Programming, Data Structures and Algorithms using Python

Prof. Madhavan Mukund
Director
Chennai Mathematical Institute

Mr. Omkar Joshi
Course Instructor
IITM Online Degree Programme

Programming, Data Structures and Algorithms using Python

Summary of weeks 1 to 3

Content

- Timing our code
- Analysing an algorithm
- Asymptotic notations
- Orders of magnitude
- Calculating complexity
- Searching algorithms
- Selection sort
- Insertion sort
- Merge sort
- Quicksort
- Comparing sorting algorithms
- Lists vs. Arrays
- Implementation of lists in Python
- Implementation of dictionaries in Python

Timing our code

```
import time
start = time.perf_counter()
…
# Execute some code
…
end = time.perf_counter()
time_elapsed = end – start
```

Analysing an algorithm

- Two parameters to measure an algorithm
  - Running time (time complexity)
  - Memory requirement (space complexity)
- Running time $T(n)$ is a function of input size n
- Upper bound on worst case gives us an overall guarantee on performance

Asymptotic notations

- Big O notation:

  f(n) is O(g(n)) means g(n) is an upper bound for f(n)

- Ω notation:

  f(n) is Ω(g(n)) means g(n) is a lower bound for f(n)

- Θ notation:

  f(n) is Θ(g(n)) means we have found matching upper and lower bounds (optimal solution)

Orders of magnitude

- Commonly encountered classes of functions
- In each case c is a positive constant and n increases without bound
- The slower-growing functions are listed first

Notation	Name
$O(c)$	Constant
$O(\log\log n)$	Double logarithmic
$O(\log n)$	Logarithmic
$O((\log n)^c), c > 1$	Polylogarithmic
$O(n^c), 0 < c < 1$	Fractional power
$O(n)$	Linear
$O(\mathrm{n}\log n)$	Loglinear
$O(n^2)$	Quadratic
$O(n^c)$	Polynomial
$O(c^n), c > 1$	Exponential
$O(n!)$	Factorial

Calculating complexity

- Depends on the type of algorithm
- Iterative programs
  - Focus on loops
- Recursive programs
  - Write and solve a recurrence relation

Searching algorithms

Linear search

- Naïve solution
- Check every element in the list one by one
- Worst case is when element is not present in the list

Best case	Average case	Worst case
$O(1)$	$O(n)$	$O(n)$

Binary search

- Prerequisite: list must be sorted
- Compare with midpoint element
- Halve the list till interval becomes empty
- Recurrence: T(n) = T(n / 2) + 1

Best case	Average case	Worst case
$O(1)$	$O(\log n)$	$O(\log n)$

Selection sort

- It is an intuitive algorithm
- Repeatedly find the minimum/maximum element and append it to the sorted list
- Swapping elements helps us avoid use of second list
- Number of comparisons: $T(n) = n + (n - 1) + \ldots + 1$

  $= n(n + 1) / 2$
- The time complexity of Selection sort remains the same irrespective of the sequence of elements

Insertion sort

- It is another intuitive algorithm
- Repeatedly pick an element and insert it into the sorted list
- Number of comparisons: $T(n) = n + (n - 1) + \ldots + 1$

  $= n(n + 1) / 2$
- The time complexity of Insertion sort varies based on the sequence of elements

Merge sort

- Divide the list into two halves
- Separately sort the left and right half
- Combine the two sorted lists A and B to get a fully sorted list C
  - If A is empty, copy B into C
  - If B is empty, copy A into C
  - Otherwise, compare first elements of A and B
  - Move the smaller of the two to C
  - Repeat till you exhaust A and B
- Recurrence: $T(n) = 2T(n / 2) + n$

Quicksort

- Choose a pivot element (typically the first element)
- Partition the list into lower and upper parts with respect to the pivot
- Move the pivot between the lower and upper partition
- Recursively sort the two partitions
- This allows an in-place sort
- Iterative implementation is possible to avoid the cost of recursive calls

Comparing sorting algorithms

Parameter	Selection sort	Insertion sort	Merge sort	Quicksort
Best case	$O(n^2)$	$O(n)$	$O(\text{n} \log n)$	$O(\text{n} \log n)$
Average case	$O(n^2)$	$O(n^2)$	$O(\text{n} \log n)$	$O(\text{n} \log n)$
Worst case	$O(n^2)$	$O(n^2)$	$O(\text{n} \log n)$	$O(n^2)$
In-place	Yes	Yes	No	Yes
Stable	No	Yes	Yes	No

Lists vs. Arrays

Lists	**Arrays**
Flexible length	Fixed size
Values are scattered in memory	Allocate a contiguous block of memory
Need to follow links to access	Random access
Insertion and deletion is easy	Insertion and deletion is expensive
Swapping elements takes constant time	Swapping elements takes linear time

Implementation of lists in Python

- Python lists are NOT implemented as flexible linked lists
- Underlying interpretation maps the list to an array
  - Assign a fixed block when you create a list
  - Double the size if the list overflows the array
- Keep track of the last position of the list in the array
  - l.append() and l.pop() are constant time, amortized – $O(1)$
  - Insertion/deletion require time $O(n)$
- Effectively, Python lists behave more like arrays than lists

Implementation of dictionaries in Python

- A dictionary is implemented as a hash table
  - An array plus a hash function
- Creating a good hash function is important (and hard!)
- Need a strategy to deal with collisions
  - Open addressing/closed hashing – probe for free space in the array
  - Open hashing – each slot in the hash table points to a list of key-value pairs

Programming, Data Structures and Algorithms using Python

Prof. Madhavan Mukund
Director
Chennai Mathematical Institute

Mr. Omkar Joshi
Course Instructor
IITM Online Degree Programme

Programming, Data Structures and Algorithms using Python

Summary of weeks 4 to 6

Content

- Graphs
- Graph representation
- Breadth First Search(BFS)
- Depth First Search(DFS)
- Applications of BFS & DFS
- Directed Acyclic Graphs(DAGs)
- Shortest Paths in Weighted Graphs
- Dijkstra’s algorithm
- Bellman-Ford Algorithm
- Floyd-Warshall algorithm
- Trees
- Spanning trees
- Prim’s algorithm
- Kruskal’s algorithm
- Efficient data structures

Graphs

- A graph represents relationships between entities
  - Entities are vertices/nodes
  - Relationships are edges
- A graph can be directed or undirected
  - A is a parent of B – directed
  - A is a friend of B – undirected
- Paths are sequence of connected edges
- Reachability: is there a path from node u to node v?

Graph representation

- G = (V, E)
  - $|V| = n$
  - $|E| = m$
  - If G is connected, m can vary from $n - 1$ to $n(n - 1) / 2$
- Adjacency matrix
  - $n \times n$ matrix, AMat[i, j] = 1 iff $(i, j) \in E$
- Adjacency list
  - Dictionary of lists
  - For each vertex i, AList[i] is the list of neighbors of i

Breadth First Search(BFS)

- Breadth first search is a systematic strategy to explore a graph, level by level
- Maintain visited but unexplored vertices in a queue
- Complexity is $O(n^2)$ using adjacency matrix, $O(m + n)$ using adjacency list
- Add parent information to recover the path to each reachable vertex
- Maintain level information to record length of the shortest path, in terms of number of edges

Depth First Search(DFS)

- DFS is another systematic strategy to explore a graph
- DFS uses a stack to suspend exploration and move to unexplored neighbors
- Complexity is $O(n^2)$ using adjacency matrix, $O(m+n)$ using adjacency list
- Useful features can be found by recording the order in which DFS visits vertices

Applications of BFS & DFS

- Paths discovered by BFS are shortest paths in terms of number of edges
- BFS and DFS can be used to identify connected components in an undirected graph
- BFS and DFS identify an underlying tree
- Use of DFS numbering
  - Strongly connected components
  - Articulation points(cut vertices) and bridges(cut edges)

Directed Acyclic Graphs(DAGs)

- Directed acyclic graphs are a natural way to represent dependencies
- Topological sorting
  - It gives a feasible schedule that represents dependencies
  - Complexity is $O(n^2)$ using adjacency matrix, $O(m+n)$ using adjacency list
- Longest paths
  - Directed acyclic graphs are a natural way to represent dependencies
  - Complexity is $O(m+n)$

Shortest Paths in Weighted Graphs

- Single source shortest paths (Dijkstra's algorithm)
  - Find shortest paths from a fixed vertex to every other vertex
- All pairs shortest paths (Floyd-Warshall algorithm)
  - Find shortest paths between every pair of vertices i and j
- Negative edge weights and Negative cycles
  - If a graph has a negative cycle, shortest paths are not defined
  - Without negative cycles, we can compute shortest paths even if some weights are negative (Bellman-Ford Algorithm)

Dijkstra’s algorithm

- Dijkstra’s algorithm computes single source shortest paths
- Uses a greedy strategy to identify vertices to visit
  - Next vertex to visit is based on shortest distance computed so far
  - Correctness requires edge weights to be non-negative
- Complexity is $O(n^2)$
  - Even with adjacency lists
  - Bottleneck is identifying unvisited vertex with minimum distance
- Complexity can be improved to $O((m+n)\log n)$ by using efficient min-heap data structure

Bellman-Ford Algorithm

- Bellman-Ford algorithm computes single source shortest paths with negative weights
- Dijkstra’s algorithm assumes non-negative edge weights
  - Final distance is frozen each time a vertex “burns”
  - Should not encounter a shorter route discovered later
  - Without negative cycles, every shortest route is a path
  - Every prefix of a shortest path is also a shortest path
- Iteratively find shortest paths of length 1, 2, …, n – 1
- Update distance to each vertex with every iteration
- Complexity is $O(n^3)$ using adjacency matrix, $O(mn)$ using adjacency list

Floyd-Warshall algorithm

- Floyd-Warshall algorithm computes all pairs shortest paths
- Complexity using simple nested loop implementation is $O(n^3)$

Trees

- A tree is a connected acyclic graph
- A tree with n vertices has exactly $n - 1$ edges
- Adding an edge to a tree creates a cycle
- Deleting an edge from a tree splits the tree
- In a tree, every pair of vertices is connected by a unique path

Spanning trees

- Retain a minimal set of edges so that graph remains connected
- A graph can have multiple spanning trees
- Minimum Cost Spanning Tree (MCST) – among the different spanning trees, choose one with minimum cost
- A graph can have multiple MCSTs, but the cost will always be unique
- Building a MCST
  - Prim’s algorithm
  - Kruskal’s algorithm

Prim’s algorithm

- Start with a smallest weight edge overall
- Incrementally grow the MCST from any vertex
- Extend the current tree by adding the smallest edge from some vertex in the tree to a vertex not yet in the tree
- Implementation is similar to Dijkstra’s algorithm
- Complexity is $O(n^2)$
  - Even with adjacency lists
  - Bottleneck is identifying unvisited vertex with minimum distance
- Complexity can be improved to $O((m+n)\log n)$ by using efficient min-heap data structure

Kruskal’s algorithm

- Start with n components, each an isolated vertex
- Process edges in ascending order of cost
- Include edge if it does not create a cycle
  - Challenge is to keep track of connected components
  - Maintain a dictionary to record component of each vertex
  - Initially each vertex is an isolated component
  - When we add an edge (u, v), merge the components of u and v
- Complexity is $O(n^2)$ due to naive handling of components, can be improved to $O(m \log n)$ by using efficient union-find data structure

Efficient data structures

1. Union-Find
   - Across m operations, amortized complexity of each Union() operation is $\log m$
   - With clever updates to the tree, Find() has amortized complexity very close to $O(1)$

2. Priority queues
   - insert() operation is $O(\sqrt{n})$
   - delete_max() operation is $O(\sqrt{n})$
   - Processing n items is $O(n\sqrt{n})$

Efficient data structures

3. Heaps
- Binary tree filled level by level, left to right
- The value at each node is at least as big the values of its children (max-heap)
- No “holes” allowed and cannot leave a level incomplete
- insert() operation is $O(\log n)$
- delete_max() / delete_min() operation is $O(\log n)$
- heapify() builds a heap in $O(n)$
- Heaps can also be used to sort a list in place in $O(n \log n)$

Efficient data structures

4. Search trees

- For each node with value v, all values in the left subtree are < v
- For each node with value v, all values in the right subtree are > v
- No duplicate values
- Each node has three fields, value, left, right
- Traversals: In-order, Pre-order, Post-order
- Worst case: An unbalanced tree with n nodes may have height $O(n)$
- find(), insert() and delete() all walk down a single path

Efficient data structures

5. Balanced search trees
   - Left and right subtrees should be "equal"
   - Two possible measures: size and height
   - Height balanced trees: height of left child and height of right child differ by at most 1 (AVL trees)
   - Using rotations, we can maintain height balance
   - AVL trees with n nodes will have height $O(\log n)$
   - find(), insert() and delete() all walk down a single path, take only $O(\log n)$

Programming, Data Structures and Algorithms using Python

Prof. Madhavan Mukund
Director
Chennai Mathematical Institute

Mr. Omkar Joshi
Course Instructor
IITM Online Degree Programme

Programming, Data Structures and Algorithms using Python

Summary of weeks 7 to 9

Content

- Greedy algorithms
  - Interval scheduling
  - Minimizing lateness
  - Huffman coding
- Divide and conquer
  - Counting inversions
  - Closest pair of points
  - Integer multiplication
  - Quick select
  - Recursion trees
- Dynamic programming
  - Memoization
  - Grid paths
  - Common subwords and subsequences
  - Edit distance
  - Matrix multiplication

Greedy algorithms

- Need to make a sequence of choices to achieve a global optimum
- At each stage, make the next choice based on some local criterion
- Never go back and revise an earlier decision
- Drastically reduces space to search for solutions
- Greedy strategy needs a proof of optimality
- Examples: Dijkstra’s algorithm, Prim’s algorithm, Kruskal’s algorithm
- Applications: Interval Scheduling, Minimizing Lateness, Huffman Coding

Interval scheduling

- IIT Madras has a special video classroom for delivering online lectures
- Different teachers want to book the classroom
- Slots may overlap, so not all bookings can be honored
- Choose a subset of bookings to maximize the number of teachers who get to use the room

1. Choose the booking whose starting time is earliest
2. Choose the booking spanning the shortest interval
3. Choose the booking that overlaps with minimum number of other bookings
4. Choose the booking whose finish time is the earliest

Minimizing lateness

- IIT Madras has a single 3D printer
- A number of users need to use this printer
- Each user will get access to the printer, but may not finish before deadline
- Goal is to minimize the lateness

1. Schedule requests in increasing order of length
2. Give priority to requests with smaller slack time
3. Schedule requests in increasing order of deadlines

Huffman coding

- At each recursive step, extract letters with minimum frequency and replace by composite letter with combined frequency
- Store frequencies in an array A
  - Linear scan to find minimum values
  - $|A| = k$, number of recursive calls is $k - 1$
  - Complexity is $O(k^2)$
- Instead, maintain frequencies in an heap
  - Extracting two minimum frequency letters and adding back compound letter are both $O(\log k)$
  - Complexity drops to $O(\mathrm{k} \log k)$

Divide and conquer

- Break the problem into disjoint subproblems
- Combine these subproblem solutions efficiently
- Examples: Merge sort, Quicksort
- Applications: Counting inversions, Closest pair of points, Integer multiplication, Quick select

Counting inversions

- Compare your profile with other customers
- Identify people who share your likes and dislikes
- No inversions – rankings identical
- Every pair inverted – maximally dissimilar
- Number of inversions ranges from 0 to $n(n - 1) / 2$
- An inversion is a pair (i, j), $i < j$, where j appears before i
- Recurrence: $T(n) = 2T(n / 2) + n = O(n \log n)$

Closest pair of points

- Split the points into two halves by vertical line
- Recursively compute closest pair in each half
- Compare shortest distance in each half to shortest distance across the dividing line
- Recurrence: $T(n) = 2T(n / 2) + O(n)$
- Overall: $O(n \log n)$

Integer multiplication

- Traditional method: $O(n^2)$

- Naïve divide and conquer strategy: $T(n) = 4T(n / 2) + n$

$$= O(n^2)$$

- Karatsuba's algorithm: $T(n) = 3T(n / 2) + n$

$$= O\left(n^{\log 3}\right)$$

$$\approx O(n^{1.59})$$

Quick select

- Find the k^{th} largest value in a sequence of length k
- Sort in descending order and look at position k – $O(n \log n)$
- For any fixed k, find maximum for k times – $O(kn)$
  - $k = \mathrm{n} \,/\, 2$ (median) – $O(n^2)$
- Median of medians – $O(n)$
- Selection becomes $O(n)$ – Fast select algorithm
- Quicksort becomes $O(n \log n)$

Recursion trees

- Uniform way to compute the asymptotic expression for $T(n)$
- Rooted tree with one node for each recursive subproblem
- Value of each node is time spent on that subproblem excluding recursive calls
- Concretely, on an input of size n
  - $f(n)$ is the time spent on non-recursive work
  - r is the number of recursive calls
  - Each recursive call works on a subproblem of size n / c

- Recurrence: $\mathrm{T(n)} = rT(n / c) + f(n)$
- Root of recursion tree for $T(n)$ has value $f(n)$
- Root has r children, each (recursively) the root of a tree for $T(n / c)$
- Each node at level d has value $f(n / c^d)$

Dynamic programming

- Solution to original problem can be derived by combining solutions to subproblems
- Examples: Factorial, Insertion sort, Fibonacci series
- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies form a DAG
- Solve subproblems in topological order
  - Never need to make a recursive call

Memoization

- Inductive solution generates same subproblem at different stages
- Naïve recursive implementation evaluates each instance of subproblem from scratch
- Build a table of values already computed – Memory table
- Store each newly computed value in a table
- Look up the table before making a recursive call

Grid paths

- Rectangular grid of one-way roads
- Can only go up and right
- How many paths from (0, 0) to (m, n)?
- Every path has (m + n) segments
- What if an intersection is blocked?
- Need to discard paths passing through
- Identify DAG structure
  - Fill the grid by row, column or diagonal
- Complexity is $O(mn)$ using dynamic programming, $O(m+n)$ using memoization

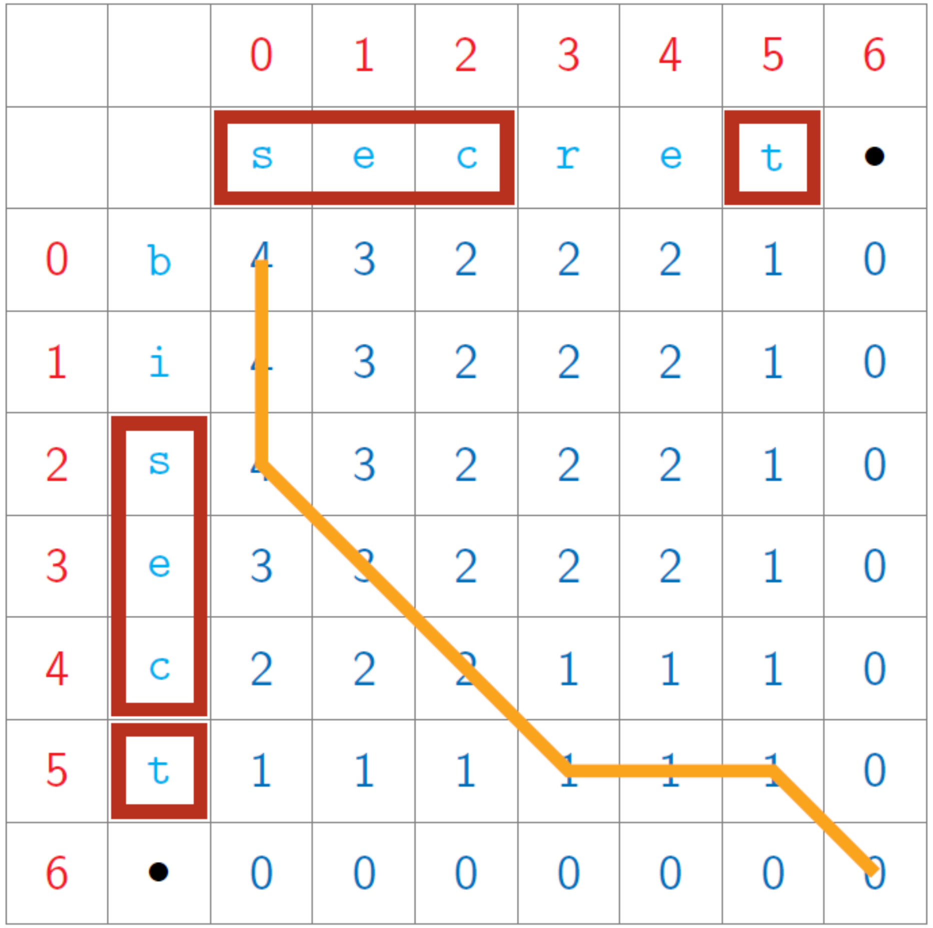
Common subwords and subsequences

- Given two strings, find the (length of the) longest common subword
- Subproblems are LCW(i, j), for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1)(n+1)$ values
- LCW(i, j), depends on LCW(i + 1, j + 1)
- Start at bottom right and fill row by row or column by column
- Complexity: $O(mn)$

		0	1	2	3	4	5	6
		s	e	c	r	e	t	●
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	●	0	0	0	0	0	0	0

Common subwords and subsequences

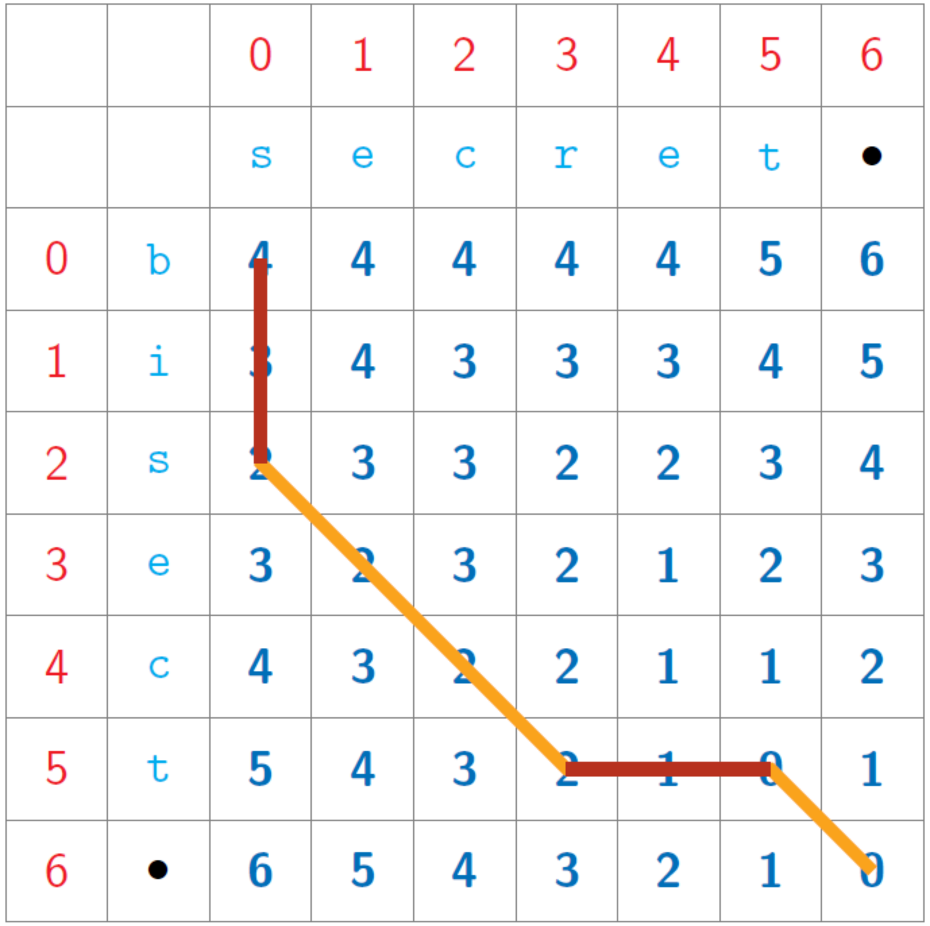
- Subsequence – can drop some letters in between
- Subproblems are LCS(i, j), for $0 \leq i \leq m$, $0 \leq j \leq n$
- Table of $(m + 1)(n + 1)$ values
- LCS(i, j), depends on LCS(i + 1, j + 1), LCS(i, j + 1), LCS(i + 1, j)
- Start at bottom right and fill row by row, column or diagonal
- Complexity: $O(mn)$



		0	1	2	3	4	5	6
		s	e	c	r	e	t	●
0	b	4	3	2	2	2	1	0
1	i	4	3	2	2	2	1	0
2	s	4	3	2	2	2	1	0
3	e	3	3	2	2	2	1	0
4	c	2	2	2	1	1	1	0
5	t	1	1	1	1	1	1	0
6	●	0	0	0	0	0	0	0

Edit distance

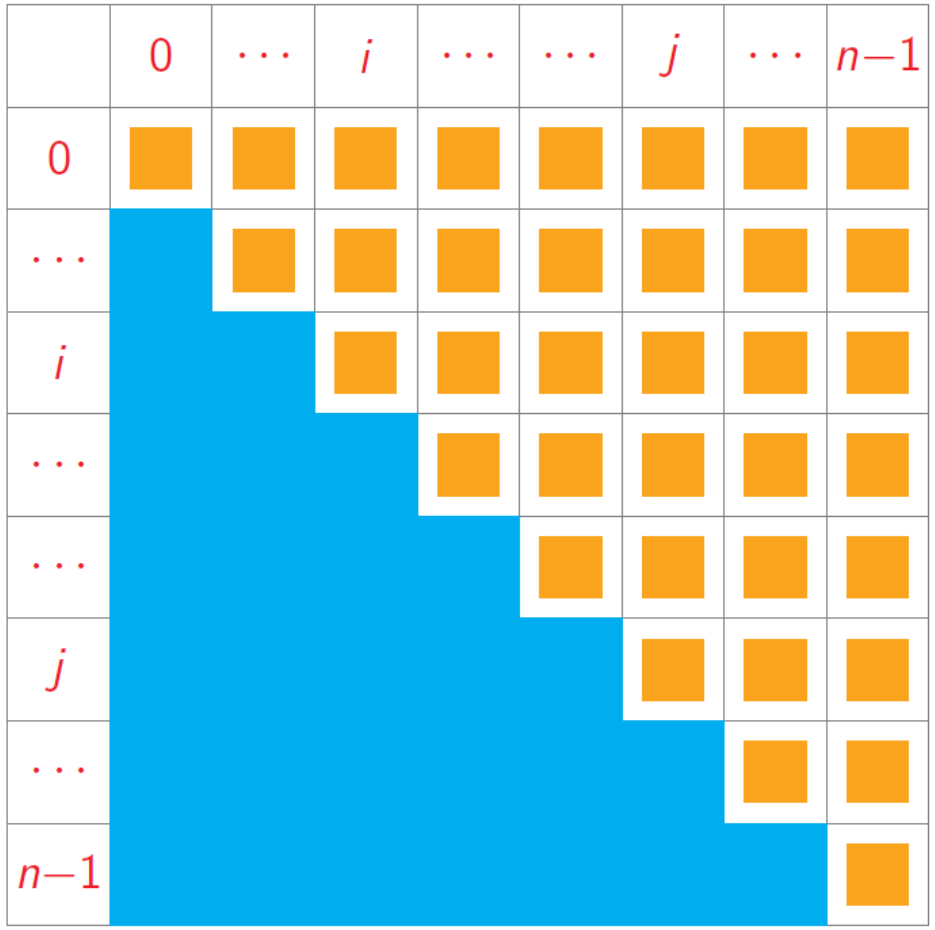
- Minimum number of editing operations needed to transform one document to the other
- Subproblems are ED(i, j), for $0 \le i \le m$, $0 \le j \le n$
- Table of $(m+1)(n+1)$ values
- ED(i, j), depends on ED(i + 1, j + 1), ED(i, j + 1), ED(i + 1, j)
- Start at bottom right and fill row, column or diagonal
- Complexity: $O(mn)$



		0	1	2	3	4	5	6
		s	e	c	r	e	t	●
0	b	4	4	4	4	4	5	6
1	i	3	4	3	3	3	4	5
2	s	2	3	3	2	2	3	4
3	e	3	2	3	2	1	2	3
4	c	4	3	2	2	1	1	2
5	t	5	4	3	2	1	0	1
6	●	6	5	4	3	2	1	0

Matrix multiplication

- Matrix multiplication is associative
- Bracketing does not change answer but can affect the complexity
- Find an optimal order to compute the product
- Compute C(i, j), $0 \leq i, j < n$, only for $i \leq j$
- C(i, j), depends on C(i, k – 1), C(i, k) for every $i < k \leq j$
- Diagonal entries are base case, fill matrix from main diagonal
- Complexity: $O(n^3)$



Programming, Data Structures and Algorithms using Python

Prof. Madhavan Mukund
Director
Chennai Mathematical Institute

Mr. Omkar Joshi
Course Instructor
IITM Online Degree Programme

Programming, Data Structures and Algorithms using Python

Summary of weeks 10 & 11

Content

- String matching
  - Boyer-Moore Algorithm
  - Rabin-Karp Algorithm
  - Automata
  - Knuth-Morris-Pratt Algorithm
  - Tries
  - Regular Expressions
- Linear programming
- Network flows
  - Ford-Fulkerson algorithm
- Reductions
- Checking algorithms
- P, NP and NP-Complete

String matching

- Searching for a pattern is a fundamental problem when dealing with text
- Formal definition:
  - A ***text*** string t of length n
  - A ***pattern*** string p of length m
  - Both t and p are drawn from an ***alphabet*** of valid letters, denoted by ∑
  - Find every position i in t such that t[i : i + m] == p
- Complexity of naïve algorithm: $O(nm)$

Boyer-Moore Algorithm

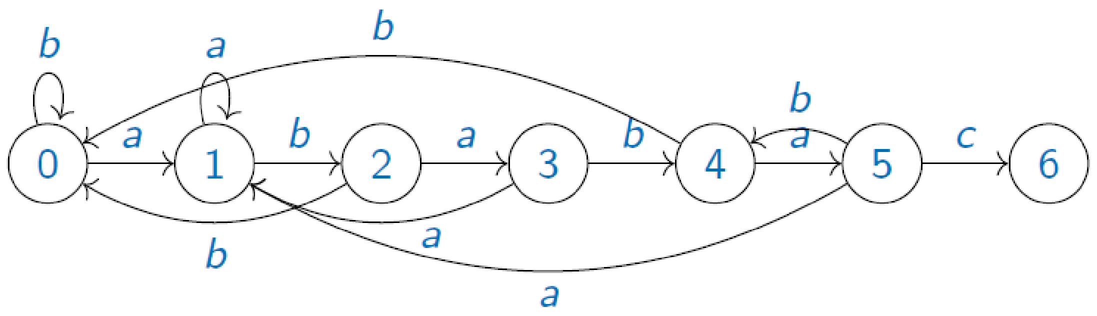
- For each starting position i in t, compare t[i : i + m] with p
  - Scan t[i : i + m] right to left
- If a letter c in t that does not appear in p
  - Shift the next scan to position after mismatched letter c
- If a letter c in t that does appear somewhere in p
  - Align rightmost occurrence of c in p with t
- Use a dictionary last
  - For each c in p, last[c] records right most position of c in p
- Without dictionary, computing last is a bottleneck, complexity is $O(|\Sigma|)$
- The algorithm works well in practice but the worst case complexity remains $O(nm)$

Rabin-Karp Algorithm

- Any string over $\sum$ can be thought of as a number in base 10
- Pattern p is an m digit number n_p
- Each substring of length m in the text t is again an m digit number
- Scan t and compare the number n_b generated by each block of m letters with the pattern number n_p
- Whenever $n_b = n_p$, scan and validate like brute force
  - It can be a false positive (spurious hit)
- In practice number of spurious matches will be small but the worst case complexity remains $O(nm)$
- If $|\sum|$ is small enough to not require modulo arithmetic, overall time is $O(n + m)$ or $O(n)$, since $m \ll n$

Automata

- It is used to keep track of longest prefix that we have matched
- It is a special type of graph where nodes are states and edges describe how to extend the match
- Using this automaton, we can do string matching in $O(n)$
- Bottleneck is precomputing the automaton
  - Overall complexity: $O(m^3 . |\sum|)$



Processing *abababac*

$0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{b} 4 \xrightarrow{a} 5 \xrightarrow{c} 6$

Knuth-Morris-Pratt Algorithm

- It is very similar to what we did in automaton
- Precomputing step can be handled effectively using fail()
- The Knuth-Morris-Pratt algorithm efficiently computes the automaton describing prefix matches in the pattern p
- Complexity of preprocessing the fail() is $O(m)$
- After preprocessing, can check matches in the text t in $O(n)$
- Overall, KMP algorithm works in time $O(m+n)$
- However, the Boyer-Moore algorithm can be faster in practice, skipping many positions

Tries

- A ***trie*** is a special kind of tree deriver from "information re***trie***val"
- Rooted tree
  - Other than root, each node labelled by a letter from ∑
  - Children of a node have distinct labels
- Each maximal path is a word
  - One word should not be a prefix of another
  - Add special end of word symbol $
- Build a trie T from a set of words S with s words and n total symbols
- To search for a word w, follow its path
  - If the node we reach has $ as a successor, $w \in S$
  - If we cannot complete the path, $w \notin S$
- Using a suffix trie we can answer the following:
  - Is w a substring of s
  - How many times does w occur as a substring in s
  - What is the longest repeated substring in s

Regular Expressions

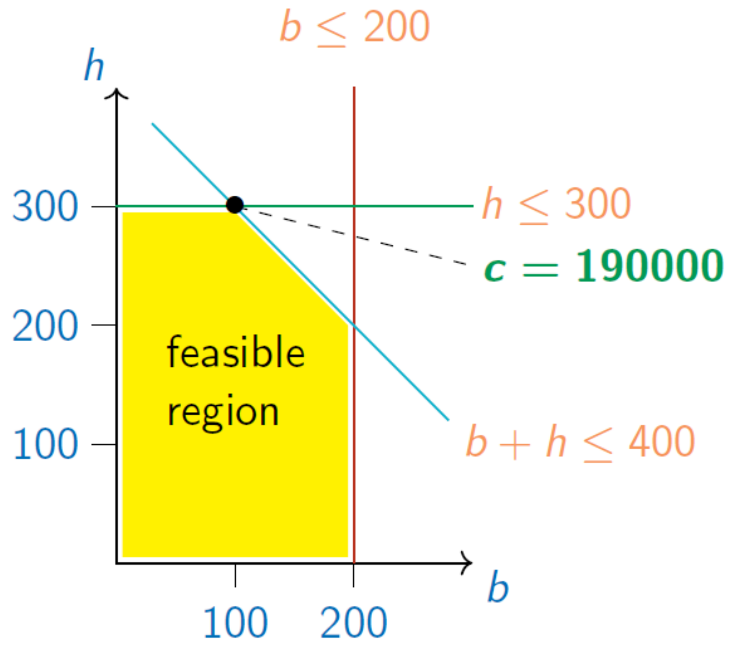
- The automaton we built had a linear structure, with a single path from start to finish
- In general, automata can follow multiple paths and accept multiple words
- The set of words that automata can accept are called regular sets
- Each pattern p describes a set of words, those that it matches
- The sets we can describe using patterns are exactly the same as those that can be accepted by automata
- Those patterns are called regular expressions
- For every automaton, we can construct a pattern p that matches exactly the words that the automaton accepts
- For every pattern p, we can construct an automaton that accepts all words that match p
- We can extend string matching to pattern matching by building an automaton for a pattern p and processing the text through this automaton
- Python provides a library for matching regular expressions

Linear programming

- Profit for each box of barfis is Rs.100
- Profit for each box of halwa is Rs.600
- Daily demand for barfis is at most 200 boxes
- Daily demand for halwa is at most 300 boxes
- Staff can produce 400 boxes a day, altogether
- What is the most profitable mix of barfis and halwa to produce?

Constrains:

- $b \leq 200$
- $h \leq 300$
- $b + h \leq 400$
- $b \geq 0$
- $h \geq 0$



Objective:

- Maximize $100b + 600h$

Linear programming

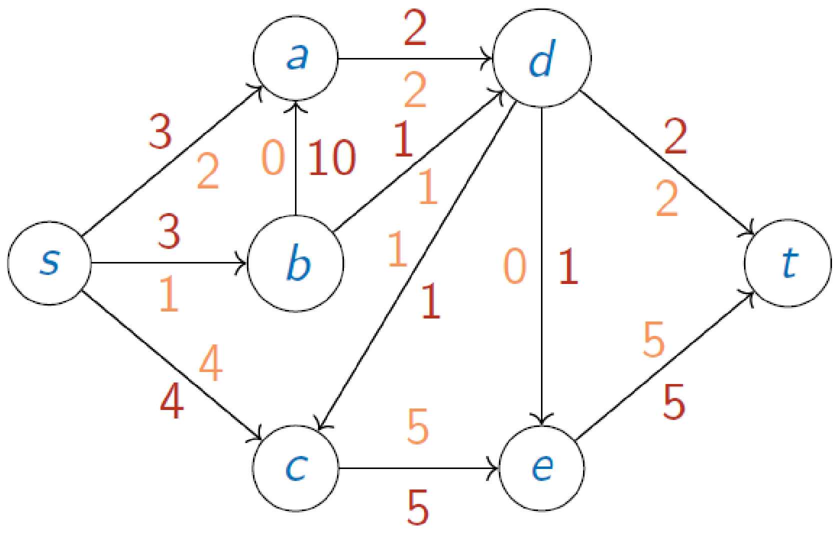
- Constraints and objective to be optimized are linear functions
  - Constrains: $a_1x_1 + a_2x_2 + \ldots + a_mx_m \leq K$, $b_1x_1 + b_2x_2 + \ldots + b_mx_m \geq L$
  - Objective: $c_1x_1 + c_2x_2 + \ldots + c_mx_m$

- Start at any vertex, evaluate objective
- If an adjacent vertex has a better value, move
- If current vertex is better than all neighbors, stop
- Can be exponential, but efficient in practice

- Feasible region is convex
- May be empty – constraints are unsatisfiable, no solutions
- May be unbounded – no upper/lower limit on objective

Network flows

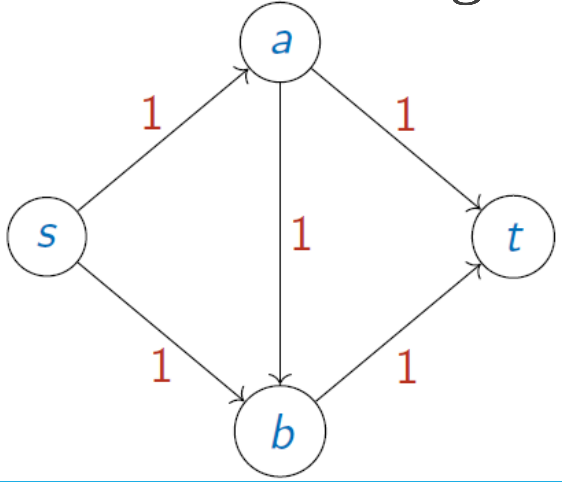
- Network: graph G = (V, E)
- Special nodes: s (source), t (sink)
- Each edge e has capacity c_e
- Flow: f_e for each edge e
  - $f_e \leq c_e$
  - At each node, except s and t, sum of incoming flows equal sum of outgoing flows
- Total volume of flow is sum of outgoing flow from s

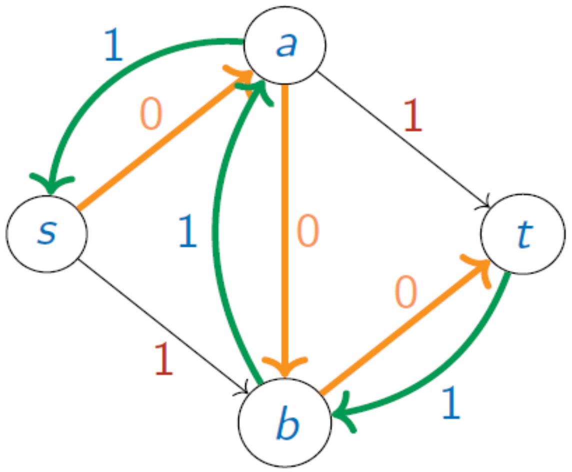


Ford-Fulkerson algorithm

- Start with zero flow
- Choose a path from s to t that is not saturated and augment the flow as much as possible
- Network given in the example has max flow 2
- What if one chooses a bad flow to begin with?
- Add reverse edges to undo flow from previous steps

- Residual graph: for each edge e with capacity c_e and current flow f_e
  - Reduce capacity to $c_e - f_e$
  - Add reverse edge with capacity f_e
- Use BFS to find augmenting path with fewest edges

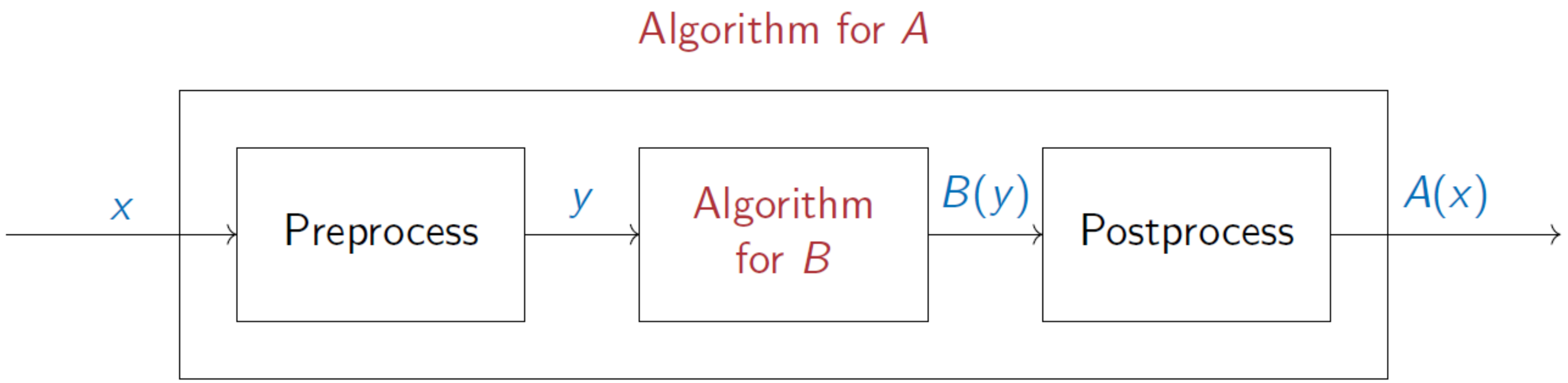




Reductions

- We want to solve problem A
- We know how to solve problem B
- Convert input for A into input for B
- Interpret output of B as output of A
- A reduces to B

- Can transfer efficient solution from B to A
- But preprocessing and postprocessing must also be efficient
- Typically, both should be polynomial time



Reductions

- Bipartite matching reduces to max flow
- Max flow reduces to LP
- Number of variables, constraints is linear in the size of the graph
- Reverse interpretation is also useful
- If A is known to be intractable and A reduces to B, then B must also be intractable
- Otherwise, efficient solution for B will yield efficient solution for A

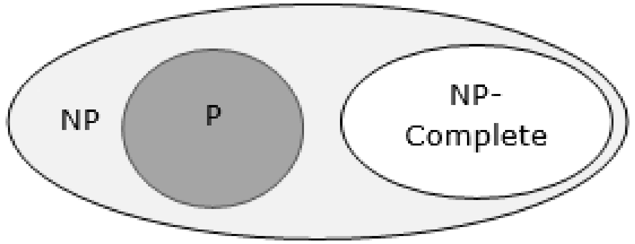
Checking algorithms

- Factorize a large number that is the product of two primes
- Generate a solution
  - Given a large number N, find p and q such that pq = N
- Check a solution
  - Given a solution p and q, verify that pq = N
- Examples: satisfiability, travelling salesman, vertex cover, independent set, etc.

- Checking algorithm C for problem P
- Takes an input instance I for P and a solution “certificate” S for I
- C outputs yes if S represents a valid solution for I, no otherwise
- For factorization
  - I is N
  - S is {p, q}
  - C involves verifying that pq = N

P, NP and NP-Complete

- P (**P**olynomial) is the class of problems with regular polynomial time algorithms (worst-case complexity)
- NP (**N**on-deterministic **P**olynomial) is the class of problems with checking algorithms
- An algorithm A is NP-Complete if it satisfies two conditions:
  - A is in NP
  - Every algorithm in NP is polynomial time reducible to A


NP
P
NP-
Complete