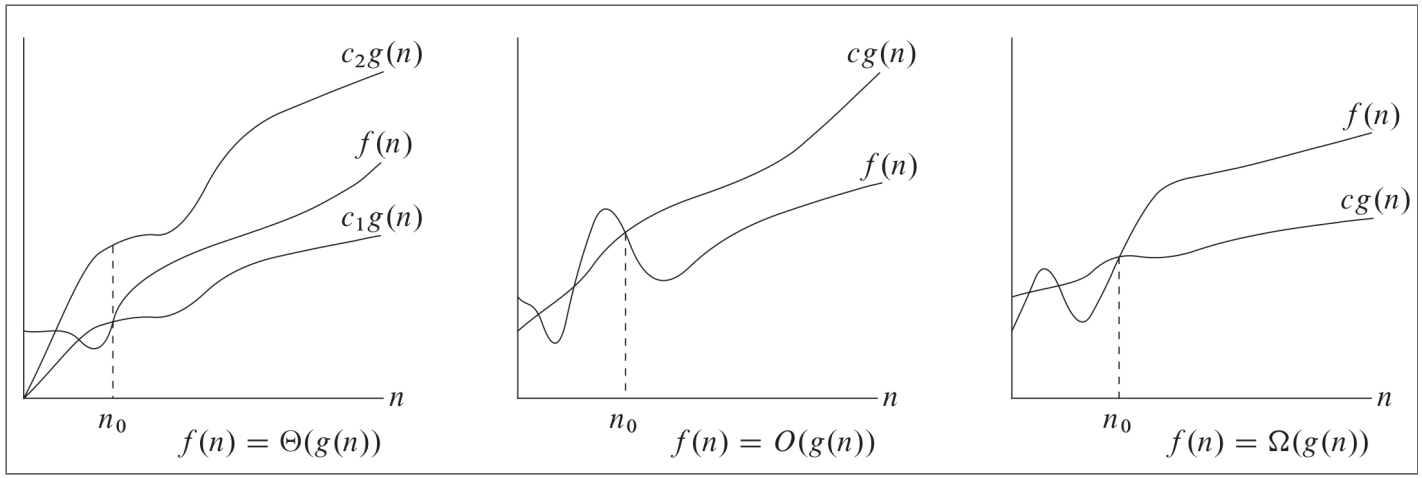
4M26 - Algorithms and Data Structures

January, 2023



Lectures 2 & 3 - Fundamentals of Algorithms

Slides: Ignas Budvytis

Formalising Algorithms

Why not just code?

- Building intuition of why and how certain algorithms work
- Proving correctness
- Proving computational properties (e.g. execution time and memory usage)
- Inventing new algorithms
- Teaching machines to write/invent algorithms

How to formalise?

- Defining a model of computation (instructions, memory)
- Defining mathematical structure
- Proving properties of such structure
    - Often by defining an invariant over initial state of an algorithm and proving that it holds as the algorithm progresses

We will closely follow the formalism of:

Introduction to Algorithms, MIT Press, *by Cormen, T. H., Leiserson, C., E., Rivest, R., L.*.

Model of Computation

Model of computation: **random-access machine** (RAM)

1. Instructions are executed sequentially.
2. No concurrent operations are allowed.
3. Each instruction takes constant amount of time.
4. Do not model memory hierarchy (e.g. cache or virtual memory).

Instructions:

1. Arithmetic (add, subtract, multiply, divide, remainder, floor, ceiling)
2. Boolean (shift left/right, and, or, not, nor)
3. Data movement (load, store, copy)
4. Control (conditional and unconditional branch, subroutine call and return)

Data types:

1. Integer
2. Floating point
3. Boolean
4. Char

Input size: e.g. number of items in the input, total number of bits, number of vertices and edges.

Running time of a particular input is the number of primitive operations executed.

Elementary Data Structures

Dynamic sets are sets manipulated by algorithms which can grow, shrink, or otherwise change over time. They can also have some additional structure, such as total ordering of the elements.

Key operations include:

- **search**(S, k) - a query, that given a set, S, and a key value, k, returns a pointer, x, to an element in S such that $x.key = k$, or NIL if no such element belongs to S.

- **insert**(S, x) - a modifying operation that augments the set, S, with the element pointed to by x.

- **delete**(S, x) - a modifying operation that, given a pointer, x, to an element in the set, S, removes x from S.

- **minimum**(S) - a query on a totally ordered set, S, that returns a pointer to the element of S with the smallest key.

- **maximum**(S) - a query on a totally ordered set, S, that returns a pointer to the element of S with the largest key.

- **successor**(S, x) - a query that, given an element, x, whose key is from a totally ordered set, S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

- **predecessor**(S, x) - a query that, given an element, x, whose key is from a totally ordered set, S, returns a pointer to the next smaller element in S, or NIL if x is the minimum element.

Stack

- A *dynamic set* in which the element deleted from the set is the one most recently inserted. Stack implements **last-in, first-out** (LIFO) policy.
- Insert operation usually called, **PUSH**, and delete operation is often called, **POP**.

Queue

- A *dynamic set* in which the element deleted from the set is the one which resided in the set for the longeset time. Queue implements **first-in, first-out** (FIFO) policy.
- Insert operation is usually called, **ENQUEUE**, and delete operation - **DEQUEUE**.

Linked list

- A data structure in which the objects are arranged in linear order. Order in the linked list is determinded by a pointer in each object.
- *Doubly linked list* is an obect with an attribute, key, and two pointer attributes, $next$ (successor) and $prev$ (predescessor).
- In a *circular linked list*, the $prev$ pointer of the head of the list points to the tail and the $next$ pointer of the tail points to the head.

Tree

- A linked data structure with pointers to parents, childrens and siblings.
- *Binary trees* usually have pointers to parent, left child and right child.
- *Rooted trees* have parent, left-child, right-sibling pointers. This allows to scale well to trees with high and varying numbers of children per node.

Dictionary

- A dynamic set which supports three types of operations: **insert**, **delete** and test the membership of an element in a set (ie. **search**).

Pseudocode and Python Code

Pseudocode is used to describe algorithms.

Unlike "real" code, in pseudocode the most convenient/clear/concise method is used to specify the given algorithm.

Issues of data abstraction, modularity and error handling are often ignored.

	Pseudo-code example
	INSERTION-SORT(A)
1	**for** $j = 2$ **to** $A.\,length$
2	$key = A[j]$
3	//Insert $A[j]$ into the sorted sequence A[1 .. j-1].
4	$i = j - 1$
5	**while** $i > 0$ and $A[i] > key$
6	$A[i + 1] = A[i]$
7	$i = i - 1$
8	$A[i + 1] = key$

Python code example

In [2]:

```
def insertion_sort (A):
    for j in range(1,len(A)):
        key = A[j]
        i = j - 1
        while i >= 0 and A[i] > key:
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
A = [4, 3, 1, 4]
print('A = ',A)
insertion_sort(A)
print('Sorted A = ',A)
```

```
A =  [4, 3, 1, 4]
Sorted A =  [1, 3, 4, 4]
```

Note that in pseudocode we follow 1-based indexing convention whereas in Python we use 1-based indexing.

Proof of Correctness

Loop invariants are used to understand why an algorithm is correct. Three things are shown about the loop invariant.

Initialization. Loop invariant is true prior to the first iteration.

Maintenance. If it is true before the iteration of the loop, it remains true before next iteration.

Termination. When the loop terminates, the invariant gives a useful property that helps to show that the algorithm is correct.

Note that loop invariant is (slightly) different to the mathematical induction as the latter is applied indefinitely.

Proof of correctnes for Insertion sort

Loop invariant. Elements $A[:j]$ are sorted before j-th execution of the for loop.

Initialisation. Clearly true for $j = 1$ since array of one element is sorted.

Maintenance. Lets assume that the proposed condition is true for $j = k$, $k >= 1$. Then we need to prove that it will be true for $j = k + 1$.

Informally, the **while** loop works by moving $A[k]$, $A[k-1]$, etc until it finds the proper position for $A[k+1]$ and updated subarray $A[:k+1]$ is sorted.

Termination. When **for** loop terminates we have $j = n$, hence the elements of the array, $A[:n]$, are sorted from the loop invariant. Which means that the entire array is sorted.

Run Time Analysis - Example

	Pseudo-code	Cost	Times
	INSERTION-SORT(A)		
1	**for** $j = 2$ **to** $A.\,length$	c_1	n
2	$key = A[j]$	c_2	$n - 1$
3	//Insert $A[j]$ into the sorted sequence $A[1 \cdots j - 1]$.	0	n-1
4	$i = j - 1$	c_4	$n - 1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i + 1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i = i - 1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i + 1] = key$	c_8	$n - 1$

Run time analysis of insertion sort

t_j - number of times the while loop test in line 5 is executed for value j.

Running time for an input with n values:

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$.

Running time may depend on which input of a particular size n is given.

Best case running time is when array is sorted.

$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$ - linear in n.

Worst case running time is when array is reverse sorted.

$T(n) = \frac{c_5 + c_6 + c_7}{2} n^2 + (c_1 + c_2 + c_4 + c_8 + \frac{c_5 - c_6 - c_7}{2})n - (c_2 + c_4 + c_5 + c_8)$ - quadratic function of n.

Types of Run Time Analysis

Worst case running time

- The longest running time for *any* input of certain size.
- Provides a guarantee that the algorithm will never take any longer.
- For some algorithms the worst case appears often. E.g. searching for non-existing item in a database.

Average case running time

- Average time per input (often) assuming that the inputs of the given size are equally likely.
- It is usually as bad as the worst case. (e.g. quadratic function of n for insertion sort).

Best case running time

- The shortest running time for *any* input of certain size.
- Rarely used.

Amortized cost

- Average cost per operation (ie. $\frac{T(n)}{n}$) for the worst case time, $T(n)$, of a sequence of n operations.
- The average cost of an operation may be small, if one averages over a sequence of operation, even if several operations within the sequence may be expensive.

Expected running time

- Expected average time per sample of an outcome of a randomised algorithm.

Asymptotic Notation

Order of growth

- Lower order terms are relatively insignificant for large values of input size, n.
- Leading term's constant coefficient is less significant than the rate of growth.
- Usually one algorithm is considered to be more efficient if its worst case running time has lower order of growth.
- For small inputs an algorithm with higher order of growth may be faster.
- For large inputs an algorithm with higher order of growth will be slower.

Asymptotic tight bound - "Big Theta"

For a given function, $g(n)$, we denote, $\Theta(g(n))$, to be a set of functions $f(n) : N \to R$ as follows:

$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 \in R^+, n_0 \in N^+ \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}.$

Note, often $f(n) = \Theta(g(n))$ is used instead of $f(n) \in \Theta(g(n))$. E.g. $2n^2 + 4n + 1 = \Theta(n^2)$.
Finally, $\Theta(n^0)$ is often noted as $\Theta(1)$.

Asymptotic upper bound - "Big O"

$O(g(n)) = \{f(n) : \exists c \in R^+, n_0 \in N^+ \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}.$
Note, $f(n) = \Theta(g(n)) \implies f(n) = O(g(n))$, or $\Theta(g(n)) \subseteq O(g(n))$.

Asymptotic lower bound - "Big Omega"

$\Omega(g(n)) = \{f(n) : \exists c \in R^+, n_0 \in N^+ \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$
For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Example Run Time Analysis: MergeSort

	Pseudo-code	Cost	Times
	MERGE-SORT(A,p,r)		
1	**if** $p < r$	c_{18}	1
2	$q = \lfloor \frac{p+r}{2} \rfloor$	c_{19}	$\mathbb{1}[p < r]$
3	MERGE-SORT(A,p,q)	$T(\lfloor \frac{n}{2} \rfloor)$	$\mathbb{1}[p < r]$
4	MERGE-SORT$(A,q+1,r)$	$T(\lfloor \frac{n}{2} \rfloor)$	$\mathbb{1}[p < r]$
5	MERGE(A,p,q,r)	$c_{20}n$	$\mathbb{1}[p < r]$

	Pseudo-code	Cost	Times
	MERGE(A,p,q,r)		
1	$n_1 = q - p + 1$	c_1	1
2	$n_2 = r - q$	c_2	1
3	let $L[1, \cdots, n_1 + 1]$ and $R[1 \cdots, n_2 + 1]$ be new arrays	c_3	1
4	**for** $i = 1$ to n_1	c_4	$n_1 + 1$
5	$L[i] = A[p + i - 1]$	c_5	n_1
6	**for** $j = 1$ to n_2	c_6	$n_2 + 1$
7	$R[j] = A[q + j]$	c_7	n_2
8	$L[n_1 + 1] = \infty$	c_8	1
9	$R[n_2 + 1] = \infty$	c_9	1
10	$i = 1$	c_{10}	1
11	$j = 1$	c_{11}	1
12	**for** $k = p$ to r	c_{12}	$n_1 + n_2 + 1$
13	**if** $L[i] \leq R[j]$	c_{13}	$n_1 + n_2$
14	$A[k] = L[i]$	c_{14}	$n_1 + n_2 - a$
15	$i = i + 1$	c_{15}	$n_1 + n_2 - a$
16	**else** $A[k] = R[j]$	c_{16}	a
17	$j = j + 1$	c_{17}	a

Recurrence Equations for MergeSort

Unifying constants:

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ 2T(\lfloor \frac{n}{2} \rfloor) + cn + c, & \text{if } n > 1 \end{cases}$$

Note two different constants may be used for the lower bound and upper bound.

Using asymptotic notation:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T(\lfloor \frac{n}{2} \rfloor) + \Theta(n), & \text{if } n > 1 \end{cases}$$

Substitution method

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that solution works.

Example:

Let $T(n) = 2T(\lfloor n/2 \rfloor) + n$, with $T(1) = 1$.

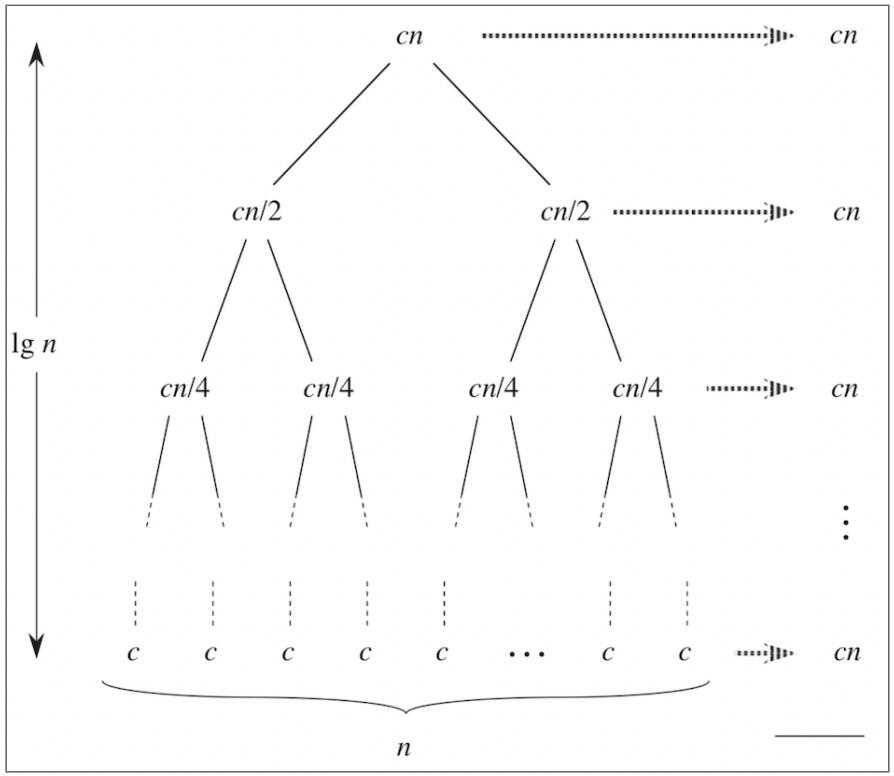
Guess: $T(n) = O(n \log_2 n)$.

Proof: Assume that the bound holds for any m such that $m < n$.

Then $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor) + n$ for some c.

Hence, $T(n) = 2T\lfloor (n/2 \rfloor) + n \leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \leq cn \log_2 n - cn \log_2 2 + n \leq cn \log n$.
Note this holds when $c \geq 1$.
Choose $n = 2$ as a boundary condition, since $T(2) = 4 \leq c2 \log_2 2$ when $c \geq 2$. Note that $c1 \log_2 1 = 0$ for any c.

Generating guesses for recurrence equations using recurrence tree



Total time: $cn \log n + cn$**. Hence** $\Theta(n \log n)$**.**
Needs to be verified using subsitution method.

Algorithm Design Strategies

Approach	Description
Divide and conquer	Breaks a problem into several **subproblems** that are similar to the original problem but smaller in size. Solve the problems recursively and then **combine** these solutions to create a solution to the original problem.
Dynamic programming	Like *divide and conquer* it breaks a problem into several subproblems. It applies when some of the subproblems overlap (i.e. share subsubproblems), making the combination step more efficient.
Greedy algorithms	In greedy algorithms the problem is split in such a way that the combination step can be performed in a locally optimal manner.
Other	Brute force, backtracking, branch and bound, transform and conquer, etc.

Divide and Conquer

Recursively applying three steps at each level of the recursion:

- **Divide** the problem into several subproblems that are similar to the original problem but smaller in size.
- **Conquer** the subproblems by solving them recursively. If the subproblem size is small enough, solve it in a straight-forward manner.
- **Combine** the solutions to subproblems into the solution to the original problem.

Examples:

- Sorting: MergeSort, QuickSort, Heapsort.
- Search: binary search.
- Computational algorithms: Strassen's matrix multiplication algorithm.
- Signal processsing: Fast Fourier Transform.
- Geometric algorithms: finding closest pair of points.

Binary Search

Given a sorted array, A, find the location of element, k. If such element does not exist, return -1.
Worst case running time - $\Theta(\log_2 n)$.

In [19]:

```
def binary_search(A,p,r,k):
    if p > r:
        return -1
    q = (p+r)//2
    if A[q] == k:
        return q
    elif k < A[q]:
        return binary_search(A,p,q-1,k)
    else:
        return binary_search(A,q+1,r,k)

A = [1, 2, 4, 6, 8, 15,20]
print('A = ',A)
k = 8
print('Element', k,'location:',str(binary_search(A,0,len(A)-1,k)))
```

```
A =  [1, 2, 4, 6, 8, 15, 20]
Element 8 location: 4
```

Closest pair in 2D

Given a list, P, of $n \geq 2$ of 2-dimensional points, find a pair (in Euclidean distance sense) of closest points.
Running time: $O(n \log_2 n)$.

	Pseudo-code example
	CLOSEST-PAIR(P,X,Y)
1	Find a vertical line that splits P in half: P_L and P_R
2	Divide X and Y into X_L, X_R and Y_L, Y_R
3	δ_L = Closest_Pair(P_L,X_L,Y_L)
4	δ_R = Closest_Pair(P_R,X_R,Y_R)
5	$\delta = \min(\delta_L, \delta_R)$
6	Select points $Y' \subseteq Y$ which are within 2δ stripe around the line
7	**for** each point, p, in Y'
8	compute the minimum distance between p and 7 consecutive points: δ'_p
9	$\delta' = \min_p \delta'_p$
10	**return** $\min(\delta, \delta')$

Dynamic Programming

Problems solved using dynamic algorithms exibit **optimal substructure** and **overlapping problems**. A problem exhibits **optimal substructure** when optimal solutions to a problem incorporate optimal solutions to related subproblems.

Key steps:

1. Characterise the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution (typically in bottom up fashion).
4. Construct an optimal solution from computed information.

Examples:

- Optimal control and RL: Bellman equation.
- Graph algorithms: Bellman-Ford, Floyd-Warshall.
- Optimization: 0-1 Knapsack problem.
- String algorithms: Longest common subsequence, longest increasing subsequence.

0-1 Knapsack Problem

Problem definition

Given a set of n items numbered from 1 to n, each with weight $w_i \in N$ and value $v_i \in N$, along with maximum weight capacity $W \in N$, maximize the total value that can be taken.

I.e. Maximize $\sum_{i=1}^{n} v_i x_i$, subject to $\sum_{i=1}^{n} w_i x_i \leq W$ and $x_i \in \{0, 1\}$.

E.g. If $W = 10$, $w = [1, 3, 4, 8, 1]$, $v = [2, 4, 1, 6, 6]$ then one would take items $1, 4, 5$ with total weight 9 and value 14.

Recursive Solution - runtime $O(2^n)$

	Pseudo-code
	KNAPSACK-Rec(v,w,W,r)
1	**if** $r > 0$ and $W \geq 0$
2	$val1$ = KNAPSACK-Rec(v,w,$W - w[r]$,$r - 1$)+$v[r]$
3	$val2$ = KNAPSACK-Rec(v,w,W,$r - 1$)
4	**return** max($val1$, $val2$)
5	**return** 0

Recursive solution with memoization - runtime $O(nW)$

Let $S[1..n, 0..W]$ be a new array. Initilise all elements of S to $-\infty$.

	Pseudo-code
	M-KNAPSACK-Rec(v,w,W,r,S)
1	**if** $r > 0$ and $W \geq 0$
2	**if** $S[r, W] \geq 0$
3	**return** $S[r, W]$
4	**else**
5	$val1$ = M-KNAPSACK-Rec(v,w,$W - w[r]$,$r - 1$,S) + $v[r]$
6	$val2$ = M-KNAPSACK-Rec(v,w,W,$r - 1$,S)
7	$S[W, r]$ = max($val1, val2$)
8	**return** $S[W, r]$
9	**return** 0

Dynamic programming solution

	Pseudo-code
	KNAPSACK-0-1(v,w,W)
1	Let $S[0..n, 0..W]$ be a new array
2	Initilise all elements of S to $-\infty$.
3	$S[0, \cdots] = 0$
4	**for** $i = 1$ to n
5	**for** $weight = 1$ **to** W
6	**if** $w[i] \leq weight$
7	$S[i,weight] = \max(S[i\text{-}1,weight], S[i-1, weight\text{-}w[i]] + v[i])$
8	**else:**
9	$S[i, weight] = S[i-1, weight]$
10	**return** $S[n, W]$

Runtime: $\Theta(nW)$

Longest Common Subsequence

Problem definition

Find the length of the longest common subsequence of two strings, x and y.

Note, a subsequence is a string that can be derived from another string by deleting some or no elements without changing the order of the remaining elements.

Runtime: $O(mn)$. Here m and n are lengths of strings, x and y, respectively.

In [20]:

```
def LCS(x,y):
    m, n = len(x), len(y)
    c = [[] for i in range(m+1)]
    for i in range(n+1):
        c[i] = [0 for j in range(n+1)]
    for i in range(m):
        for j in range(n):
            if x[i]==y[j]:
                c[i+1][j+1]=c[i][j]+1
            elif c[i][j+1]>= c[i+1][j]:
                c[i+1][j+1] = c[i][j+1]
            else:
                c[i+1][j+1] = c[i+1][j]
    return c[n][m]
x, y = 'abcdefghi', 'ndnnnennf'
print('x =',x, 'y =', y)
print('Length of the longest common subsequence:',LCS(x,y))
```

```
x = abcdefghi y = ndnnnennf
Length of the longest common subsequence: 3
```

Greedy Algorithms

Greedy algorithms is a subclass of dynamic programming algorithms for which the locally optimal solution of a subproblem gives a globally optimal solution.

Examples:

- Graph algorithms: Kruskal's and Prim's algorithms, Dijkstra.
- Compression: Huffman codes.
- Optimisation: fractional knapsack problem.
- Scheduling: activity selection.

Activity Selection Problem

Problem definition

Let $S = \{a_1, a_2, \cdots, a_n\}$ be a set of n proposed activities that wish to use a resource (e.g. a lecture hall), which can serve only one activity at a time.

- Each activity has a start time, s_i, and a finish time, f_i, where $0 \leq s_i < f_i < \infty$.
- If selected, activity, a_i, takes place during the half-open time interval $[s_i, f_i)$.
- Activities, a_i and a_j, are compatible if the intervals, $[s_i, f_i)$ and $[s_i, f_i)$, do not overlap.
- Also assume, $f_1 \leq f_2 \leq \cdots \leq f_{n-1} \leq f_n$.

In the **activity selection problem**, we wish to select a maximum-size subset of mutually compatible activities.

Run time: $\Theta(n)$.

	Pseudo-code
	GREEDY-ACTIVITY-SELECTOR(s, f)
1	$n = s.length$
2	$A = \{a_1\}$
3	$k = 1$
4	**for** $m = 2$ **to** n
5	**if** $s[m] \geq f[k]$
6	$A = A \cup \{a_m\}$
7	$k = m$
8	**return** A

Huffman Codes

Huffman codes are binary character codes in which each character is represented by a unique variable length binary string, called codeword. Only codes (i.e. prefix codes) for which no codeword is also a prefix of some other codeword are considered.

Huffman codes construct optimal prefix codes, given a set of characters, C, where each character has a frequency $c.freq$ assigned.
The cost of the code is $B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$, where $d_T(c)$, is the length of the variable length code for character, c, encoded in a binary tree, T.

Example

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Total length: 300 bits for fixed-length code and 224 bits for variable length code.

Run time: $O(n \log_2 n)$.

	Pseudo-code
	HUFFMAN(C)
1	$n = \|C\|$
2	$Q = C$
3	**for** $i = 1$ **to** $n - 1$
4	allocate a new node z
5	$z.left = x$ = EXTRACT-MIN(Q)
6	$z.right = y$ = EXTRACT-MIN(Q)
7	$z.freq = x.freq + y.freq$
8	INSERT(Q,z)
9	**return** EXTRACT-MIN(Q)