wikipedia.org/wiki/Euler%27s_totient_function)**):**



**Carmichael’s Numbers:**

* Numbers that pass the primality test but are not prime numbers

**Randomized Algorithms:**

Types:

1. Monte Carlo Algorithms

* Give correct answer with high probability
* Sample, then solve the problem on said sample
* An example of this is the Factory Example a few pages down

1. Las Vegas

* Always give correct answers but the runtime is random
* Examples are quick sort and quick select (runtime depends on random pivot)

**LOOK AT THE HASH FUNCTION PART OF THE LECTURES AGAIN**

**Probability of an Event:**

**Universal Hash Functions:**

**Approach 1:**

← ℤm defines the “mod loop”

*n* can be any number but *m* must be **prime**

*Proof:*

Show

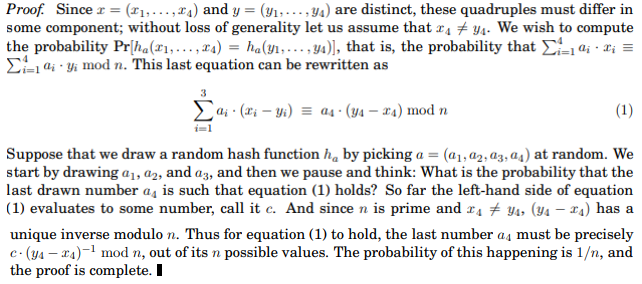
Cases:

Let us define as *r* and as *s*

So we can then re-write this as: and therefore

*Note: the reason is on top is because it is the interaction of mods* m *and* n *(I think) and the reason is on the bottom is because* s *is bounded by mod(m)*

**Approach 2:**



**General proof:**

*Goal: Compute expected lookup time*

Consider a key x



[**Linearity of Expectation:**](https://www.geeksforgeeks.org/linearity-of-expectation/)

**Probabilistic Analysis of an Algorithm:**

1. Exact (more detailed and challenging) WILL NOT DO IN ALGO
2. Fast and good approximation using an Indicator Random Variable which is either 0 or 1

*Example: Birthday Paradox:*

Question: how many people must we have in a room for a 50% chance of a shared birthday?

Solving:

1. Assign each person an index 1 through k
2. Assume birthdays are independent

for:

* can be rewritten as
* can be rewritten as

Recall from the k-chose-r algorithms that can be rewritten as

Multiply them to get

Plug in *365* for the days of the year to get

**Application: Divide and Conquer**

normally or

using Euclid’s method (3-mult)

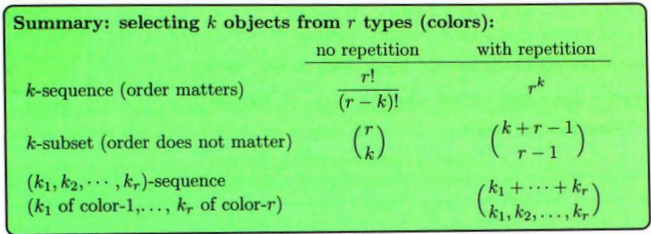
Params:

* Problem size (n)
* Number of sub-problems (a)
* Size of each sub-problem (
* How much it costs to combine the answers

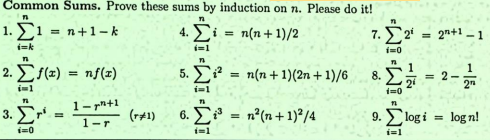
**Master Theorem:**

If for some





**Common Sums**



**QuickSort (Hoare’s Method):**

import random;

def **quicksort**(arr, start, stop):

if(start < stop):

pivotindex = partitionrand(arr, start, stop)

quicksort(arr , start , pivotindex)

quicksort(arr, pivotindex + 1, stop)

def **partitionrand**(arr , start, stop):

randpivot = random.randrange(start, stop)

arr[start], arr[randpivot] = arr[randpivot], arr[start]

return partition(arr, start, stop)

def **partition**(arr,start,stop):

pivot = start *#pivot*

i = start - 1

j = stop + 1

while True:

while True:

i = i + 1

if arr[i] >= arr[pivot]:

break

while True:

j = j - 1

if arr[j] <= arr[pivot]:

break

if i >= j:

return j

arr[i] , arr[j] = arr[j] , arr[i]

# Driver Code

if \_\_name\_\_ == "\_\_main\_\_":

array = [10, 7, 8, 9, 1, 5]

quicksort(array, 0, len(array) - 1)

print(array)

*Note: The commented version can be found* [*here*](https://www.geeksforgeeks.org/quicksort-using-random-pivoting/)

## BEGIN LECTURE NOTES FOR 2/11/23

**Quicksort Algorithm:**

* Randomized selection of the pivot element
* Probabilistic analysis to computer the average case of behavior

**Divide and Conquer - Matrix Multiplication:**

→ the ith row and jth column of Z

We can look at this as “we have n2 elements and each one takes O(n) to compute” → total runtime: O(n3)

We can reduce this via divide and conquer:

**Strassen’s Algorithm:**



This can be re-written as:

← no improvement, terrible

**Applying Gauss’ Trick:**



P1 = A(F - H)

P2 = H(A + B)

P3 = E(C + D)

P4 = G(G - E)

P5 = (A + D) \* (E + H)

P6 = (B - D) \* (G + H)

P7 = (A - C) \* (E \* F)

**Graph Algorithms:**

Graph:

G = (V, E)

V is the set of vertices of size n

E is the set of edges (or links) of size m

and there is some relationship between u and v

**Handshaking Theorem:** sum of vertex degrees = 2 |E|

**Isomorphic:** all edges/vertices are in both graphs  
**Disjoint:** don’t share any common edges or vertices

**Connected:** every pair of vertices is connected

**Induced subgraph:** requires all edges to be included

**Bipartite Graphs:** every edge in V1 ONLY connects to edges in V2 and vica versa

**Directed Graphs:** the way we travel along edges matters

**Degree of a Vertex:** the amount of edges connected that node

Example - Highway maps:

If a highway connect two cities (u and v) then the line between them would be in E

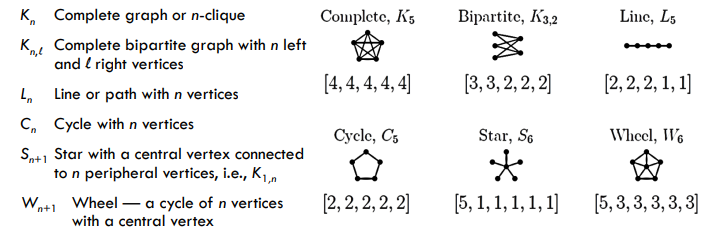
* Can be used to model temporal relationships (ex: cut → sand → polish)

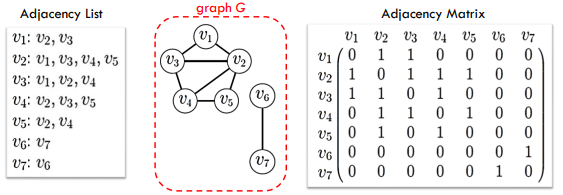
Simple Graphs:

No self-loops and no multiple edges between two nodes

*Note: an edge is a pairwise relationship. If an edge is a set ({ }) relationship, then we have Hypergraph*

Other Types of Graphs:





**Spanning Tree:** connected using minimum # of edges

**Planar Graphs:**

No lines overlap (vertex positions may change when converting)

**Weighted Graphs:**

Weights is a user-given value for each edge (AKA graphs with edge length?)

Example:



The weighting above w = (0.7, 0.2, 0.3) represents the probability taking each path would give

**Cycle:** you can make it back to the original node without retracing any paths

If the graphs is sparse [O(m) = O(n)] then prefer the adjacency matrix list. If the graph is

dense [O(m) = O(n2)] then prefer the adjacency matrix.

**Traversing Graphs:**

**Walk:** any alternating matrix vertex-edge pairs in a graph (vertex→edge→vvertex→edge)

**Path:** is a walk without any duplicate vertices (AKA no cycles)

**Searching Graphs:**

Given graph G = (V, E) and a vertex *u* in the vertex set, what other vertices are reachable from *u*

* If all the reachable nodes/vertices are the same as the entire vertex set, then G is **connected**

[**Depth-first search algorithm (DFS):**](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/)

N(u) = neighborhood of u (i.e. the nodes that are adjacent to u)

DFS(u):

visited(u) = true;

for x in N(u):

If (!visited):

DFS(x);

This creates a tree (a graph with no cycles)

*We need some bookkeeping during the DFS: Store pre and post order numbers for visiting nodes. For example, when going down the tree, give each node a number and assign another number when going back. We will then use these numbers to:*

* Get different types of edges (back-edge, tree-link, sibling-link)

*END LECTURE 9*

In an undirected graph, |N(u)| = degree of u

In a directed graph, the **in-degree** of u is the number of links (edges) heading to u and the **out-degree** is the number of edges headed away from u

**Source Node:** a node with only outgoing edges

**Sink Node:** a node with only incoming edges

**Sparse Graph:** O(m) = O(n)

**Dense Graph:** O(m) = O(n2)

*Note: pre and post refer to the numbers each node is given on the way “up” and “down”a graph*

Property: for any nodes u and v two intervals (pre u, post u) and (pre v, post v) are either:

1. Are contained in each other or
2. Disjoint

In Directed Graphs:

DFS tree has the root as the Source Node

“Sibling” nodes have the same parent node

After we run DFS on a directed graph and assign pre and post order numbers then we can identify different types of links in the graphs with respect to DFS

**Types of Links:**

1. Tree links/edges
2. foreward edges
3. Back edges
4. Cross edges: join two branches of the tree (*Note: this WILL create cycles*)

A directed graph has a cycle IFF its DFS tree has a back-edge

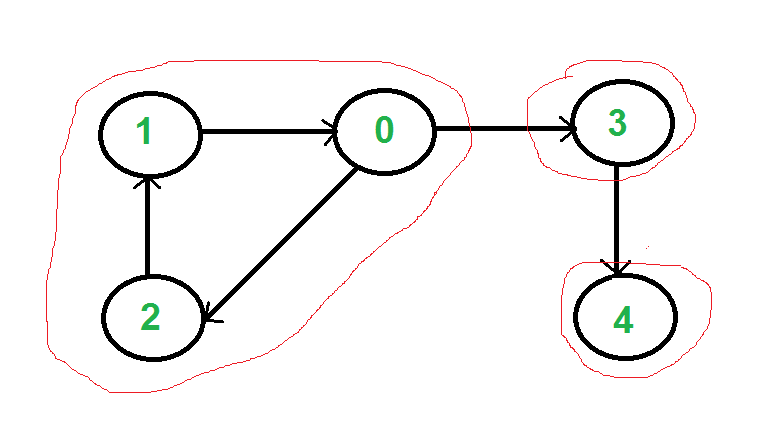
Why: if (u, v) is a back edge then there is a cycle going from u to v and a back-edge from v to u

Conversely, if the graph has a cycle, the DFS tree must have a back edge

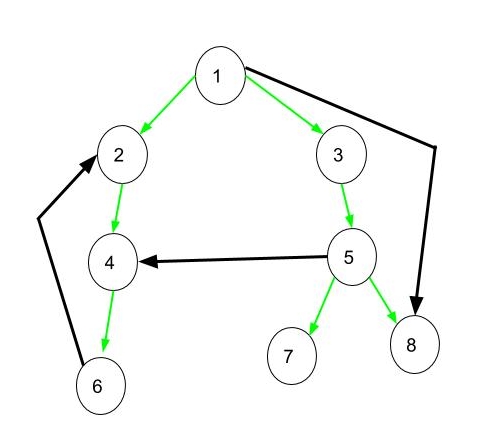
*Note: if you want to check for a cycle, delete a node and check if you can still reach all the nodes you could before. If you can, there is a cycle*

**Strongly Connected Components (SCCs) in Connected Graphs:**

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (**SCC**) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph. [LINK HERE](https://www.geeksforgeeks.org/strongly-connected-components/)



Consider a directed graph given in below, DFS of the below graph is 1 2 4 6 3 5 7 8. In below diagram if [DFS](https://www.geeksforgeeks.org/depth-first-traversal-for-a-graph/) is applied on this graph a tree is obtained which is connected using green edges. [LINK HERE](https://www.geeksforgeeks.org/tree-back-edge-and-cross-edges-in-dfs-of-graph/)



* **Tree Edge**: It is an edge which is present in the tree obtained after applying DFS on the graph. All the Green edges are tree edges.
* **Forward Edge**: It is an edge (u, v) such that v is a descendant but not part of the DFS tree. An edge from **1 to 8** is a forward edge.
* **Back edge**: It is an edge (u, v) such that v is the ancestor of node u but is not part of the DFS tree. Edge from **6 to 2** is a back edge. [Presence of back edge indicates a cycle in directed graph](https://www.geeksforgeeks.org/detect-cycle-in-a-graph/).
* **Cross Edge**: It is an edge that connects two nodes such that they do not have any ancestor and a descendant relationship between them. The edge from nodes **5 to 4** is a cross edge.

**Directed Acyclic Graph (DAG):**

* A directed graph that has no cycles
* Every node leads to a vertex with a lower post number

Given a DAG we can linearize it using [**Topological Sort**](https://www.geeksforgeeks.org/topological-sorting/) in decreasing order of post numbers

Intuition: this order ensures that all of the pre-requisites of a node must be satisfied before that node is visited

*Note: a helpful tutorial video can be found* [*here*](https://youtu.be/eL-KzMXSXXI)

**Topological Sort Algorithm(G):**

1. Do a DFS numbering of pre, post visits
   * O(|V| + |E|) ⇒ O(n + m) ← linear graph
2. Output vertices in revere-order of the post-ordered list
   * This will give a sorted list in O(|V|) ⇒ O(n)

*Note: a strongly-connected component graph will give us a DAG this also gives us an algorithm to compute strongly-connected components (SCCs)*

*END LECTURE 10*

**SCC Algoorithm:**

1. Identify all the sink SCCs
   1. The one with the highest post-number after DFS must be the source in SCC
2. Do a DFS from each sink node to construct the SCC then remove it (mark all the nodes as visited)

**Properties:**

1. Every directed graph is a DAG of it’s SCC
2. Explore(u) starting from node u will cover/reach all nodes reachable from u
3. The node that has the highest post-number in the first search must be in a source SCC
4. If we have two strongly connected components C1 and C2 and there is a link from a node in C1 to a node in C2 then the highest post-number in C1 is larger than the one in C2

**Reverse Graphs:** The reverse graph of G is GR in which all the edges of G have reversed directions

*Note: This would make the old souce a sink*

**Revised Algorithm:**

DFS(GR)

1. Run DFS
2. Traverse in decreasing order of post-numbers and find the nodes reachable form V

[**Kosarajus Algorithm:**](https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm) ([YouTube video link](https://www.youtube.com/watch?v=RpgcYiky7uw))

The primitive graph operations that the algorithm uses are to enumerate the vertices of the graph, to store data per vertex (if not in the graph data structure itself, then in some table that can use vertices as indices), to enumerate the out-neighbours of a vertex (traverse edges in the forward direction), and to enumerate the in-neighbours of a vertex (traverse edges in the backward direction); however the last can be done without, at the price of constructing a representation of the transpose graph during the forward traversal phase. The only additional data structure needed by the algorithm is an ordered list L of graph vertices, that will grow to contain each vertex once.

If strong components are to be represented by appointing a separate root vertex for each component, and assigning to each vertex the root vertex of its component, then Kosaraju's algorithm can be stated as follows.

1. For each vertex *u* of the graph, mark *u* as unvisited. Let L be empty.
2. For each vertex *u* of the graph do Visit(u), where Visit(u) is the recursive subroutine:  
    If *u* is unvisited then:
   1. Mark *u* as visited.
   2. For each out-neighbor *v* of *u*, if *v* is unvisited, do Visit(v).
   3. Prepend *u* to L.
3. Otherwise do nothing.
4. For each element *u* of L in order, do Assign(u,u) where Assign(u, root) is the recursive subroutine:  
    If *u* has not been assigned to a component then:
   1. Assign *u* as belonging to the component whose root is *root*.
   2. For each in-neighbor *v* of *u*, do Assign(v, root).
5. Otherwise do nothing.

**Review of Chapers:**

Chapter 0:

* Growth of functions
  + Describe the behavior as the function reaches it’s limit
  + Asymptotic behavior
  + Analysis of Algorithms with respect to time complexity
  + Ignore low-order terms/constants (Big O)
  + Bounds: Big O (upper bound), Big Θ (exact bound) Big ⍵ (lower bound)
* I am skipping most exponent and log rules if you wanna watch that it’s from 41:30 to 52:00

Chapter 1:

* A number N requires bits to represent AKA
  + This makes some algorithm’s bit-complexity exponential
* Modular Arithmetic
  + Addition, multiplication, exponentiation, inverse
  + We can take advantage of intermediate reductions
  + (bc 4 mod(3) = 1)
  + One can split the modulus into prime numbers and solve with each of those using Euler’sTheorem/varient (see homework 2?)

Chapter 2:

* Divide and conquer
  + Sorting
    - Lower-bounds on sorting problems
  + Medains
  + Splitting lists
* Randomization
  + Primality tests
  + Deterministic algorithms improvement
    - Pivot elements help avoid worst-case behavior
* Hash collision
  + Probabilistic analysis by rnadomized aolgorithm (has collisions)
  + Random indicator variables (see the Birthday Paradox)
  + Universal Hash Functions (all the stupid stuff)
  + Probability and Expectation

Chapter 3:

* DFS
* SCC
* Linearization
  + Given a graph, determine SCC, as well as sink and source nodes

--------------------------------------- I was dumb and started at lecture 11 ------------------------------------------

Given an unsorted array of n integers

You must find an integer which is >= the median of the array

Deterministic: MUST look at entries

Randomized Sampling: Choose C elements at random, then return the max of those.

Runtime: O(C)

Probability of failure: probability that all C elements are smaller than the medium. Recall that every element is independent so P(½) for every element in C or for all elements in C.

*Note: for larger lists, the probability if failure is essentially 0 because is tiny irrelevant of what n is*

Example - Factory Testing:

Let's say that we make billions of items and we want to check if < 99% of items have no flaws. Even if we only test 1000 of them, and none of them have flaws, then the probability that all items have no flaws would be , which is basically 0.

Example - Find a median:

Given an array of n inserted integers, find the median faster than linear time

This isn’t really possible, but we *can* find an ε-approximate median (within ε of the median):

Choose O() elements randomly and find the median of those.

Runtime: O() to find those and to sort so total runtime is

O(), which is tiny compared to the actual size of the array

Results in an ε-approximate median with a probability

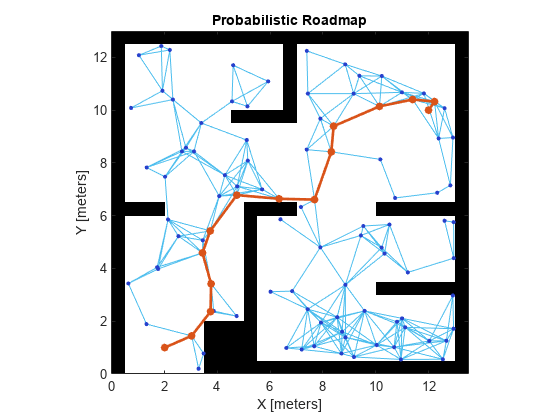
Proof ??????:

**Non-Array Monte-Carlo Algorithm Uses:**

* [Estimating PI](https://academo.org/demos/estimating-pi-monte-carlo/)

PRM Characteristic Rapmap Planning:

Given an area surrounded by obstacles, try random points and check if they fall on an obstacle, ignore them. Once you get enough points, check the area around them and connect them.



*Note: This is actually how machine learning works, as the model will check “random” data points and look for patterns using local search (linear/logistic regression)*

*END LECTURE 11*

**Dijkstra shortest path algorithm using Prim’s Algorithm in O(V2) [**[**LINK**](https://youtu.be/bZkzH5x0SKU)**]:**

1. Make a new list of unvisited nodes and a list of visited nodes
2. Create a table with three columns
   1. The node name
   2. The shortest distance
   3. The previous node
3. Assign to all nodes a tentative distance value of ∞ except the root node, to which you assign a value of 0
4. Loop through the root node’s neighbors, adding the line weight and the root node as their previous node in the table
5. Move the root to the “visited” queue, then pick the neighbor with the smallest weight and make it the new root node
6. Add the new root’s weight to each of its unvisited neighbors and calculate the new shortest path, loop through them and start the process over again until you reach the last unvisited node, marking it as visited

To get the shortest path, simply start at any node and work your way back through the list, going to the previous node each time

## Lecture 13 - REFACTOR THESE NOTES ASAP

* The midterm will be on chapters 0-4
* Check syllabus for CRIB sheet

*Last time we looked at reachability in graphs*

Which nodes are reachable from u?

* Run DFS to see

*Note: See homework 5 (and* [*this section*](https://docs.google.com/document/d/1atwgQnQQkbhdqNoLACfZOk2ZfnFkE3xqdSBURxjNwIg/edit?disco=AAAAsHqYha4)*) for more info on DFS*

In a directed graph, consider using Koserajus Algorithm to turn it into a DAG

In an undirected graph, see [this link](https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/)

**Finding the Shortest Distances Between Nodes in a Graph:**

* Unweighted graphs: BFS tree
* Weighted graphs: Use Dijkstra’s algorithm
* “ “ with some weight’s negative value ⇒ [check lecture notes]
* “Negative cycles” ⇒ WTF??????

sub-set of the nodes that are idk man his lectures are so ass I need to watch the videos/review the notes/chapters

How many links do we need to cut to make the graph un-connected?

* if we have E edges and N nodes in a connected graph we can remove E-N+1 edges so that graph remains connected.
* How to do this?
  + Just do DFS/BFS to find any spanning tree of the graph
  + since the spanning tree is connected we can just remove all other edges

What in god’s name is FIFO?

[FIFO (First-In-First-Out) approach in Programming - GeeksforGeeks](https://www.geeksforgeeks.org/fifo-first-in-first-out-approach-in-programming/)

dist(u) + w(uv) is minimized

* dist(u) is the distance from the origin to node u
* What is the rest one?
* dist(u) + w(uv) < dist(v)
* dist(v) <= dist(u) + w(uv) ?
* u → parent(V) ?

Once you “split” the graph into “partitions”, you can…do something?

This is called “cutting” the graph

**Negative Edge Weights:**

The edge has a negative weight

* You can find these by running Dijkstra’s algo twice, and if there is a difference, there is a shortest-edge cycle
* This is bizarre, look into further

**The Binary Heap:**

* A data structure
* Ex: Organize elements (i.e vertices) into a binary tree
* This tree has a **heap** property, that has a weight/key value which is:
  + Smaller than both its children ⇒ min heap
  + Larger than both it’s children ⇒ max heap
* Therefore, the minimum value will be in the root in a min-heap

<https://www.geeksforgeeks.org/heap-data-structure/?ref=gcse>

*Note: building a heap should take O(n) - IMPORTANT NOTE*

**How is Binary Heap represented?**

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.

* The root element will be at Arr[0].
* Below table shows indexes of other nodes for the ith node, i.e., Arr[i]:

| Arr[(i-1)/2] | Returns the parent node |
| --- | --- |
| Arr[(2\*i)+1] | Returns the left child node |
| Arr[(2\*i)+2] | Returns the right child node |

**Midterm (Questions?) and Information:**

1 question about randomization and random indicator variables

2 questions about graphs

1 or 2 questions on solving recurrence relation using Master’s Theorem

Questions on asymptotic functions and bounds

Be comfortable with Divide and Conquer algorithms

ONE-PAGE CRIB SHEET (FRONT AND BACK)

**Begin Lecture 3/2/2023:**

To find the shortest path in a graph (**Single source shortest path problem):**

1. Weighted
   1. Negative weight cycles
      1. Problem is no solvable
      2. Run ? one more time, if the shortest path changes, there is a neg weight cycle
   2. No negative weight cycles
      1. Run Deikstra’s Algoritm
2. Unweighted
   1. Run BFS

**All pairs shortest path problem**:

* Terrible, will not cover in this class but here’s a [link](https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/) anyways

**Bellman-Ford StyleShortest Path Algorithm [**[**LINK**](https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/)**]:**

Given a weighted, directed graph G = (V, E) and a weight function w: E → R

return YES or NO answer for indicating whether there is a

negative-weight cycle reachable from the source node

* This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is the source vertex
  + Runtime:
* This step calculates shortest distances. Do the following |V|-1 times where |V| is the number of vertices in the given graph. Do the following for each edge u-v
  + If dist[v] > dist[u] + weight of edge uv, then update dist[v] to dist[v] = dist[u] + weight of edge uv
  + Runtime: O(|V| \* |E|)
* This step reports if there is a negative weight cycle in the graph. Again traverse every edge and do the following for each edge u-v
  + If dist[v] > dist[u] + weight of edge uv, then **Graph contains a negative weight cycle**
* The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn’t contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

**Optimal Substitute Property Lemma [**[**LINK**](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)**]:** Subpaths of shortest paths are also shortest paths

You can **Prove** this by **Contradiction:**

Assume there exists another path p’ such that the cost of the path is less than the cost of path p form i to j

1. Do a cut and paste operation, splitting the path p and replacing it with p’
   1. This would make the cost smaller, which contradicts the fact that p is the smallest path

Therefore, we have proven that p is the shortest path from i to j

## BEGIN LECTURE 3/16/23 (#15?)

**MST Problems:**

* Minimum edge weight
* Till has all vertices

Distinct MST = Disjoint MST ⇒

**Cut [**[**LINK**](https://en.wikipedia.org/wiki/Cut_%28graph_theory%29)**]:** a partition of the vertices of a graph into two disjoint subsets. Any cut determines a **cut-set**, the set of edges that have one endpoint in each subset of the partition. These edges are said to cross the cut. In a connected graph, each cut-set determines a unique cut, and in some cases cuts are identified with their cut-sets rather than with their vertex partitions.

**Prim’s Algorithm:**

* There always exists a sub-tree with optimal width (least cost)
* To use the cut property in a greedy fashion to complete the tree?
* Cut that “requests” T = (V, E). If there is an edge in E’ crosses from S to S’ and vice versa

*Side Note*

**Union-Find Algorithm [**[**LINK**](https://www.geeksforgeeks.org/union-by-rank-and-path-compression-in-union-find-algorithm/)**] O(log(|V|)):**

*Note: This is used in Kruskal’s Algorithm*

*Note: there are two implementations of this algorithm, but we will only be focusing on one, for information on the other, see* [***this link***](https://www.geeksforgeeks.org/introduction-to-greedy-algorithm-data-structures-and-algorithm-tutorials/)

* Used to determine if there is a cycle in a graph
* Has two operations: union and find
  + Union: joins the group some x is in with the group some y is in
  + Find: finds the **root** of the group x or y is in

Union procedure (Pseudocode): Union(x, y)

* Parent(find(y)) = find(x)

Union procedure (Code): Union(x, y)

function find(x):

if (Parent(x) != x): return find(Parent(x))

else return x;

function union(x, y):

Parent[find(y) = find(x);

**Proof:** see Lectures\_GreedyAlgs-15\_16\_17.pdf Page 9

**Application: Path compression:**

* Changes the parent pointer for all nodes on the path to point to the new root

**Prim’s Algorithm [**[**LINK**](https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/)**] O(V2):**

starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way

* at every step of Prim’s algorithm, find a cut, pick the minimum weight edge from the cut, and include this vertex in MST Set (the set that contains already included vertices).

**Steps (Theoretical):**

1. Determine an arbitrary vertex as the starting vertex of the MST.
2. Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).
3. Find edges connecting any tree vertex with the fringe vertices.
4. Find the minimum among these edges.
5. Add the chosen edge to the MST if it does not form any cycle.
6. Return the MST and exit

**Implementation (Practical):**

* Create a set **mstSet** that keeps track of vertices already included in MST.
* Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign the key value as 0 for the first vertex so that it is picked first.
* While **mstSet** doesn’t include all vertices
  + Pick a vertex **u** that is not there in **mstSet**and has a minimum key value.
  + Include **u** in the **mstSet**.
  + Update the key value of all adjacent vertices of **u**. To update the key values, iterate through all adjacent vertices.
    - For every adjacent vertex **v**, if the weight of edge **u-v** is less than the previous key value of **v**, update the key value as the weight of **u-v**.

*Note: while this usually takes O(V2), the runtime* can *be reduced to O(E \* log(V)) using a binary heap.*

**Kruskal’s Algorithm [**[**LINK**](https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/)**]:**

**Implementation (Practical):**

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step #2 until there are (V-1) edges in the spanning tree.

**Time Complexity:** O(E \* log(E)) or O(E \* log(V))

*Reasoning:*

* Sorting of edges takes O(E \* log(E)) time.
* After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(log(V)) time.
* So overall complexity is O(E \* log(E) + E \* log(V)) time.
* The value of E can be at most O(V2), so O(log(V)) and O(log(E)) are the same. Therefore, the overall time complexity is O(E \* log(E)) or O(E\*log(V))

**Kruskal’s Algorithm (Yener):**

1. Insert all nodes into a priority queue PQ
2. While PQ is not empty:
   1. u = delete min(PQ)
   2. for x in neighbors(u):
      1. if (cos(x) > w(u, x))
         * cost(x) = w(u,x)
         * decrease key(x, cost(x))

**Heuristic Algorithms [**[**LINK**](https://en.wikipedia.org/wiki/Heuristic_(computer_science))**]:**

* Designed for solving problems faster
* Trade optimality, completeness, accuracy, or precision for speed
* Ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow

*Note: for more information on the trade-offs see* [*Heuristic (computer science) - Wikipedia*](https://en.wikipedia.org/wiki/Heuristic_(computer_science)#trade-off)

**Greedy Algo Application Example - Bit Compression:**

Using Huffman Encoding [[LINK](https://en.wikipedia.org/wiki/Huffman_coding)] O(n log(n)):

* Encodes the data based on the number of occurrences

*Note: Did you know there is a Tom Scott video for this?* [*LINK HERE*](https://youtu.be/JsTptu56GM8)

Steps to build a Huffman Tree:

1. Create a leaf node for each unique character and build a min-heap of all leaf nodes

* The value of the frequency is used as a positioner in queue

1. Extract two nodes with the min frequency from the min heap
2. Create a new node with a sum equal to that of the previous two node’s frequency. Make the first extracted node the left child and the second extracted node as the right child. Add this node to the min heap
3. Repeat steps 2 and 3 until the heap contains only one node. This is the root node.
4. Create an array arr. Traverse the tree formed starting from the root. While moving to the left child, write 0 to arr. While moving to the right child, write 1 to arr. Print the array when a leaf node is encountered
5. Replace every occurrence of the letter with the corresponding encoded value in arr

The cost of this tree is where is the occurrence frequency and is the bit length

*Note: you can prove this is one of the shortest ways to represent chars by contradiction* [LINK TO PROOF](https://inst.eecs.berkeley.edu/~cs170/fa20/assets/notes/huffman.pdf)

**Show when Greedy Algorithms fail:**

Example - Change in Bills:

* Go to a shop. Buy something. Say you have to pay 71 dollars for it. You give a cashier a 100. You want your change back in the least number of notes possible

1. You take the biggest note which is at most 29, so you take a 20-dollar note.
2. You need 9 more dollars. You take the biggest note that is not more than 9, so you take 5 dollar note.
3. You take the biggest note less than 4. So you take 2 dollar note.
4. You take the biggest note that is not more than 2. So you take 2 dollar note.

* Now suppose the only notes you have are 25, 14, 1
* if we take our change greedily we take 25, 1, 1, 1, 1. In total 5 notes.
* Note we could use fewer notes if we would take 14,14,1 however we were greedy and optimized only depending on the current status. Hence greedy is not always optimal.

General steps for developing a greedy algorithm:

1. Determine the optimal sub-structure (in a MST, it is a subset with the same nodes)
2. Develop a recursive solution (formulate)
3. prove that at every stage of recursion, one of the optimal choices is the greedy choice
4. show all but some of the sub-problems induced by making the greedy choice are empty????
5. Develop a recursive algorithm to implement the greedy strategy
6. Convert the recursive (set?) to an iterative one

Greedy choice: a global optimal choice can be obtained by making a locally optimal choice without considering the sub-problems resulting from our choice

## Begin Dynamic Programming:

LINKS:

* [Dynamic Programming - GeeksforGeeks](https://www.geeksforgeeks.org/dynamic-programming/)
* [Dynamic programming - Wikipedia](https://en.wikipedia.org/wiki/Dynamic_programming)
* [Introduction to Dynamic Programming - Data Structures and Algorithm Tutorials - GeeksforGeeks](https://www.geeksforgeeks.org/introduction-to-dynamic-programming-data-structures-and-algorithm-tutorials/)
* [The complete beginners guide to dynamic programming - Stack Overflow Blog](https://stackoverflow.blog/2022/01/31/the-complete-beginners-guide-to-dynamic-programming/)
* [Dynamic Programming: Examples, Common Problems, and Solutions](https://www.makeuseof.com/dynamic-programming-tutorial/)

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for the same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial

**How to tell if a problem should be solved using DP [**[**LINK1**](https://www.byte-by-byte.com/when-to-use-dp/)**]?**

1. Overlapping Subproblems:
   1. The same solution may be used repeatedly
   * Fibonacci contains the same numbers used repeatedly so it is useful to use DP
   * Binary search contains no common sub-problems, so it is not useful to use DP

If asked to show that a greedy algo does not work, provide a counter-example (see coin-change problem)

Applicative Example (Fibonacci Numbers):

int fib(int n):

If (n <= 1):

return n;

return fib(n-1) + fib(n-2);

Applicative Example (Weighted Scheduling Problem):

Case 1: if then opt(n) = opt(n-1)

Case 2: if n then P(n) we could simply delete each request that conflicted with request n

?????????????

Two types of DP [[LINK](https://www.geeksforgeeks.org/tabulation-vs-memoization/)]:

1. Tabulation
   1. Bottom-up approach
   2. Stores the results of subproblems in a table
   3. Iterative implementation
   4. Well-suited for problems with a large set of inputs
   5. Used when the subproblems do not overlap
2. Memoizatation
   1. Top-down approach
   2. Caches the results of function calls
   3. Recursive implementation
   4. Well-suited for problems with a relatively small set of inputs
   5. Used when the subproblems have overlapping subproblems

Tabulation vs Memoization Practical Example: Fibonacci:

Given lookup = new int[MAX\_SIZE];

Memoization Implementation:

int fib(int n):

if (lookup[n] == NIL) {

if (n <= 1)

lookup[n] = n;

else

lookup[n] = fib(n - 1) + fib(n - 2);

}

return lookup[n];

Tabulation implementation:

int fib(int n):

int f[n + 1];

int i;

f[0] = 0;

f[1] = 1;

for (i = 2; i <= n; i++)

f[i] = f[i - 1] + f[i - 2];

return f[n];

**Solution Methods (in more detail):**

Memoization (Top Down):

* The same as a recursive function, but looks up previously-calculated values in a table to avoid redundancy
  + If the solution is there, use it
  + If not, calculate it and store it for later use

Tabulation (Bottom Up):

* Store the numbers in the table as they are calculated to be used later

Let us define P(j) for an interval j as the largest index x i < j ????????

Look at the examples in the 3/27/23 notes (lecture 18)

**Largest Sum Contiguous Subarray (Kadane’s Algorithm) [**[**LINK**](https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/)**]:**

Given an array arr[] of size N. The task is to find the sum of the contiguous subarray within an arr[] with the largest sum.

* The subarray with a negative sum is discarded (*by assigning max\_ending\_here = 0 in code*).
* We carry subarray until it gives a positive sum.

Pseudo-code:

1. max\_so\_far = INT\_MIN
2. max\_ending\_here = 0
3. for arr**:**
   1. max\_ending\_here = max\_ending\_here + a[i]
   2. if (max\_so\_far < max\_ending\_here)

* max\_so\_far = max\_ending\_here
  1. if (max\_ending\_here < 0)
* max\_ending\_here = 0

1. return max\_so\_far

**Coin Change Problem:**

Dynamic Programming

**2D Dynamic Programming Problems:**

Solve sum problem:

Given n items and each has a given non-negative weight wi for i = 1 to n and give a bound N

**Find/Select** a subset S of these items S[i] that and is as large as possible or is as large as possible

Memoization:

## Example 1 - Fibonacci:

Draw it out: [*see image below - key:* fib M]

//recursive solution

const fibRec = (n) => {

if (n <= 2) return 1;

return fibRec(n-1) + fibRec(n-2);

}

//DP solution

S = {};

const fibIter = (n) => {

if (n in S) return S[n];

if (n <= 2) return 1;

S[n] = fibIter(n-1) + fibIter(n-2);

return S[n];

}

## Example 2 - Grid Traveller:

Given an m x n grid, in how many ways can you travel from the top left to the bottom right moving only down and right?

Drawing it out: [*see* *image below - key:* gridtraveler M]

//recursive solution

const gridTravelerRec = (n, m) => {

if (n == 1 && m == 1) return 1;

if (n == 0 || m == 0) return 0;

return gridTravelerRec(n - 1, m) + gridTravelerRec(n, m-1);

}

//DP solution

S = {};

const gridTravelerIter = (n, m) => {

if ((n,m) in S) return S[(n,m)];

if (n == 1 && m == 1) return 1;

if (n == 0 || m == 0) return 0;

S[(n,m)] = gridTravelerIter(n - 1, m) + gridTravelerIter(n, m-1);

return S[(n,m)];

}

## Memoization Recipe:

1. Make it work
   1. Visualize the problem as a tree
   2. Implement the tree using recursion
   3. Test it (use small values for time efficiency) - find substitute during exams?
2. Make it efficient
   1. Add/define a memo object (S)
      * S = {}
   2. Add a base case(s) to return memo values
      * If (n in s) return S[n];
   3. Add recursion base cases back
      * If (n < 0) return 0;
      * If (n == 1) return 1;
   4. Store return values in memo
      * S[n] = rec(n-1)
   5. Return the index in memo
      * return S[n]

## Example 3 - canSum:

Given a sum targetSum and an array numbers return a boolean indicating whether or not it is possible to generate targetSum using numbers from the array. You may reuse numbers. Assume all numbers are non-negative.

Drawing it out: [*see* *image below - key:* canSum M]

//Recusive solution

const canSumRec = (targetSum, numbers) => {

if (targetSum == 0) return true;

if (targetSum < 0) return false;

for (const i of numbers) {

if (canSumRec(targetSum - i, numbers)) return true;

}

return false;

}

//DP Solution

const S = {};

const canSumIter = (targetSum, numbers) => {

if (targetSum in S) return S[targetSum];

if (targetSum == 0) return true;

if (targetSum < 0) return false;

for (const i of numbers) {

const rem = targetSum - i;

S[i] = canSumIter(rem, numbers);

if (S[i]) return true;

}

S[targetSum] = false;

return S[targetSum];

}

## Example 4 - HowSum:

Given a sum targetSum and an array numbers return any combination that adds to targetSum using numbers from the array. You may reuse numbers. Assume all numbers are non-negative.

Drawing it out: [*see* *image below - key:* howSum M]

//Recursive solution

const howSumRec = (targetSum, numbers) => {

if (targetSum == 0) return [];

if (targetSum < 0) return null;

for (const i of numbers) {

const rem = targetSum - i;

const res = howSumRec(rem, numbers);

if (res != null) { res.push(i); return res; }

}

return null;

}

//DP solution

const S = {};

const howSumIter = (targetSum, numbers) => {

if (targetSum in S) return S[targetSum];

if (targetSum == 0) return [];

if (targetSum < 0) return null;

for (const i of numbers) {

const rem = targetSum - i;

S[i] = howSumIter(rem, numbers);

if (S[i] != null) { S[i].push(i); return S[i]; }

}

S[targetSum] = null;

return S[targetSum];

}

## Example 4 - BestSum:

Given a sum targetSum and an array numbers, return the smallest combination that adds to targetSum using numbers from the array. You may reuse numbers. Assume all numbers are non-negative.

Drawing it out: [*see* *image below - key:* bestSum M]

//Recursive solution

const bestSumRec = (targetSum, numbers) => {

if (targetSum == 0) return [];

if (targetSum < 0) return null;

var best = null;

for (const i of numbers) {

const rem = targetSum - i;

const res = bestSumRec(rem, numbers);

if (res != null) {

res.push(i);

if (best == null || res.length < best.length) best = res;

}

}

return best;

}

//DP solution

const S = {};

const bestSumIter = (targetSum, numbers) => {

if (targetSum in S) return S[targetSum];

if (targetSum === 0) return [];

if (targetSum < 0) return null;

let best = null;

for (const i of numbers) {

const rem = targetSum - i;

const remCombo = bestSumIter(rem, numbers);

if (remCombo != null) {

var newCombo = [...remCombo, i];

if (best == null || newCombo.length < best.length) {

best = newCombo;

}

}

}

S[targetSum] = best;

return best;

}

## Example 5 - CanConstruct:

Given a string targetString and an array of strings wordBank, return a boolean indicating whether or not targetString can be made by concatonating elements in wordBank. You may reuse words in wordBank.

Drawing it out: [*see* *image below - key:* CanConstruct M]

//Recursive Solution

const canConstructRec = (targetString, wordbank) => {

if (targetString == "") return true;

for (const i of wordbank) {

if (targetString.indexOf(i) == 0) {

const newStr = targetString.slice(i.length);

if (canConstructRec(newStr, wordbank) == true) {

return true;

}

}

}

return false;

}

//Recursive Solution

const S = {};

const canConstructIter = (targetString, wordbank) => {

if (targetString in S) return S[targetString];

if (targetString === "") return true;

for (const i of wordbank) {

if (targetString.indexOf(i) == 0) {

const newStr = targetString.slice(i.length);

if (canConstructIter(newStr, wordbank)) {

S[i] = true;

return true;

}

}

}

S[targetString] = false;

return S[targetString];

}

## Example 6 - CountConstruct

Given a string targetString and an array of strings wordBank, return the number of ways targetString can be made by concatenating elements in wordBank. You may reuse words in wordBank.

Drawing it out: [*see* *image below - key:* CountConstruct M]

//recursive solution

const countConstructRec = (targetWord, wordbank) => {

if (targetWord == '') return 1;

var c = 0;

for (const i of wordbank) {

if (targetWord.indexOf(i) == 0) {

const rem = targetWord.slice(i.length);

if (countConstructRec(rem, wordbank)) c++;

}

}

return c;

}

//DP solution

const S = {};

const countConstructIter = (targetWord, wordbank) => {

if (targetWord in S) return S[targetWord];

if (targetWord == '') return 1;

let c = 0;

for (const i of wordbank) {

if (targetWord.indexOf(i) == 0) {

const newWord = targetWord.slice(i.length);

const res = countConstructIter(newWord, wordbank);

if (res) {

if (S[i]) S[i]++;

else S[i] = 1;

c++;

} else S[i] = 0;

}

}

S[targetWord] = c;

return S[targetWord];

}

## Example 7 - AllConstruct:

Given a string targetString and an array of strings wordBank, return all ways targetString can be made by concatonating elements in wordBank. You may reuse words in wordBank.

Drawing it out: [*see* *image below - key:* allConstruct M]

//Recursive solution

const allConstructRec = (targetString, wordbank) => {

if (targetString === '') return [[]];

var sols = [];

for (const i of wordbank) {

if (targetString.indexOf(i) == 0) {

const newStr = targetString.slice(i.length);

const resp = allConstructRec(newStr, wordbank);

if (resp) {

const newPath = resp.map((path) => [i, ...path]);

sols.push(...newPath);

}

}

}

if (sols.length == 0) return null;

return sols;

}

//DP solution {time: n^m, space: m}

const S = {};

const allConstructIter = (targetString, wordbank) => {

if (targetString in S) return S[targetString];

if (targetString === '') return [[]];

var sols = [];

for (const i of wordbank) {

if (targetString.indexOf(i) == 0) {

const newStr = targetString.slice(i.length);

const resp = allConstructIter(newStr, wordbank);

if (resp) {

const newPath = resp.map((path) => [i, ...path]);

sols.push(...newPath);

S[i] = newPath;

}

}

}

if (sols.length == 0) { S[targetString] = null; return null; }

S[targetString] = sols;

return S[targetString];

}

Tabulation

## Example 1 - Fib:

//DP solution

const fibIter = (n) => {

const T = Array(n+1).fill(0);

T[1] = 1;

for (let i = 0; i <= n; i++) {

T[i+1] += T[i];

T[i+2] += T[i];

}

return T[n];

}

## Example 2 - Gridtraveler:

//DP solution

const gridTravellerIter = (m, n) => {

const T = Array(m+1)

.fill()

.map(() => Array(n+1).fill(0));

T[1][1] = 1;

for (let i = 0; i <= m; i++) {

for (let j = 0; j <= n; j++) {

if (i+1 <= m) T[i+1][j] += T[i][j];

if (j+1 <= n) T[i][j+1] += T[i][j];

}

}

return T[m][n];

}

## Tabulation Recipe

1. Visualize it as a table *with a size based on the inputs*
   1. Initialize table with default values (0 for int, false for bool, etc)
   2. Seed the trivial values into the table (T[1] = 1 for fib, T[1][1] = 1 for GridTraveller)
2. Iterate through the table
   1. Fill further positions based on the current position (T[i+1] += T[1], T[i+2] += T[i] for fib)

**Interval Scheduling:**

Given a list of requests in the format {start, end, weight} find the non-overlapping requests that have the largest total weight

[HW08.pdf](https://drive.google.com/file/d/1M90T_igno5Xy0XB4rUnAf5Z2y4yfUe6K/view?usp=share_link)

**Reduction:**

* NP stands for nondeterministic polynomial
* NP is the set of problems that you can **verify** in polynomial time
* Hard means it can not be solved in polynomial time
* Given a problem which we want to prove is hard if we can reduce a hard problem to our problem, we know our problem is hard.
* If we’re trying to prove Vertex coloring is hard using Set Color then we
  + Reduce Set to Vertex

**Linear Programming:**

TODO: COMPLETE THIS SECTION

**NP-REDUCTION**

## Yener Notes:

2-step approach

1. Show that Q belongs to NP
   1. We need to use a deterministic approach such as greedy to produce a candidate solution
   2. We need to check if the solution is valid in polynomial time
2. To show that it’s NP-hard mainly that it is at least as hard as a problem in NPC
   1. Reduction of a known problem P in NPC class to our candidate problem Q
   2. If Q has a polynomial-time solution, then P also has one and vice versa
      * But P does not admit a poly-time solution since it's already in NPC





**I’M MISSING A PAGE HERE**

Can we find some approximation solutions for the optimization version of problems in the NPC set?

* If opt is the optimal solution, then we are interested in finding an algorithm that gives us a solution SOL such that
  + The optimization of the maximizing function is <= 2 \* SOL and opt >= 2 \* SOL
  + 2 should be replaced by f(n)

DEFINITION:

* let C be the cost of a solution for a problem of size n found by an algorithm A.
* Let C^A be the optimal solution for that problem
* We say that algorithm A has an approximation ratio f(n)
* minimization problem
* maximization problem
* If A has f(n) approximation then A is called f(n)...approximation algorithm to the problem at hand
* If f(n) is something like (1 + ε) or (2 + ε) then we say A os a poly-time approximation algorithm with ratio (1 + ε)

**Constant Approximation Example - Vertex Cover:**

Given an undirected graph G = (V, E) is there a subset of the vertices V’ subset of V such that if there is an edge (u, v) ∈ E then either

* or
* or both

Find a subset of vertices V’ ⊆ V

*Note:*

* A ⊆ B “A is a subset of B” (everything in A is in B)
* A ⊂ B “A is a proper subset of B” everything in A is in B but

approx\_VC(G):

1. C ← ∅
2. E’ ← E[G] i.e. set of edges in the graph G
3. While E’ ≠ ∅
   1. Let (u,v) be an arbitrary edge of E’
   2. C ← C U {(u,v)}
   3. Remove from E all the edges that are ???????????

**ADD NOTES FOR PROOF HERE**

**Proof:**

1. Correctness
   1. It will loop until every edge in E is covered
2. Terminates?
   1. It will run in O(|E| + |V|) time

* We need to show that this solution is no more than twice the size of the optimal solution C\* → C <= 2 C\*

1. We will find a lower bound on the optimized solution by the algorithm above
   1. Let A be the set of edges chosen by the said algorithm
   2. Any VC must cover/include at least one endpoint of every edge in A
   3. No two edges in A share a vertex. From previous steps, we can say that in order to cover the edges in A, C\* must have at least as many vertices as the cardinality of A → |A| = k → |A\*| <= |C\*|
   4. Since each execution of the algorithm picks an edge for which neither edge? pivots in C **MISSING NOTES HERE**

**Theorem:** VC problem admits a 2-approximation algorithm

## Video 1 [[LINK](https://www.youtube.com/watch?v=O7pq43hIE_0)]

* P is in NP

**Poly-time reducibility**

## Video 2 [[LINK](https://www.youtube.com/watch?v=pK8VQd6U7BI)]

**Np-reduction from Subset Sum to Knapsack**

Definition of Subset-sum:

* Given a set of non-negative integers S, and a value sum k, determine if there is a subset of S with sum equal to k.

Definition of Knapsack

* Given a set of items, each with a weight and a value, determine which items to include in the collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

???????????

## Video 3 [[LINK](https://youtu.be/u5W32YxmnL8)]

**Independent Set:** within an undirected graph, the independent set is a set of nodes that have no edges between them.

The independent set problem asks: Given a set Gi = (Vi , Ei) does Gi have an independent set of size ki ?

**Clique Problem:**

Given ki , Gi = (Vc , Ec), does Gc have a clique of size kc ?

1. Let Gc = not{Gi} and kc = ki

* Gi has an independent set of size ki IFF Gc has a clique of size kc
* This is a reduction of IndSet to Clique
  1. Given an independent set, run a poly-time program to create an instance of clique such that