# Unit 1.1 – Optimization Models: 0/1 knapsack

## Part A – Overview for 6.00.2x

① Optimization Models ② Statistical Models ③ Simulation Models

## Part B – Optimization models overview

① What's optimization model?

- An objective function that is to be maximized or minimized, e.g.,
  - Minimize time spent traveling from New York to Boston
- A set of constraints (possibly empty) that must be honored, e.g.,
  - Cannot spend more than $100
  - Must be in Boston before 5:00PM


② a Greedy algorithm is often a practical approach to finding a pretty good approximate solution to an optimization problem.

③ Example: Knapsack and Bin-packing Problems
- 0/1 problem (discrete)
- Continuous or fractional knapsack problem

④ 0/1 knapsack problem, formalized

### 0/1 Knapsack Problem, Formalized

- Each item is represented by a pair, *<value, weight>*
- The knapsack can accommodate items with a total weight of no more than *w*
- A vector, *L*, of length *n*, represents the set of available items. Each element of the vector is an item
- A vector, *V*, of length *n*, is used to indicate whether or not items are taken. If *V[i] = 1*, item *I[i]* is taken. If *V[i] = 0*, item *I[i]* is not taken

Find a V that maximizes

$$\sum_{i=0}^{n-1} V[i] * I[i].value$$

subject to the constraint that

$$\sum_{i=0}^{n-1} V[i] * I[i].weight \leq w$$

⑤ Brute force algorithm – often NOT practical

0/1 knapsack problem is inherently exponential

### Brute Force Algorithm

- 1. Enumerate all possible combinations of items. That is to say, generate all subsets of the set of subjects. This is called the power set.
- 2. Remove all of the combinations whose total units exceeds the allowed weight.
- 3. From the remaining combinations choose any one whose value is the largest.

# Part C - 0/1 Knapsack "Greedy" Algorithm

① approximate solution - Greedy Algorithm

- while knapsack not full
  put "best" available item in knapsack
- But what does best mean?
  - Most valuable
  - Least expensive
  - Highest value/units

```
class Food(object):
    def __init__(self, n, v, w):
        self.name = n
        self.value = v
        self.calories = w

    def getValue(self):
        return self.value

    def getCost(self):
        return self.calories

    def density(self):
        return self.getValue()/self.getCost()

    def __str__(self):
        return self.name + ': <' + str(self.value)\
                + ', ' + str(self.calories) + '>'
```

```
def buildMenu(names, values, calories):
    """names, values, calories lists of same length.
       name a list of strings
       values and calories lists of numbers
       returns list of Foods"""
    menu = []
    for i in range(len(values)):
        menu.append(Food(names[i], values[i],
                         calories[i]))
    return menu
```

```
def greedy(items, maxCost, keyFunction):
    """Assumes items a list, maxCost >= 0,
         keyFunction maps elements of items to numbers"""
    itemsCopy = sorted(items, key = keyFunction,
                       reverse = True)
    result = []
    totalValue, totalCost = 0.0, 0.0

    for i in range(len(itemsCopy)):
        if (totalCost+itemsCopy[i].getCost()) <= maxCost:
            result.append(itemsCopy[i])
            totalCost += itemsCopy[i].getCost()
            totalValue += itemsCopy[i].getValue()

    return (result, totalValue)
```

```
def testGreedy(items, constraint, keyFunction):
    taken, val = greedy(items, constraint, keyFunction)
    print('Total value of items taken =', val)
    for item in taken:
        print('   ', item)
```

foods,

```
def testGreedys(maxUnits):
    print('Use greedy by value to allocate', maxUnits,
          'calories')
    testGreedy(foods, maxUnits, Food.getValue)
    print('\nUse greedy by cost to allocate', maxUnits,
          'calories')
    testGreedy(foods, maxUnits,
               lambda x: 1/Food.getCost(x))
    print('\nUse greedy by density to allocate', maxUnits,
          'calories')
    testGreedy(foods, maxUnits, Food.density)

testGreedys(800)
```

these obj. methods & lamda function can be used as "keyFunction".

② lambda - used to create anonymous functions

syntax: > lambda <var1, var2, ...> : <expression> ⇒ returns a function of n arguments

e.g. > f1 = lambda x, y : x+y

> f2 = lambda x,y : "factor" if (x%y==0) else "not factor"

Caution: do NOT use lambda for >1 lines functions because it's inefficient. Use def instead.

③ Pros and Cons of Greedy: 3.1 Easy to implement 3.2 Computationally efficient

3.3 NOT always yield the best solution & do NOT even know how good the approximation is

# Part D – Brute Force: Decision Tree (0/1 Knapsack)

① idea for optimal solution & search tree implementation (decision tree) "binary tree"

## Brute Force Algorithm

- 1. Enumerate all possible combinations of items. That is to say, generate all subsets of the set of subjects. This is called the power set. $O(2^n)$
- 2. Remove all of the combinations whose total units exceeds the allowed weight.
- 3. From the remaining combinations choose any one whose value is the largest.

## Search Tree Implementation

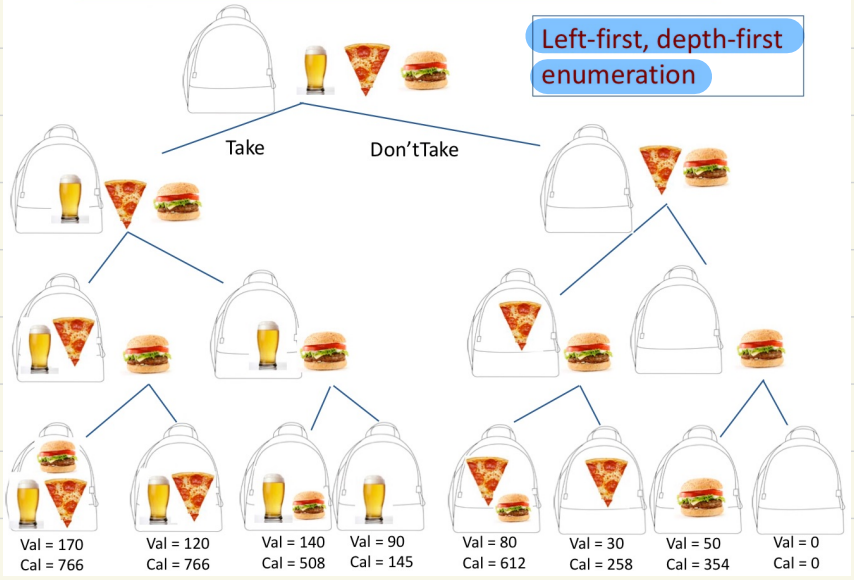
- The tree is built top down starting with the root
- The first element is selected from the still to be considered items
  - If there is room for that item in the knapsack, a node is constructed that reflects the consequence of choosing to take that item. By convention, we draw that as the left child
  - We also explore the consequences of not taking that item. This is the right child
- The process is then applied recursively to non-leaf children
- Finally, chose a node with the highest value that meets constraints

left-first, → depth-first method

② a search tree example

Remark: this a "odd #" menu, without any overlapping nodes

## A Search Tree Enumerates Possibilities




③ Computational Complexity

## Computational Complexity

- Time based on number of nodes generated
- Number of levels is number of items to choose from
- Number of nodes at level *i* is $2^i$
- So, if there are *n* items the number of nodes is
  - $\sum_{i=0}^{i=n} 2^i$
  - I.e., $O(2^{i+1})$
- An obvious optimization: don't explore parts of tree that violate constraint (e.g., too many calories)
  - Doesn't change complexity
- Does this mean that brute force is never useful?
  - Let's give it a try

④ Algorithm by decision tree implementation

## Header for Decision Tree Implementation

```
def maxVal(toConsider, avail):
    """Assumes toConsider a list of items,
               avail a weight
       Returns a tuple of the total value of a
           solution to 0/1 knapsack problem and
           the items of that solution"""
```

toConsider. Those items that nodes higher up in the tree (corresponding to earlier calls in the recursive call stack) have not yet considered

avail. The amount of space still available

## Body of maxVal (without comments)

```
if toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getUnits() > avail:
        result = maxVal(toConsider[1:], avail)
    else:
        nextItem = toConsider[0]
        withVal, withToTake = maxVal(toConsider[1:],
                                     avail - nextItem.getUnits())
        withVal += nextItem.getValue()
        withoutVal, withoutToTake = maxVal(toConsider[1:], avail)
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    return result
```

explore left branch

right bran. → (indent here)

if taken (choose left)

if not taken (choose right)

Does not actually build search tree

Local variable `result` records best solution found so far

```
def testMaxVal(foods, maxUnits, printItems = True):
    print('Use search tree to allocate', maxUnits,
          'calories')
    val, taken = maxVal(foods, maxUnits)
    print('Total value of items taken =', val)
    if printItems:
        for item in taken:
            print('   ', item)
```

⑤ the Algorithm slows down exponentially but can be rescued by Dynamic Programming

## Part E - Dynamic Programming (Recursive Fibonacci)

the name is somewhat deceptive

① Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler sub-problems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solutions to its sub-problems in a recursive manner.

(combinations with overlapping)

② below "recursive implementation of fibonacci" repeats large % of element and is terribly slow

```
def fib(n):
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)


fib(120) = 8,670,007,398,507,948,658,051,921
```

③ Key idea of DP - memorization

utilize "try: ... except xxx: ..."

- Trade a time for space ※
- Create a table to record what we've done
  - Before computing fib(x), check if value of fib(x) already stored in the table
    - If so, look it up
    - If not, compute it and then add it to table
  - Called memoization

## Using a Memo to Compute Fibonnaci

```
def fastFib(n, memo = {}):
    """Assumes n is an int >= 0, memo used only by
       recursive calls
       Returns Fibonacci of n"""
    if n == 0 or n == 1:
        return 1
    try:
        return memo[n]
    except KeyError:
        result = fastFib(n-1, memo) +\
                 fastFib(n-2, memo)
        memo[n] = result
        return result
```

※ this a nice idea

④ When does Dynamic Programming work?

e.g. Merge Sort

we will consider DP when the two conditions are met

## When Does It Work?

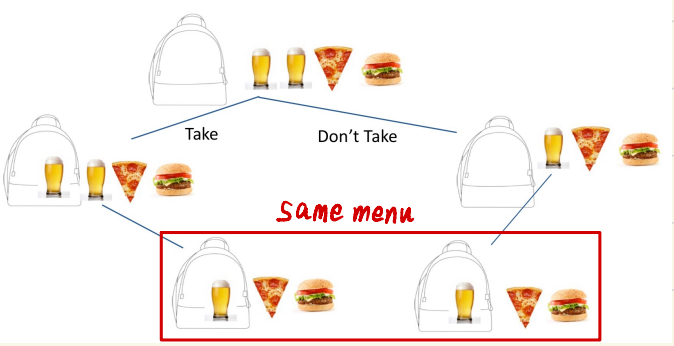
- Optimal substructure: a globally optimal solution can be found by combining optimal solutions to local subproblems
  - For x > 1, fib(x) = fib(x - 1) + fib(x – 2)
- Overlapping subproblems: finding an optimal solution involves solving the same problem multiple times
  - Compute fib(x) or many times

# Part F - Dynamic Programming: 0/1 Knapsack - trade time for space

① 0/1 Knapsack: Optimal substructure?

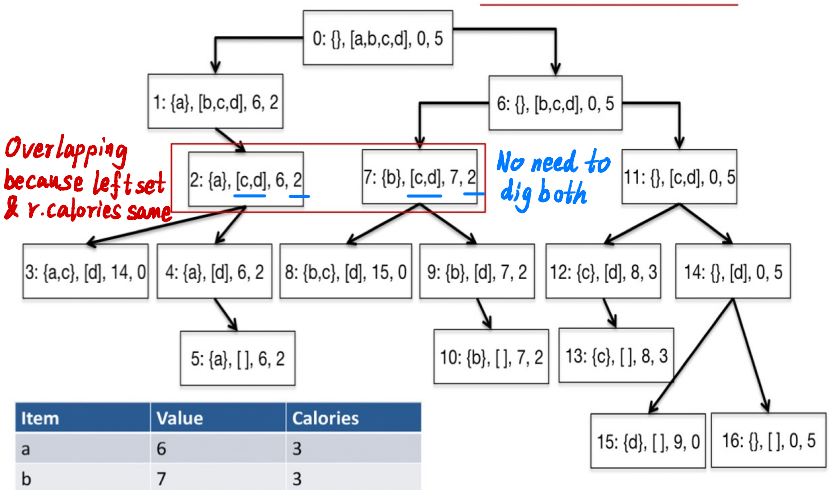
Overlapping subproblems?

## A Different Menu




## Search Tree

- Each node = <taken, left, value, remaining calories>




| Item | Value | Calories |
|---|---|---|
| a | 6 | 3 |
| b | 7 | 3 |
| c | 8 | 2 |
| d | 9 | 5 |

② from above right picture we see even partially same nodes can indicate same situation functioning as a parent; further digging both is no necessary (overlapping) because the substructures' optimal solutions are exactly the same.

③ add a "memo(dictionary)" parameter to keep track of necessary nodes info, and to trade time for space

## Modify maxVal to Use a Memo

- Add memo as a third argument
  - `def fastMaxVal(toConsider, avail, memo = {}):`
- Key of memo is a tuple
  - (items left to be considered, available weight)
  - Items left to be considered represented by `len(toConsider)` "trick" works because items are always removed from front of the list, so knowing len is enough to indicate which items are left. (so we separately manipulate the "left" set)
- First thing body of function does is check whether the optimal choice of items given the the available weight is already in the memo
- Last thing body of function does is update the memo

this works fine with many decision tree problem

automatically done within recursion

Remark: this 0/1 problem's decision tree is implemented with "from left to right order"

## Performance

| len(items) | 2**len(items) | Number of calls |
|---|---|---|
| 2 | 4 | 7 |
| 4 | 16 | 25 |
| 8 | 256 | 427 |
| 16 | 65,536 | 5,191 |
| 32 | 4,294,967,296 | 22,701 |
| 64 | 18,446,744,073,709,551,616 | 42,569 |
| 128 | Big | 83,319 |
| 256 | Really Big | 176,614 |
| 512 | Ridiculously big | 351,230 |
| 1024 | Absurdly big | 703,802 |

## How Can This Be?

- Problem is exponential
- Have we overturned the laws of the universe?
- Is dynamic programming a miracle?
- No, but computational complexity can be subtle
- Running time of `fastMaxVal` is governed by number of distinct pairs, `<toConsider, avail>`
  - Number of possible values of `toConsider` bounded by `len(items)` $\leq n$
  - Possible values of `avail` a bit harder to characterize
    - Bounded by number of distinct sums of weights $\leq 2^n$
  - Covered in more detail in assigned reading

## Summary of Lectures 1-2

- Many problems of practical importance can be formulated as optimization problems
- Greedy algorithms often provide adequate (though not necessarily optimal) solutions
- Finding an optimal solution is usually exponentially hard
- But dynamic programming often yields good performance for a subclass of optimization problems—those with optimal substructure and overlapping subproblems
  - Solution always correct
  - Fast under the right circumstances

However, DP can NOT change big O and worst case still there.

(consider cases with less overlapping – values are chosen from very large space & the knapsack can hold nearly all the items)

here DP falls into a complexity class called pseudo-polynomial. We need to constrain the behavior of inputs to ensure the good performance

```
import random

class Food(object):
    def __init__(self, n, v, w):
        self.name = n
        self.value = v
        self.calories = w
    def getValue(self):
        return self.value
    def getCost(self):
        return self.calories
    def density(self):
        return self.getValue()/self.getCost()
    def __str__(self):
    # print(<obj.>) shows following info instead of address
        return self.name + ': <' + str(self.value)\
               + ', ' + str(self.calories) + '>'


def buildLargeMenu(numItems, maxVal, maxCost):
    items = []  # list of Food objects
    # ordinary to see "maxXXX" in building random list of objects' functions
    for i in range(numItems):
        items.append(Food(str(i),
                          random.randint(1, maxVal),
                          random.randint(1, maxCost)))
    return items


def testMaxVal(foods, maxUnits, algorithm, printItems = True):
    # printItems flags whether to print or not
    print('Menu contains', len(foods), 'items')
    print('Use search tree to allocate', maxUnits,
          'calories')
    val, taken = algorithm(foods, maxUnits) # the algorithm must return the two
    if printItems:
        print('Total value of items taken =', val)
        for item in taken:
            print('   ', item)  # follow __str__(self) method
```

```
#this code keeps track of number of calls
def countingFastMaxVal(toConsider, avail, memo = {}):
    """Assumes toConsider a list of subjects, avail a weight
       memo supplied by recursive calls
      Returns a tuple of the total value of a solution to the
       0/1 knapsack problem and the subjects of that solution"""
    global numCalls # global avoids abandonment after local calling
    numCalls += 1   # should be defined in the "main"/"test" area

    # len(toConsider) stands for unique list of tail elements
    if (len(toConsider), avail) in memo:
        result = memo[(len(toConsider), avail)]
    elif toConsider == [] or avail == 0:
        result = (0, ())
    elif toConsider[0].getCost() > avail:
        #Explore right branch only
        result = countingFastMaxVal(toConsider[1:], avail, memo)
    else:
        nextItem = toConsider[0]
        #Explore left branch
        withVal, withToTake =\
                countingFastMaxVal(toConsider[1:],
                           avail - nextItem.getCost(), memo)
        withVal += nextItem.getValue()
        #Explore right branch
        withoutVal, withoutToTake = countingFastMaxVal(toConsider[1:],
                                        avail, memo)
        #Choose better branch
        if withVal > withoutVal:
            result = (withVal, withToTake + (nextItem,))
        else:
            result = (withoutVal, withoutToTake)
    memo[(len(toConsider), avail)] = result # update memo before return
    return result


for numItems in (2, 4, 8, 32, 64, 128, 256, 512, 1024):
# for numItems in (32,):
    numCalls = 0
    items = buildLargeMenu(numItems, 90, 250)
    testMaxVal(items, 750, countingFastMaxVal, True)
    print('Number of calls =', numCalls)
```

# Part G – Bitwise Operations

① Binary bitwise operations have inbuilt operators in Python

The Operators:

**x << y** ←
Returns x with the bits shifted to the left by y places (and new bits on the right-hand-side are zeros). This is the same as multiplying x by 2**y. = x * 2**y

**x >> y** →
Returns x with the bits shifted to the right by y places. This is the same as //'ing x by 2**y. = x // 2**y

**x & y** both 1 = 1
Does a "bitwise and". Each bit of the output is 1 if the corresponding bit of x AND of y is 1, otherwise it's 0.

**x | y** either 1 = 1
Does a "bitwise or". Each bit of the output is 0 if the corresponding bit of x AND of y is 0, otherwise it's 1.

**~ x** switch 0 & 1
Returns the complement of x - the number you get by switching each 1 for a 0 and each 0 for a 1. This is the same as -x - 1.

**x ^ y** exclusive or
Does a "bitwise exclusive or". Each bit of the output is the same as the corresponding bit in x if that bit in y is 0, and it's the complement of the bit in x if that bit in y is 1.

Just remember about that infinite series of 1 bits in a negative number, and these should all make sense.

② "n base" bitwise operations can be performed in the following way:

2.1 bits of x shifting left by y places: x * n**y ←

2.2 bits of y shifting right by y places: x // n**y →

③ For m items, if every item has n distinct status, then we consider utilising "n base" bitwise operation. However, such problems are usually inherently exponential ($n^m$)

e.g. 0312223111 0 ← 12 items, 4 distinct states ($4^{12}$)

④ There are $3^n$ possible combinations for n items when there are 2 bags, and each item can only be in bag1, bag2, or neither bag. Following is the code to generate combinations

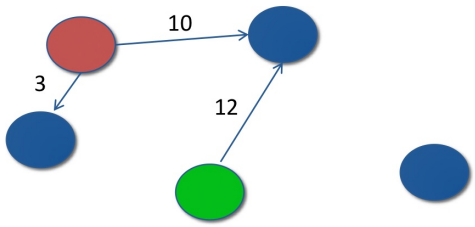
```
def yieldAllCombos(items):
    """
    Generates all combinations of N items into two bags, whereby each item
    is in one or zero bags.

    Yields a tuple, (bag1, bag2), where each bag is represented as a list of
    which item(s) are in each bag.
    """
    N = len(items)
    # Enumerate the 3**N possible combinations
    for i in range(3**N):
        bag1 = []
        bag2 = []
        for j in range(N):
            if (i // (3 ** j)) % 3 == 1:
                bag1.append(items[j])
            elif (i // (3 ** j)) % 3 == 2:
                bag2.append(items[j])
        yield (bag1, bag2)
```

# Unit 1.2 – Optimization Models: Graphs

## Part A – Graphs

① What: Set of nodes (vertices) + Set of edges (arcs)

- Set of nodes (vertices)
  - Might have properties associated with them
- Set of edges (arcs) each consisting of a pair of nodes
  - Undirected (graph)
  - Directed (digraph)
    - Source (parent) and destination (child) nodes
  - Unweighted or weighted



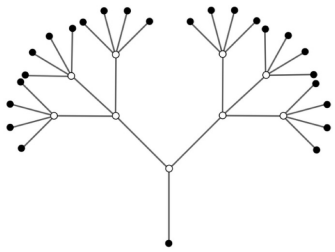

② "Tree" is an important special case graph & Common directed graph's representations

### Common Representations of Digraphs

- Adjacency matrix
  - Rows: source nodes
  - Columns: destination nodes
  - Cell[s, d] = 1 if there is an edge from s to d
    0 otherwise
- Adjacency list
  - Associate with each node a list of destination nodes

### Trees: An Important Special Case

- A directed graph in which each pair of nodes is connected by a single path
  - Recall the search trees we used to solve knapsack problem



③ Codes skeleton for directed & undirected graphs (OOP Adjacency List)

### Class Node

```
class Node(object):
    def __init__(self, name):
        """Assumes name is a string"""
        self.name = name
    def getName(self):
        return self.name
    def __str__(self):
        return self.name
```

### Class Edge

```
class Edge(object):
    def __init__(self, src, dest):
        """Assumes src and dest are nodes"""
        self.src = src
        self.dest = dest
    def getSource(self):
        return self.src
    def getDestination(self):
        return self.dest
    def __str__(self):
        return self.src.getName() + '->'\
               + self.dest.getName()
```

### Class Digraph, part 1

```
class Digraph(object):
    """edges is a dict mapping each node to a list of
    its children"""

    def __init__(self):
        self.edges = {}

    def addNode(self, node):
        if node in self.edges:
            raise ValueError('Duplicate node')
        else:
            self.edges[node] = []

    def addEdge(self, edge):
        src = edge.getSource()
        dest = edge.getDestination()
        if not (src in self.edges and dest in self.edges):
            raise ValueError('Node not in graph')
        self.edges[src].append(dest)
```

{<obj.> : [<obj 1>, <obj 2>, ...]} a dictionary mapping 1 node to all other that it connects

③ (Cont'd)

## ass Digraph, part 2

```
def childrenOf(self, node):
    return self.edges[node]

def hasNode(self, node):
    return node in self.edges

def getNode(self, name):
    for n in self.edges:
        if n.getName() == name:
            return n
    raise NameError(name)

def __str__(self):
    result = ''
    for src in self.edges:
        for dest in self.edges[src]:
            result = result + src.getName() + '->'\
                     + dest.getName() + '\n'
    return result[:-1] #omit final newline
```

## Class Graph

```
class Graph(Digraph):
    def addEdge(self, edge):
        Digraph.addEdge(self, edge)
        rev = Edge(edge.getDestination(), edge.getSource())
        Digraph.addEdge(self, rev)
```

- Why is Graph a subclass of digraph? graph extends digraph
- Remember the substitution rule from 6.00.1x?
  - If client code works correctly using an instance of the supertype, it should also work correctly when an instance of the subtype is substituted for the instance of the supertype
- Any program that works with a Digraph will also work with a Graph (but not *vice versa*)

④ Build a graph example:

## Build the Graph

```
def buildCityGraph():
    g = Digraph()
    for name in ('Boston', 'Providence', 'New York', 'Chicago',
                 'Denver', 'Phoenix', 'Los Angeles'): #Create 7 nodes
        g.addNode(Node(name))
    g.addEdge(Edge(g.getNode('Boston'), g.getNode('Providence')))
    g.addEdge(Edge(g.getNode('Boston'), g.getNode('New York')))
    g.addEdge(Edge(g.getNode('Providence'), g.getNode('Boston')))
    g.addEdge(Edge(g.getNode('Providence'), g.getNode('New York')))
    g.addEdge(Edge(g.getNode('New York'), g.getNode('Chicago')))
    g.addEdge(Edge(g.getNode('Chicago'), g.getNode('Denver')))
    g.addEdge(Edge(g.getNode('Denver'), g.getNode('Phoenix')))
    g.addEdge(Edge(g.getNode('Denver'), g.getNode('New York')))
    g.addEdge(Edge(g.getNode('Chicago'), g.getNode('Phoenix')))
    g.addEdge(Edge(g.getNode('Los Angeles'), g.getNode('Boston')))
```

# Part B – Graph Problems

① Common Graphs Problems:

## A Classic Graph Optimization Problem

1-1 Shortest path from n1 to n2
- Shortest sequence of edges such that
  - Source node of first edge is n1
  - Destination of last edge is n2
  - For edges, e1 and e2, in the sequence, if e2 follows e1 in the sequence, the source of e2 is the destination of e1

1-2 Shortest weighted path
- Minimize the sum of the weights of the edges in the path

② 2 algorithms:
- Depth-first search (DFS) for problems 1-1 & 1-2
- Breadth-first search (BFS) for problem 1-1 only

## ③ Depth-first Search (DFS)

for "shortest path from n1 to n2"

(this is the recursive version)

## Finding the Shortest Path

- Algorithm 1, depth-first search (DFS)
- Similar to left-first depth-first method of enumerating a search tree (Lecture 2)
- Main difference is that graph might have cycles, so we must keep track of what nodes we have visited

※ these 3 parameters must be in the same graph

## Depth First Search (DFS)

```
def DFS(graph, start, end, path, shortest):
    path = path + [start]
    if start == end:
        return path
    for node in graph.childrenOf(start):
        if node not in path: #avoid cycles
            if shortest == None or len(path) < len(shortest):
                newPath = DFS(graph, node, end, path,
                              shortest, toPrint)
                if newPath != None:
                    shortest = newPath
    return shortest

def shortestPath(graph, start, end):
    return DFS(graph, start, end, [], None, toPrint)
```

this will assign "path" to a new address list, leaving the ori. unchanged

← next linked nodes

DFS called from a wrapper function: shortestPath

Gets recursion started properly

Provides appropriate abstraction

## Test DFS

```
def testSP(source, destination):
    g = buildGraph()
    sp = shortestPath(g, g.getNode(source), g.getNode(destination)
    if sp != None:
        print('Shortest path from', source, 'to',
              destination, 'is', printPath(sp))
    else:
        print('There is no path from', source, 'to', destination)

testSP('Boston', 'Chicago')
```

(recursion actually "starts" from the deepest level)

Remark: avoid changing the original list because there may be useful aliasing variable(s) pointing to that original list, unless I'm sure NO aliasing issue. Recursion extremely relies on aliasing.

In this algorithm, changing to both "> path.append(start)" and "> path += [start]" NOT correct because those two both modifies the original list directly.

So always code in the format "> a = a+b" if no time complexity issue for mutable data.

## ④ Breadth-First Search (BFS) for "shortest path from n1 to n2" (quicker method)

(non-recursive method!)

these 3 parameters must be in the same graph!

idea: explore all paths with n hops before exploring any path with more than n hops.

## Algorithm 2: Breadth-first Search (BFS)

```
def BFS(graph, start, end, toPrint = False):
    initPath = [start]
    pathQueue = [initPath]
    if toPrint:
        print('Current BFS path:', printPath(pathQueue))
    while len(pathQueue) != 0:
        #Get and remove oldest element in pathQueue
        tmpPath = pathQueue.pop(0)
        print('Current BFS path:', printPath(tmpPath))
        lastNode = tmpPath[-1]
        if lastNode == end:
            return tmpPath
        for nextNode in graph.childrenOf(lastNode):
            if nextNode not in tmpPath:
                newPath = tmpPath + [nextNode]
                pathQueue.append(newPath)
    return None
```

a list of lists

the only concrete? return (the result)

Explore all paths with n hops before exploring any path with more than n hops

## ⑤ for "Weighted Shortest Path", DFS works but BFS does NOT work.

- Want to minimize the sum of the weights of the edges, not the number of edges
- DFS can be easily modified to do this
- BFS cannot, since shortest weighted path may have more than the minimum number of hops

## ⑥ a real life example involving Graphs (Euler's Circuit): Students in a Line

### 6.1 problem modeling

Students in a line

Second graders are lining up to go to their next class, but must be ordered alphabetically before they can leave. The teacher only swaps the positions of two students that are next to each other in line.

If we want to represent this situation as a graph, which variables should be represented as edges and vertices?

- (●) A) Vertices represent permutations of the students in line. Edges connect two permutations if one can be made into the other by swapping two adjacent students. ✔
- ( ) B) Vertices represent students. Edges connect two students if they are next to each other in line.
- ( ) C) Vertices represent permutations of the students, and each edge represents an individual student. An edge connects two vertices if that student is involved in swap between the two permutations.

### 6.2 properties of the graph

3. For questions 3 and 4, consider the general case of our previous problem (permutations of *n* students in a line). Give your answer in terms of *n*.

When represented as a tree, each node will have how many children?

n-1

$n-1$

**Answer:** n - 1

**Explanation:**

In any given permutation, *n* students are lined up. Since one may only swap the positions of two adjacent students, there are exactly $n-1$ pairs we are able to swap. Each of these swaps will create a distinct ordering, so there are exactly $n-1$ childern of each node.

# of edges away

4. Given two permutations, what is the maximum number of swaps it will take to reach one from the other?

n*(n-1)/2

$\frac{n\cdot(n-1)}{2}$

remember the graph is a Euler's Circuit and can go through 2 directions at any node

**Answer:** n * (n - 1)/2

**Explanation:**

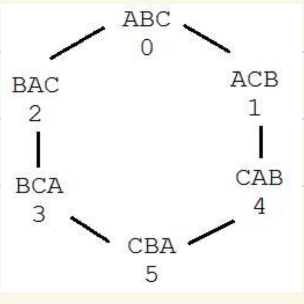
Answer: $n \cdot \frac{n-1}{2}$

worst case

Consider the case where the two permutations whose exchange would take the maximum number of swaps. Clearly these are two whose orders are opposite. It takes $n-1$ swaps to move the last person in line to the first position. This leaves the rest of the line's old order intact.

Next it takes $n-2$ swaps to move the last person in line to the second position. We continue until only one more swap is needed (switching the last two people in line). This takes $(n-1)+(n-2)+\ldots+2+1 = n\cdot\frac{n-1}{2}$ swaps.

### 6.3 a 3-students' visualization




$\frac{3\times(3-1)}{2}$ = 3 edges maximum

# 6.4 Further DFS algorithm investigation through Euler Circuit like "students in a line problem"

We saw before that for permutations of 3 people in line, any two nodes are at most three edges, or four nodes, away. But DFS has yielded paths longer than three edges! In this graph, given a random source and a random destination, what is the probability of DFS finding a path of the shortest possible length?

but eventually return the shortest path correctly

2/3 (1/3 chances extend to impractical length, i.e. > 3)

**Answer:** 2/3

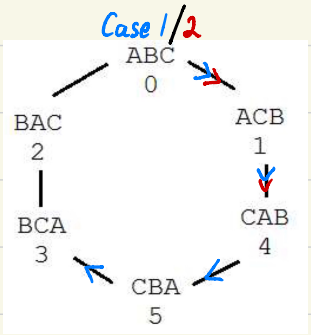
$\frac{2}{3}$

**Explanation:**
First, realize that the structure of this graph is a set of six nodes, all connected in a circle. Each node has two edges that connect it to adjacent nodes.
Given any node, we know that DFS will prioritize the lower-numbered neighbor. Thus, for any destination, we first check for paths along this side. If our destination is our source, we terminate the DFS, and return a path of length zero, which is clearly the shortest. Otherwise, we continue in a circle in one direction. We cannot change direction once we have begun to traverse the circle, as the path may not include any node more than once. It will have found the shortest path for the nodes that are 0, 1, 2, or 3 edges away, but will yield paths of length 4 and 5 for the last two nodes that are, in reality, 2 and 1 edges away, respectively. As it has found the shortest path for 4 nodes, but not for 2, the probability is 4 in 6, or 2/3.

For large # of people in line, there are slightly > 50% chances DFS will never extend its search to impractical length (but will still go through the head parts of those paths)

Remark: DFS algorithm will always explore all possible paths up to the length "shortest" currently stores.

for example (assume lower no. first & no head backward):




Case 1. DFS fully go through impractical length path (because the impractical path searched first)
testSP("ABC","BCA")

Current DFS path: ABC
Current DFS path: ABC->ACB
Current DFS path: ABC->ACB->CAB
Current DFS path: ABC->ACB->CAB->CBA
Current DFS path: ABC->BAC
Current DFS path: ABC->BAC->BCA
Shortest path from ABC to BCA is ABC->BAC->BCA

==========

Case 2. DFS partially go through impractical length path (only the head part due to shortest searched first)
testSP("ABC","CAB")

Current DFS path: ABC
Current DFS path: ABC->ACB
Current DFS path: ABC->ACB->CAB
Current DFS path: ABC->BAC
Current DFS path: ABC->BAC->BCA
Shortest path from ABC to CAB is ABC->ACB->CAB

# ⑦ Time Complexity (big-O) of DFS in complete graph with n nodes: K(n)

In the following examples, assume all graphs are undirected. That is, an edge from A to B is the same as an edge from B to A and counts as exactly one edge.

A **clique** is an unweighted graph where each node connects to all other nodes. We denote the clique with $n$ nodes as **KN**. Answer the following questions in terms of $n$.

1. How many edges are in **KN**?

Answer: $n \cdot \frac{(n-1)}{2}$

In a directed graph, each node would connect to all other nodes, yielding $n \cdot (n-1)$ edges. In our undirected graph, an edge from A to B and from B to A are the same edge, so there are, in fact, half as many.
Alternatively - if you are familiar with the binomial coefficient - see that for each edge, you must choose two nodes to connect. Thus there are

$$\binom{n}{2} = n \cdot \frac{(n-1)}{2}$$

edges.

2. Consider the new version of DFS. This traverses paths until all non-circular paths from the source to the destination have been found, and returns the shortest one.

Let A be the source node, and B be the destination in **KN**. How many paths of length 2 exist from A to B?

n-2

**Answer:** n - 2

## ⑦' (Cont'd)

3. How many paths of length 3 exist from A to B?

(n-2)*(n-3)

**Answer:** (n - 2) * (n - 3)

$(n-2)\cdot(n-3)$

**Explanation:**
Answer: $(n-2)\cdot(n-3)$
Use the same reasoning as used for the previous problem. After knowing our source and destination, we must travel through 2 additional nodes without touching any node twice. For the first node, we have $n-2$ choices, and for the second, we have $n-3$ choices.
Note that this is equivalent to $\frac{(n-2)!}{(n-4)!}$

4. Continuing the logic used above, calculate the number of paths of length $m$ from A to B, where $1 \leq m \leq (n-1)$, and write this number as a ratio of factorials.

To indicate a factorial, please enter `fact(n)` to mean $n!$; `fact(n+2)` to mean $(n+2)!$, etc.

**Explanation:**
Answer: $\frac{(n-2)!}{(n-m-1)!}$
Following the previous problems, it is clear that in choosing our first node between A and B, we have $(n-2)$ choices. Similarly, in choosing the second, we have $(n-3)$ choices.
In fact, in choosing the $j$th node, we have $(n-j-1)$ choices. Taking the product from $j$ = 1 to $m$ - 1 (since $m$ - 1 nodes exist between A and B in a path of length $m$), we get $\frac{(n-2)!}{(n-m-1)!}$

DFS for complete graph has a factorial big-O, which is even worse than exponential.

5. Using the fact that for any $n$, $\frac{1}{0!}+\frac{1}{1!}+\frac{1}{2!}+\ldots+\frac{1}{n!} \leq e$ for all $n$, where $e$ is some constant, determine the asymptotic bound on the number of paths explored by DFS. For simplicity, write $O(n)$ as just n, $O(n^2)$ as n^2, etc.

**Explanation:**
Answer: $O((n-2)!)$
Note that DFS will traverse every path from A to B. To calculate the number of paths, we must sum the paths of every length (from 1 to $n-1$). This sum can be written as:

$$\frac{(n-2)!}{(n-2)!}+\frac{(n-2)!}{(n-3)!}+\frac{(n-2)!}{(n-4)!}+\ldots+\frac{(n-2)!}{0!}$$

This is equal to $(n-2)!\cdot\left(\frac{1}{0!}+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\ldots+\frac{1}{(n-2)!}\right)$.
Since $\left(\frac{1}{0!}+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\ldots+\frac{1}{(n-2)!}\right) \leq e$, which is a constant, the number of paths is $O((n-2)!)$.

## ⑧ DFS vs. BFS for unweighted graphs

A **clique** is an unweighted graph where each node connects to all other nodes. We denote the clique with $n$ nodes as **KN**. Answer the following questions in terms of $n$.

1. What is the asymptotic worst-case runtime of a Breadth First Search on KN? For simplicity, write $O(n)$ as just n, $O(n^2)$ as n^2, etc.

n-1

**Answer:** n

$n-1$

**Explanation:**
Answer: $O(n)$
BFS begins by checking all the paths of length 1. In its worst case, it must check the paths to every node from the source to find the destination. This is at most, $n-1$ checks.

2. BFS will always run faster than DFS.

- ○ True
- ◉ False ✔ ✔

**Explanation:**
Consider a graph of two nodes, A and B, connected by an edge. You wish to search for a path from A to B. As there is exactly one edge in the graph, and exactly one path from A to B, both run in an equal number of steps.

## ⑧' (Cont'd) DFS vs. BFS

5. Regardless of node priority, BFS will always run at least as fast as Shortest Path DFS on two nodes in any connected unweighted graph.

- (●) True ✔
- ( ) False

**Explanation:**
Shortest Path DFS must always explore every path from the source to the destination to ensure that it has found the shortest path. Once BFS has found a path, it knows that it is the shortest, and does not have to explore any other paths.

## ⑨ a taste of weighted graph: students in a line with extra conditions

Consider once again our permutations of students in a line. Recall the nodes in the graph represent permutations, and that the edges represent swaps of adjacent students. We want to design a weighted graph, weighting edges higher for moves that are harder to make. Which of these could be easily implemented by simply assigning weights to the edges already in the graph?

- [x] A) A large student who is difficult to move around in line. ✔
- [x] B) A sticky spot on the floor which is difficult to move onto and off of. ✔
- [ ] C) A student who resists movement to the back of the line, but accepts movement toward the front.

↑ can NOT be accomplished with undirected graph algorithms

**Explanation:**
Answer: A, B
A) is easily implemented by weighting heavily all edges that involve moving the particular student.
B) is implemented by increasing the weight of all edges that involve a swap with that spot in line.
C) cannot be done without weighting two directions of an edge differently. In this case, putting one student behind another is not the same as putting the first student in front of the other, but in our undirected graph, it is.

## Code of an extra class: (inheritance)

Write a `WeightedEdge` class that extends `Edge`. Its constructor requires a weight parameter, as well as the parameters from Edge. You should additionally include a `getWeight` method. The string value of a `WeightedEdge` from node A to B with a weight of 3 should be "A→B (3)".

```
1 class WeightedEdge(Edge):
2     def __init__(self, src, dest, weight):
3         Edge.__init__(self, src, dest)
4         self.weight = weight
5     def getWeight(self):
6         return self.weight
7     def __str__(self):
8         return Edge.__str__(self) + " (" + str(self.weight) + ")"
9
```

/dəikstrə/

⑩ Dijkstra's algorithm is a general method to find the shortest distances from a node to all other nodes in a graph (weighted/unweighted).

Breadth-first search (BFS) can be viewed as a special case of dijkstra's algorithm on unweighted graphs, where the priority queue degenerates into a FIFO queue.

The A* algorithm is a generalization of Dijkstra's Algorithm that cuts down the size of the subgraph that must be explored, if additional is available that provides a lower bound on the "distance" to the target. This approach can be viewed from the perspective of linear programming & related to solutions to its dual linear program.