Week 1 – Overview

Introduction

In this module, we will study the fundamental concepts that will setup the foundation for the rest of the course. We will talk about three notations: O ("big oh"), Ω ("big omega"), and θ ("big theta"), these will become our language for discussing the efficiency of algorithms. We will see how we can prove the correctness of an algorithm using the induction technique. We will use this knowledge to prove the correctness of a few algorithms and analyze their efficiency.

Exploration: The Framework for Analyzing an Algorithm

Introduction

As computer science students we know that "knowing about algorithms" is very important. But what is that we should know? For a given problem we should be able to write an algorithm that is 'correct' and 'efficient', and we should be able to communicate this to others.

At the end of this section, you will be able to write an algorithm in pseudocode and define the framework for analyzing the efficiency of an algorithm.

The Analysis Framework

As you study this section, consider the following questions

- What is an algorithm?
- How to write pseudocode?
- What is meant by running time of an algorithm?
- What is the order of growth?

Steps to Approach a Problem

When given a problem description, a good programmer doesn't directly jump to his or her favorite editor to start coding. One would follow these steps:

- Understand the Problem:
    - Read the problem carefully
    - Ask questions if you have doubts
    - Do a few small examples by hand
    - Think about special cases or 'boundary/edge' cases
- Restate the problem in terms of math objects:
    Sample problem statement: sort a sequence of numbers into non-decreasing order.
    Formally defining the sorting problem
    Input: A sequence of n numbers (a1, a2, …, an)
    Output: A recorded input sequence (a1', a2', …., an') such that a1' < a2 '< …. < an'
- Write an algorithm
    An algorithm is a set of specific instructions intended to solve a problem. Algorithms are not programs, rather algorithms are abstract mechanical procedures that can be implemented in a programming language.

- The clearest way to present an algorithm is using 'pseudocode', which conveys the instructions and structure of the algorithm clearly enough for us to implement it in the language of our choice. What separates pseudocode from 'real' code is that in pseudocode, we employ whatever expressive method is most clear and concise to specify a given algorithm. The convention that we follow here would appear similar to a Python language code, but we will not worry about syntax.
- Prove that the algorithm is correct & analyze how fast the algorithm can run:
    - We will use formal techniques to prove the correctness of our algorithm and we will analyze the efficiency of our algorithm. We will see these two steps in detail in this section and next explorations
- Code the algorithm in a programming language:
    - We finally write the code.

The Efficiency of an Algorithm

When we have our pseudocode ready, we will ask an important question: how quickly will the algorithm run? Why this question, because ideally, we want the fastest possible algorithm for any problem. This is also a common question in technical interviews coding rounds: can you make the algorithm more efficient?

Time efficiency is also called time complexity, it indicates how fast an algorithm runs (in other words, how complex the algorithm is in terms of the time it takes).

These questions about efficiency can be asked for memory (space) as well, called as space complexity, this refers to the number of memory units required by the algorithm in addition to the space needed for it is input and output.

Running Time

To identify the efficiency of an algorithm we would consider the time it takes to execute (run), i.e., the running time of the algorithm. For example, let's consider the problem to sort an array of integers and solve this problem using the BubbleSort technique.

```
Bubble-sort(a)
    for I = a.length() downto 0
        for j = 0 to I – 1
            if a[j] > a[j + 1]
                swap(a[j], a[j + 1]);
            end if
```

How fast an algorithms runs depend on: size of the array, speed of the CPU, programming language used to code the algo.

Did you get it right? For our algorithm analysis are we interested in these? No. We just want to compare how fast one algorithm runs in comparison to others that solve the same problems, i.e. we want to compare the running times of an algorithms.

But how do we measure running time? We can observe that the algorithm would take longer to sort larger arrays. Hence running time is a function of parameter n, where n is the input size of the algorithm. In the case of the BubbleSort algorithm, n is the size of the input array.

Mathematically, the running time can be expressed as a function of T(n).

The choice of the appropriate parameters that would impact efficiency would depend on the algorithm and the problem. For example, how should we evaluate the efficiency of a spell-checking algorithm, that examines individual characters of its input? The parameters that would impact efficiency are the number of the characters in the input words.

Order of Growth

Now, let us see how to write the function of T(n). For this, we would see how many times each algorithm's operation is executed.

Consider the following algorithm that prints "Hello" n times:

Line number (i)		Time taken to execute the instruction	Number of times the instruction is executed
1	def PrintHelloNTimes(n):	t_1	1
2	print("Start printing Hello in a Loop")	t_2	1
3	for i in range(n):	t_3	n+1
4	print("Hello")	t_4	n
5	print("End printing Hello")	t_5	1

Time taken to execute the code T(n) can be written as

$$T(n) = t_1*1 + t_2*1 + t_3*(n+1) + t_4*(n) + t_5*1$$

$$T(n) = (t_3+t_4)*n + (t_1 + t_2 + t_3 + t_5)$$

We can express this running time as T(n) = an + b, where a and b are constants that depend on the execution time of each line. Where a = (t3+ t4) and b = (t1 + t2 + t3 + t5)

Since T(n) is of form an + b, we can say T(n) of PrintHelloNtimes is a linear function of n.

Here is the time calculation of each line

```
Bubble-sort(a)
 1 for i = a.length() downto 0    ------------------------------(n+1) * t1
 2     for j = 0 to i-1      ----------------------------------(n   +   n-1  +  n-2  +.....+1)*t2
 3                if a[j]>a[j+1]  --------------------------------(n-1  +  n-2   +  n-3  +.....+1)*t3
 4                        swap(a[j],a[j+1]);------------------ at most (n-1  +  n-2   +  n-3  +.....+1)*t4
                  end if
```

For lines 3 & 4:

When I = 0:

J = 0 to -1. # j loop will not enter in this case

When I = 1:

J = 0 to 0 # even in this case loop will not be entered.

When I = 2:

J = 0 to 1 # loop will be executed for j = 0

When I = n:

J = 0 to n – 1 # loop will be executed for j = 0, 1, 2, …. n-2

When I = 0 and I = 1, no comparisons are made by the program. When I = 2, one comparison is made. When I = 3, two comparisons are made, and so on. Thus, we can conclude that when I = m, m – 1 comparison are made. Hence, in an array of length n, it does 1 + 2 + … + (n-2) + (n -1) comparisons. To solve this, you might want to review the following summation formula

$$\sum_{q=1}^{p} q = \frac{p(p+1)}{2}$$

using the summation formula $1 + 2 + \cdots + (n-2) + (n-1)$ can be written as,

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

So we can write T(n) as, T(n) = an^2 +bn + c. We call this a quadratic function as n is raised to a power of 2.

When T(n) = an^2 + bn + c, for simplifying the analysis we consider only the leading term of formula (e.g., an^2), since the lower-order terms are relatively insignificant for large values of n.

T(n) = n^2.

Now, let's see how the time taken by the algorithm changes as n grows. How much longer will the algorithm run if we double the input size? Since T(n) has a rate of growth of n^2, running time will become 2^2, i.e., 4 times. If input becomes 10 times, T(N) will increase 100 times. In the analysis, it is the rate of growth or order of growth of the running time that really interests us.

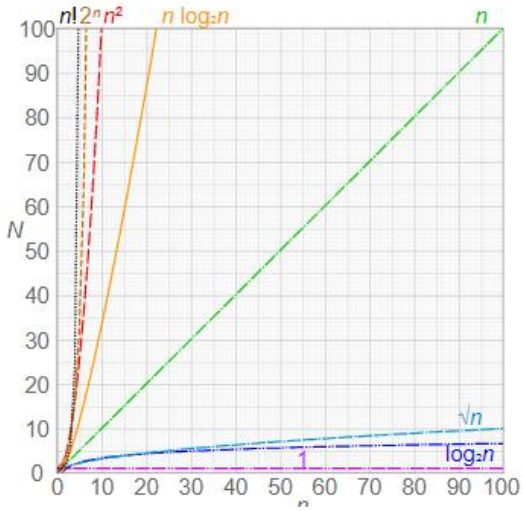
By focusing on the order of growth we can compare different algorithms. Look at the following table which displays the order of growth of different functions. Observe how logarithm functions grow the slowest and on the other end 2^n and n! grow so rapidly that we didn't even bother to calculate their values after the second row.

n	$\log_2 n$	linear n	$n \log_2 n$	quadratic n^2	polynomial n^3	exponential 2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

The below table shows running times for various input sizes for different algorithms on a processor that performs a million instructions per second. When running time exceeds 10^25 it is recorded as 'very long'. Observe how the running time increases for different functions. Notice that the nlogn algorithm runs in 20 seconds whereas the same takes days in case of quadratic function and years for polynomial function.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 hours	31,710 years	very long	very long	very long

Look at the below image and see how each function grows with respect to others. To do this comparison, take a value of n say, 50 and compare each colored line's y-coordinate's N values.



Did you notice that log n grows very slowly in contrast to n, which grows linearly and n! and 2^n have exponential growth, which means they grow very rapidly?

Worst case, best case, average-case efficiencies

Is it possible to have different running times for same input n? Let's see.

Consider LinearSearch Algorithm code that we have seen before.

```
LinearSearch(A[0...n-1],k)
    i = 0
    while i < n and A[i] != k
        i = i+1
    if i<n return "Key found"
    else return "Key not found"
```

What is running time? T(n) = n

Does its running time change for different lists of size n? Yes

The LinearSearch searches for a given item (some search key k) in a list of n elements by checking successive elements in the list until either a match with the search key is found or the list is exhausted. The running time of this algorithm can be quite different for different lists of same size, n.

In the worst case, there would be no matching element and the algorithm would run till the end of the list, hence T(n) would be n. It is the largest possible running tie of an algorithm. It generally captures efficiency.

In the best case, the first element itself might be the element that is being searched. Then T(n) would be 1. Best case running time is the minimum possible running time for the input of size n.

In the average case, the search element could be present anywhere in the list, and to calculate T(n) we need to use probabilistic analysis by randomly choosing a key from the elements of the list. Finding the average case is considerably more difficult than finding the worst case or best case.

Hence, there are many algorithms for which time depends not only on an input size but also on the specifics of a particular input.

Exploration: Asymptotic Notations

Introduction

You would hear the word 'asymptotic' very often when studying any text on algorithms. The dictionary definition of 'asymptotic' is a line that approaches a curve but never touches. In analyzing the efficiency of an algorithm, we are concerned with how its running time increases as the size of the input increases without bound. To compare efficiency, computer scientists use three notations: O (big oh), omega (big omega), and theta (big theta).

In this section, we will define these asymptotic notations and use them with non-recursive algorithms.

In the following reading, T(n) and g(n) can be any nonnegative functions defined on positive integers.

T(n) will be the algorithm's running time.

g(n) will be some simple function that will be used to compare T(n) with.

O Notation (Big O)

Informally, O(g(n)) is a set of all functions with a lower or same order of growth as g(n) (within a constant multiple, as n goes to infinity).

We say a function T(n) "is O(g(n))" if T(n) is an element of O(g(n)).

For example:

1. n is an element of O(n^2) and 5n + 100 is an element of O(n^2). Here n is a linear function hence it has a lower order of growth than g(n) = n^2.
2. 3n(n-1) is an element of O(n^2). Here 3n(n-1) is a quadratic function and it has the same order of growth as n^2).
3. But 3n(n-1) is not an element of O(n). Here 3n(n-1) has a higher order of growth compared to n.

Formal Definition

A function T(n) is said to be in O(g(n)), denoted by T(n) is an element of O(g(n)), if $T(n) \leq c(g(n))$ for all $n \geq n_0$, for some positive constant c.

Graphically

How to prove Big O bounds for a function

Let us prove 5n + 100 is an element of O(n^2)

To prove this we have to play around the expression and select the values of c and n_0

Here g(n) = n^2, f(n) = 5n + 100.

To prove big O, we want to get f(n) ≤ c*g(n); for this let us try to get n^2 term into the equation while maintaining ≤ inequality. How about use the terms (5 and 100) from f(n) along with n^2 as shown below.

It is true that, 5n + 100 ≤ 5n^2 + 100n^2 (this inequality is true for all n ≥ 1)

$\Longrightarrow 5n + 100 \leq 105n^2$ (for all n ≥ 1)

We got the equations in the form of f(n) ≤ c*g(n). So, we can select c = 105 and n_0 = 1

Note that this is not the only solution, you can solve it in different ways.

Prove that 3n^2 + 10n ∈ O(n^2).

f(n) = 3n^2 + 10n g(n) = n^2

3n^2 + 10n ≤ c*n^2 n ≥ n_0

3n^2 + 10n ≤ 3n^2 + 10n^2 for n ≥ 1

3n^2 + 10n ≤ 13n^2 when n ≥ 0 therefore c = 13 and n_0 = 1

Observe how we are building f(n) and g(n) relation using the terms of f(n)

Consider

$n \leq n^2$ for all n ≥ 1 (obvious)

$10n \leq 10n^2$ for all n ≥ 1 (multiply both sides by 10)

$3n^2 + 10n \leq 3n^2 + 10n^2$ for all n ≥ 1 (add $3n^2$ on both sides)

$3n^2 + 10n \leq 13n^2$ for all n ≥ 1 (Simplifying)

Hence we can choose c = 13, $n_0 = 1$

$3n^2 + 10n \leq cn^2$

$3n^2 + 10n \in O(n^2)$

By definition of Big-O, this proves that $3n^2 + 10n \in O(n^2)$.

Omega

Informally, Omega(g(n)) is a set of all functions with a higher or same order of growth as g(n) (within a constant multiple, as n goes to infinity).

We say a function T(n) “is O(g(n))” if T(n) is an element of O(g(n)).

Formal Definition

A function T(n) is said to be in Omega(g(n)), denoted by T(n) is an element of Omega(g(n)), if T(n) ≥ c*g(n)) for all n ≥ n_0, for some positive constant c.

Graphically

How to Prove Big Omega bounds of a function

Let us prove 5n^2 is Omega(n)

To prove this, we have to select the values of c and n_0. Similar to Big-O proof we will use the terms in f(n) to build the relation between f(n) and g(n).

$5n^2 \geq 5n$ for all $n \geq 1$ (obvious)

We can select c = 5, n_0 = 1

The below video explains another example.

Prove $2n^3 - 7n + 1 \in \Omega(n^3)$

Prove 2n^3 – 7n + 1 >= c*n^3 for n > n_0

(n^3 + 1) + (n^3 -7n) >= n^3 + 1 when n^3 – 7n >= 0 only occurs when n(n^2 – 7) n>= 0 n >= sqrt(7)

(n^3 + 1) + (n^3 -7n) >= n^3 when n >= sqrt(7). Therefore c = 1 and n_0 = sqrt(7)

Observe how f(n) is being transformed to obtain g(n).

$$2n^3-7n+1=n^3+(n^3-7n)+1 2n^3-7n+1=n^3+(n^3-7n)+1$$
$$\Longrightarrow\Longrightarrow 2n^3-7n+1\geq n^3+1 for all n\geq 3 2n^3-7n+1\geq n^3+1 for all n\geq 3$$
$$\Longrightarrow\Longrightarrow 2n^3-7n+1\geq 2n^3 for all n\geq 3 2n^3-7n+1\geq 2n^3 for all n\geq 3$$

$c = 1, n_0 = 3$

Wondering how we get $n \geq 3 n\geq 3$?
In $2n^3-7n+1=n^3+(n^3-7n)+1 2n^3-7n+1=n^3+(n^3-7n)+1$ we want to make $(n^3-7n)\geq 0 (n^3-7n)\geq 0$ to maintain our inequality in the equations that we wrote.
$$\Longrightarrow\Longrightarrow n^2\geq 7 n^2\geq 7$$
$$\Longrightarrow\Longrightarrow n\geq 3$$

Theta Notation

Informally, theta(g(n)) is a set of all functions that have the same order of growth as g(n) (bounded both above and below by some positive constant multiples of g(n), as n goes to infinity)

We say a function T(n) “is theta(g(n))” if T(n) is an element of theta(g(n)).

Formal Definition

A function T(n) is said to be in theta(g(n)), denoted by T(n) is an element of O(g(n)), if c_1*g(n) <= T(n) <= c_2*g(n) for all n >= n_0, for some positive constants c_1 and c_2.

Graphically

How to prove Big Theta bounds for a function

Let us prove (n + 5 sqrt(n)) is an element of theta (n)

First, we will prove the right inequality (upper bound). Observe how f(n) is being transformed into g(n) satisfying <= inequality.

(n + 5 sqrt(n)) <= (n + 5 sqrt(n)) for all n >= 0

(n + 5 sqrt(n)) <= (n + 5n) for all n >= 0 (as 5 sqrt(n) <= 5n)

(n + 5 sqrt(n)) <= 6n for all n >= 0

Second, we prove the left inequality (lower bound).

n <= (n + 5 sqrt(n)) for all n >= 0

Hence we have n <= (n + 5sqrt(n) <= 6n for all n >= 0

We can select c1 = 1, c2 = 6, and n0 = 0

Question

Prove ½n(n-1) is an element of O(n^2).

c1*n^2 <= ½n(n-1) <= c2*n^2 when n >= n_0 c2 = 1/2 c1 = ¼ n_0 = 2

You don't have to worry about the proving the bounds of a function, the above examples are providing to give an idea about how it is done. Observe how the graphs of each of the notations differ from one another.

Comparing Orders of Growth

Though we could use the formal definitions of O, omega, and theta notation proofs that we discussed above to compare the order of growth of functions, a more convenient method would be to compute the limit of the ratio of the two functions in the question.

$$\lim_{n\to\infty} \frac{T(n)}{g(n)} = \begin{cases} 0 & \text{implies that } T(n) \text{ has a smaller order of growth than } g(n). \quad T(n) \in O(g(n)) \\ C & \text{implies that } T(n) \text{ has the same order of growth as } g(n). \quad T(n) \in \theta(g(n)) \\ \infty & \text{implies that } T(n) \text{ has a larger order of growth than } g(n). \quad T(n) \in \Omega(g(n)) \end{cases}$$

The limit based approach is convenient because it takes benefit of the power of calculus techniques.

The below listed two rules might be helpful to calculate limits. We will not go further into math than this.

L'Hopital's Rule:

lim n->inf T(n)/g(n) = lim n->inf T'(n)/g'(n)

T'(n) is derivative of T(n)

Just refresher for you on derivatives.

If T(n) = n^5 then T'(n) = 5n^4

If T(n) = log_2(n) then T'(n) = 1/(nlog_e(2)

Stirling's formula:

n! ≈ sqrt(2*pi*n)((n/e)^n) for large values of n

Example 1

Compare the order of growth of 3n^2 + 5n + 20 and n^2

lim n->inf (3n^2 + 5n + 20)/(n^2) = lim n->inf (3 + 5/n + 20/n^2) = 3

Since, the limit is equal to a positive constant, the functions have the same order of growth, 3n^2+5n+20 is an element of O(n^2)

Example 2

Show that 5n^2 + n is an element of omega(n)

lim n->inf (5n^2 + n)/n = lim n->inf 5n + 1 = inf

This implies 5n^2 + n has a bigger order of growth than n, hence 5n^2 + n is an element of omega(n)

Example 3

Compare the order of growth of functions log_2 n and sqrt(n)

Hint: use L'Hopital's Rule

lim n->inf log_2 n / sqrt(n) = lim n->inf (log_2 n)'/(sqrt(n))'

= lim n->inf log_2 e*(1/n)/(1/(2sqrt(n))) [since 1/log_e 2 = log_2 e]

= log_2 e * lim n->inf 2 sqrt(n)/(sqrt(n)*sqrt(n)) [since n= sqrt(n) * sqrt(n)]

= 2 log_2 e*lim n->inf 1/sqrt(n)

= 2 log_2 e * 0 [since liim n->inf 1/sqrt(n) = 0]

= 0

Since the limit is equal to 0, log_2 n has a lower order of growth than sqrt(n).

Properties Involving Asymptotic Notations

Transitivity

$$f(n) = \theta(g(n)) \text{ and } g(n) = \theta(h(n)) \text{ imply } f(n) = \theta(h(n))$$

$$f(n) = O(g(n)) \text{ and } g(n) = O(h(n)) \text{ imply } f(n) = O(h(n))$$

$$f(n) = \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) \text{ imply } f(n) = \Omega(h(n))$$

Reflexivity

$$f(n) = \theta(f(n))$$

$$f(n) = O(f(n))$$

$$f(n) = \Omega(f(n))$$

Symmetry

$$f(n) = \theta(g(n)) \text{ if and only if } g(n) = \theta(f(n))$$

Transpose Symmetry

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n))$$

Exploration: Mathematical Analysis of Algorithms

Introduction

In this exploration, we will use the notations that we have seen in the previous exploration. By the end of the section, you would be able to analyze the time complexity of simple non-recursive algorithms.

Analysis of Non-Recursive Algorithms

Let us consider an example of finding the maximum number in an array of size n and arrive at the general steps to find the time complexity mathematically.

```
FindMax(Arr):
    max = Arr[0]
    for i = 1 to n-1
        if Arr[i] > max
            max = Arr[i]
    return max
```

What is the parameter that would impact the execution time of this algorithm? It is array size n.

What is the operation that would be executed most often in this algorithm? The instructions in the loop 'if Arr[i] > max' and 'max = Arr[i]'. Among these the comparison would execute for each iteration of the

loop and the assignment will be executed only when the comparison is true. Hence we can say 'if Arr[i] > max' is the most executed operation in the algorithm. In identifying the basic operation, why do we not consider the for loop, which is executed more number of times? This is answered towards the end of the exploration.

Would the number of comparisons vary for different arrays of size n? No, the number of comparisons will be the same for all arrays of size of n. Hence, we need not distinguish between worst case, best case and the average case for this algorithm.

Let's see the number of the times the comparison is executed, and find its expression as a function of its input parameter n. The algorithm makes one comparison for each iteration of the for loop, which repeats from 1 to n-1. Hence we can write our expression as.

T(n) = $\sum_{i=1}^{n-1} 1$

This is easy to compute as it is 1 repeatedly added n-1 times.

T(n) = $\sum_{i=1}^{n-1} 1$ = n -1 is an element of theta(n)

These are the general steps that we followed:

Step 1: Identify the input parameter(s) that would impact the running time of the algorithm.

Step 2: Identify the **basic** operation, that which would be executed a maximum number of times and impacts the execution time of the algorithm (this is usually located in the innermost loop)

Step 3: Determine worst, average, and best cases for the input of size n, if the number of times the basic operation gets executed varies with specific instances of input.

Step 4: Set up a sum for the number of times the basic operation is executed and simplify it using standard summation formulas.

Let's see one more example:

Consider an algorithm that checks whether the elements in an array are distinct or not.

```
IsUnique(A[0..n – 1]):
    for i =0 to n – 2
        for j = i + 1 to n – 1
            if A[i]= A[j]
                return false
    return true
```

What is the parameter that would impact the execution time of this algorithm? Obviously the array size n.

What is the basic operation that would be executed most often in this algorithm? The comparison instruction 'if A[i] = A[j]' in the innermost for loop.

Would the number of times the basic operation is executed vary for different arrays of a given size n? Yes. There can be a scenario for a given n when the loop would exit prematurely without completing all

the iterations of the loop. This would happen when the first two elements are equal (ex: [3,3,1,4,7]). This would be the best case of execution. There can also be a scenario for the same size of array when the loops would run until their maximum bounds (ex: [2,3,1,4,6]). This would be the worst case of execution as the algorithm would take maximum time to give us results.

Let us set up the summation for the worst case of execution. For each value I of the outer loop, inner loop runs between limit i+1 to n-1. This is repeated for all values of I from 0 to n-2.

$$T_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

Inner summations can be simplified as:

$\sum_{j=i+1}^{n-1} 1 = \sum_{j=1}^{n-1} 1 - \sum_{j=1}^{i} 1 = n - i - 1$; in other words adding 1 a total of $[n-1 - (i+1) + 1]$ times

$$\begin{aligned} T_{worst}(n) &= \sum_{i=0}^{n-2} [(n-i-1)] \\ &= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i \\ &= (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i \\ &= (n-1)^2 - \frac{(n-2)(n-1)}{2} \\ &= \frac{(n-1)n}{2} \\ &\approx \frac{1}{2} n^2 \\ &\in \theta(n^2) \end{aligned}$$

Exploration: Proving Correctness of an Algorithm

Introduction

An important aspect of an algorithm is that it produces correct output over a broad range of inputs. We might run our algorithm for some sample input values and verify its correctness, but this ensures correctness for the values that we have tested not for all possible inputs. At the end of this section, you will be able to formally prove the correctness of an algorithm.

As you read this section consider answering the following questions:

1. What are the general steps to formally prove the correctness of an algorithm?

2. What is a loop invariant?
3. How to identify a loop invariant?

Verifying an algorithm

The first thing to do when we write or encounter a new algorithm is to convince ourselves that it is correct, which means it produces correct output for all valid inputs.

To get some insights into empirically verifying correctness by taking some sample inputs and executing the algorithm read 'empirical analysis section' in the khanAcademy site.

Empirical analysis cannot assure the correctness for all range of inputs that we have not tried, so formally we could use mathematical reasoning over all possible inputs.

Intuitively, to prove that a loop executes correctly, we want to show that it works for the first iteration, then for the second iteration and so on until the last iteration. Additionally, we want to show that the loop terminates. This idea is similar to 'mathematical induction'.

There are two sets to prove the correctness of an algorithm:

1. We will prove using induction that a "loop invariant", a property of the loop in the algorithm is always true.
2. We will show that the algorithm eventually terminates.

Loop Invariant

The word 'invariant' means 'never changing or something that is true always'. A loop invariant is some condition that holds true for every iteration of the loop.

For example, consider the pseudocode to find the maximum number in a list of numbers.

```
def findMax(list):
 max = list[0]
 for i from 0 to len(list)-1:
  if list[i] > max:
   max = list[i]
 return max
```

Here the loop invariant is: the variable 'max' always holds the maximum number among the first I elements of the list.

You can use these general tips to find a loop invariant.

- Use your intuition and try to understand what the code does.
- In a list, a loop invariant usually speaks about the part of the list that has been processed so far in the loop.
- A loop invariant is true even if the loop condition is false i.e. before the beginning of the loop and when the loop terminates.

Common mistake is to describe the code in the loop and state it as loop invariant. For findMax function it would be incorrect to state the loop invariant as "If i^th element is greater than max then max is

assigned with the i th element": although technically it might be true but we are looking for a property of the loop that could be a loop invariant.

Week 2

Introduction

In this module, we will talk about implementing a recursive algorithm. We will look at the techniques to solve recurrence relations, this will help us analyze the time complexity of recursive algorithms.

Then we will look at a commonly used strategy of solving recursive algorithms called 'Divide and Conquer'. Divide and conquer is a very popular algorithmic paradigm, once you have mastered this you will be able to reduce the time complexity of various algorithmic problems drastically and write efficient programs. At the end of the module, you will be able to implement algorithms using the divide-and-conquer technique.

Exploration: Recursion

Introduction

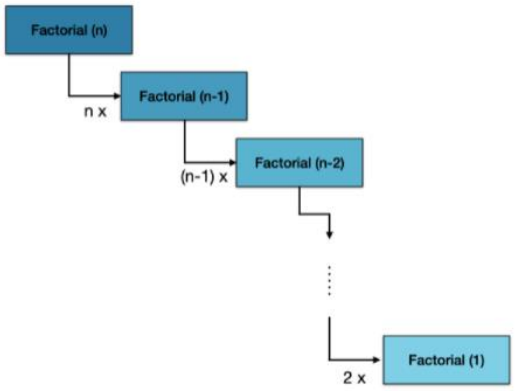
Recursion is a helpful algorithmic technique that would help us to solve specific kind of algorithm problems. In this section, we will see when to use recursion and how to write a recursive algorithm. You might already have some idea about recursion, try to see if you can build the intuition of how to solve a problem using recursion.

Recursion

When solving a problem, if you see that the problem can be solved by breaking the problem into smaller problems of the same form, then you can use recursion to solve it. For instance, take the problem of computing factorial. The factorial of a number n is given by:

Factorial of n = n * (n-1) * (n-2) … 3 * 2 * 1

When we want to find the factorial of a number n (Factorial(n)), we can compute it easily if we have a factorial of (n-1), (Factorial(n-1); which we can compute if we have factorial of (n-1)-1, Factorial(n-2), and so on. The figure below shows it diagrammatically.



Note that we expressed Factorial function F in terms of itself. This is called a recurrence relation. A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Remember this world (recurrence relation), we will visit it again. Recurrence relations of recursive algorithms help us analyze the running time of an algorithm.

Two important parameters to consider when writing recursive programs:

1. You must have a base case (the simplest form of the problem that you are attempting to solve and which can be answered directly)
2. You can break the problem into sub-problems that solves the smaller instances of the big problem and these sub-problems should be ultimately lead to the base case.

Our pseudocode for the factorial function can be:

```
1 Factorial(n):
2    if n = 0:
3        return 1
4    else:
5        return n * Factorial(n-1)
```

You can watch the animation of the execution of recursion in the link provided by USFCA. Each recursive call is a frame that is put on the execution call stack.

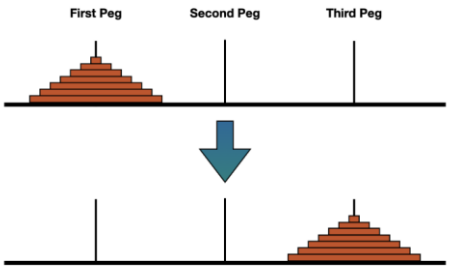
The general strategy that you could follow to solve problems using recursion is to ask the following questions:

1. How can I reduce the problem into smaller versions of the same problem?
2. What is the base case?
3. Will I always reach the base case?

Let us look at one more example

Towers of Hanoi

The Towers of Hanoi is a mathematical puzzle, where we have three pegs. The first peg has n number of disks of different sizes that can slide on any on the three pegs, but the condition is that we cannot place on a larger disk on a smaller disk. The goal is to move all the disks from the first peg to the third peg, using the second peg as a temporary peg.



The (8-disk) Tower of Hanoi Puzzle

To get the recurrence relation, T(n), for the Towers of Hanoi problem we need to find the time taken to move all the n disks.

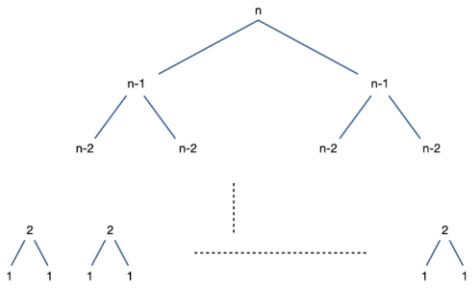
To calculate the time taken to move n disks, we have to consider:

- The time taken to move n-1 disks from the first peg to the second peg (this takes T(n-1) time).
- Then, the time taken to move the last disk to the third peg (this takes constant time as it is one move).
- Finally, the time taken to move the n-1 disks from the second peg to the third peg (this takes T(n-1) time).

Combining these we get T(n) = T(n-1) + 1 + T(n-1) for n > 1.

Obviously, the best case will be when n = 1, i.e. when we have one disk. T(1) = 1.

We can draw a tree to represent this recurrence relation, which represents the recursive calls made by the algorithm. To calculate T(n), we need to make calls to T(n-1) two times. To calculate T(n-1), we call T(n-2) two times. The tree looks like this:



The tree is called a recursion tree (remember this term). It can be used to analyze the time complexity of an algorithm. We will see this in the future explorations.

Your function definition:

```
hanoi(n, source, temp, target)
```

The function returns: source, temp, target

Sample input: hanoi(5,[5,4,3,2,1),[],[])

Output: [], [], [5,4,3,2,1]

Explanation first peg is source, second peg is temp, third peg is target, the disks got moved from source to target.

Exploration: Recurrence Relations

Introduction

In the previous exploration, we have seen that we can express a recursive solution in terms of a recurrence relation. In this section, we will dive deeper into recurrence relations and look at how to solve them. As you master this you will begin to feel comfortable analyzing any complex recursive algorithm that we will see further in this course and that which you would come across outside academics.

Recurrence Relations

To analyze a recursive algorithm, similar to an iterative algorithm, we need to first express the algorithm in terms of an equation that expresses how the algorithm grows and then solve it to obtain the order of the growth.

An equation that expresses how a recursive algorithm grows is called a recurrence relation. Recurrence relation, as we have seen before, is an equation that is expressed in terms of its smaller values.

Example 1:

Let us write recurrence relation for the Factorial function.

```
Factorial(n):
    if(n==0):
        return 1
    else:
        return n * Factorial(n-1)
```

The if condition code runs in a constant time (say, c).

If we represent the recursive function's execution by T(n), the recursive call in the else condition makes a subsequent call on T(n-1).

The recurrence relation can be written as:

T(n) = c_1 or theta(1) when n = 0 [This represents base case, and c_1 is some constant]

T(n) = T(n-1) + c when n > 0 [This represents recursive case]

or, the recursive case can also be written as:

T(n) = T(n-1) + theta(1) when n > 0 [since, theta[1] represents a constant time]

Example 2:

Let us write the recurrence relation of the Tower's of Hanoi code

```
def hanoi(n, source, temp, target):
    if n > 0:
        # move n-1 disks from source to temp:
        hanoi(n - 1, source, target, temp)
        # move disk from source peg to target peg
```

```
        if source:
           target.append(source.pop())
        # move n-1 disks from temp to target
        hanoi(n - 1, temp, source, target)
```

The if condition is executed in a constant time, c_1.

The first recursive call 'hanoi(n-1, source, target, temp)' makes a recursive call on T(n – 1)

Then there is a constant time (c2) check by the if condition 'if source:'; followed by a second recursive call on T(n-1)

Note that in the recursive of the Hanoi function although there are other parameters 'temp', 'source', 'target', we don't have to present them in the recurrence relation as they don't impact the running time of our algorithm

The recurrence relation will be:

T(n) = c1 + T(n-1) + c2 + T(n-1)

T(n) = 2T(n-1) + c [for n > 0, this is for the recursive case]

T(n) = c [for n = 0, this is for the base case]

Example 3:

Let's write recurrence relation for the below pseudocode of a foo function. Don't worry about what the algorithm does you will understand it as you read further.

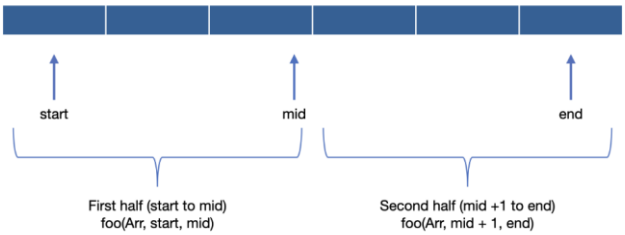
```
#Arr is an array of size n, start and end point to starting index and last index of t
he array
foo(Arr,start,end):
    if(start<end):
        mid = (start+end)//2 #Computes floor of middle value

        foo(Arr,start,mid)
        foo(Arr,mid+1,end)

        #Let foo2 be some function that has Theta of n time complexity
        foo2()
```

foo is some function that takes as input: an array 'Arr' of size n, starting index of the array 'start' and last index of the array 'end'. We need to find the time taken by the foo function, T(n). T(n) is a function of its input parameter, the size of the array.

If the start value is less than the end value we calculate the middle position ('mid') between 'start' and 'end' values. Calculation of 'mid' value takes a constant amount of time and is not dependent on size of the array, n. So, the cost of execution of this line of code can be some constant, say c1. Similarly, the if condition execution will take constant time O(1).

In the 'if' condition, we call the foo function recursively. In the first call, we pass 'start' and 'mid' of the array as the function parameters, so in this call, the function works only on n/2 elements (first half of

the array). In the second call, we pass 'mid + 1' and 'end' as parameters hence in this call, the foo function will work on the second half of the array, hence n/2 elements. We will not worry if n is odd, for ease of analysis we shall take n/2 (this is acceptable as we are analyzing for very large values of n. Is that not what asymptotic analysis all about?!)



Each call on size n/2 will take T(n/2) time. Hence, a total of 2T(n/2) time will be taken for the two calls.

Say, foo2 is some function takes O(n) time.

```
foo(Arr,start,end):
    if(start<end):
        mid = (start+end)//2 #Computes floor of mid value

        foo(Arr,start,mid)
        foo(Arr,mid+1,end)

        #Let foo2 be some function that has Theta of n time complexity
        foo2()
```

→ Constant Time
→ Constant Time
→ $T\left(\frac{n}{2}\right)$
→ $T\left(\frac{n}{2}\right)$
→ $\Theta(n)$

The if-statement takes constant time because it is just checking a condition. It takes constant time, constants are in order of 1, i.e., O(1) or better to say O(1).

Then the equation that represents the time taken by the recursive part of the foo function can be written as:

T(n) = 2T(n/2) + O(n) + O(1)

The last O(1) can be neglected.

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

So, we will get the recurrence relation:

$$T(n) = \begin{cases} c_1 & \text{for n=1 (when start = end)} \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{for n>1} \end{cases}$$

O(n) represents a linear time complexity, we can replace it with cn where c is some constant (since cn is tightly bound over O(n)). We are making this replacement so that we don't have anything in terms of 'Theta'. Our recurrence relation will then be:

$$T(n) = \begin{cases} c_1 & \text{for n=1 (when start = end)} \\ 2T\left(\frac{n}{2}\right) + cn & \text{for n>1} \end{cases}$$

Let's look at more examples. You may try them for yourself before looking at the explanations.

Example 1: Let us find the recurrence relation for the foo1 function.

```
#Arr is an array of size n, start and end point to starting index and last index of t
he array
#key an arbitrary value
foo1(Arr, start, end, key):
    if(start > end):
        return False
    else:
        mid=(start+end)//2
        if(Arr[mid] == key):
            return True
        if (Arr[mid]>key):
            return foo1(Arr,start,mid, key)
        if(Arr[mid]<key):
            return foo1(Arr, mid+1, end, key)

        #Let foo2 be some function that has Theta of n time complexity
        foo2()
```

It is similar to the previous algorithm. When performing the comparison and calculating the mid-value it takes constant time, O(1). Time taken by foo2 can be represented as O(n). When foo1 is called over the first half and second half of the array it takes T(n/2) time. Hence, its recurrence relation will look as shown below. In the recursive case, we can ignore O(1) as it has a lower order of growth compared to O(n). That would also be a correct recurrence relation.

$$T(n) = \begin{cases} \Theta(1) & \text{for n=1 (when start = end)} \\ T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(1) & \text{for n>1} \end{cases}$$

O(1) represents a constant time, we can replace it with c.

We can replace O(n) with n, as n has strictly O(n) order of growth. Our recurrence relation modifies to:

$$T(n) = \begin{cases} c & \text{for n=1 (when start = end)} \\ T\left(\frac{n}{2}\right) + n & \text{for n>1} \end{cases}$$

Example 2:

```
foo(n):
    if(n=1):
        return 1
    else:
        return foo(n/2)*foo(n/2)
```

The if condition check and the return statement runs in a constant time c1.

When the else statement is executed foo(n/2) would take T(n/2) execution time; this call is made twice hence a total of 2T(n/2) for the return statement. Additionally, a constant time c2 to check if the 'if' condition fails and then reach the else clause.

The recurrence relation will be:

T(n) = 2T(n/2) + c2 for n > 1

T(n) = c1 for n = 1

Exploration: Substitution Method

Introduction

In this section we see how to solve the recurrence relation using the substitution method.

Substitution Method

The idea behind the substitution method is that we want to solve the recurrence relation by substitution the smaller values of the recurrence relation in the equation. For example, if the recurrence relation has T(n/2) we will find the equation of T(n/2) from T(n) and substitute it back in the recurrence relation. We will go on doing this until we reach the base case, where the recurrence stops.

Let's try to solve the below recurrence relation. For a better learning experience, try to follow the steps using a pen and paper.

$$T(n) = \begin{cases} c_1 & \text{for n=1} \\ T\left(\frac{n}{2}\right) + c & \text{for n>1} \end{cases}$$

First, we will take the expression of T(n)

$T(n) = T(n/2) + c$ –> Eq1

We have to substitute the value of T(n/2) in terms of its smaller values.

Substituting n -> (n/2) in Eq1, we get

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

$T(n/2) = T(n/4) + c$

Substituting T(n/2) in Eq1

$T(n) = T(n/4) + c + c$

$T(n) = T(n/4) + 2c$ -> Eq2

To solve this further we need to substitute T(n/4) in Eq2.

To find T(n/4), substitute n - > (n/4) in Eq1. We get

$T(n/4) = T(n/8) + c$

Substituting T(n/4) in Eq2, we get:

$T(n) = T(n/8) + 3c$ -> Eq3

Do you notice a pattern of how numbers are increasing in the equation?

T(n) equation was Eq1: $T(n) = T(n/2) + c$

Eq2 can be written as: $T(n) = T(n/(2)^2) + 2c$

Eq3 can be written as: $T(n) = T(n/(2)^3) + 3c$

We will go on substituting the smaller values of T on the right-hand side of the equation until we reach the base case, i.e. when the recurrence terminates, which is T(1) here.

Let's say at Kth equation we reach the base case. By observing the pattern of Eq 1,2,3, we can write our kth equation as,

$T(n) = T(n/2^k) + kc$ -> kth Eq

Since we arrived at the base case (T(1)) in the kth equation, we can say

$T(n/2^k) = T(1)$ -> Eq*

Substituting (Eq*) in the (kth Eq) modifies as:

$T(n) = T(1) + kc$

From the base case, we know

$T(1) = c1$

Substituting this in our modified kth equation: T(n) = c1 + kc

Now, our equation does not have any smaller T(n) terms in this equation

We just need to get rid of k by expressing it in terms of n.

From our (Eq*) we can say (n/2^k) = 1

- ➔ n = 2^k
- ➔ log2n = log2 2^k
- ➔ log2n = klog2 2
- ➔ log2n = k (since, log2 2 = 1)
- ➔ k = log2n

Substituting this for the value of k in the kth equation:

T(n) = c1 + clog2n

Viola, we solved our T(n)

Our final solved equation for T(n)

T(n) = clog2n + c1 where c1 and c are some constants

Now from the previous knowledge of asymptotic analysis can you tell the time complexity of T(n)? Since, the highest order term is log n, time complexity will be theta(log n).

We will skip mentioning the base of the log since most of the times we will be dealing with the logarithmic notation of base 2. If that is not the case we will write it.

Let's revisit what we just did, we took our recurrence relation T(n) and substituted the smaller values of itself

(n/2 -> n/4 -> n/8 -> …. -> k) by using the T(n) equation. After a couple of substitutions, we could find the pattern of how the question is changing with each substitution. Using this pattern we wrote the equation for the kth substitution, where our recurrence relation reaches the base case. Then, we used the base case equation to solve for the value of k. Hence, we obtained our T(n) purely in terms of n, devoid of any k's or smaller values of T(n). Using this solved recurrence equation we could tell the time complexity.

Exploration: Recursion-Tree method

Introduction

In this section, we see how to solve recurrence relations using the recursion-tree method

Recursion-Tree

When a recursive algorithm makes more than a single call to itself, analysis can be done by constructing a tree of the recursive calls. It is just a visual representation of recursion with each node of the tree represents the cost of a certain recursive sub-problem. This tree is called a Recursion-Tree. Summing up the value of the nodes would give us the cost of the entire algorithm. This method is sometimes

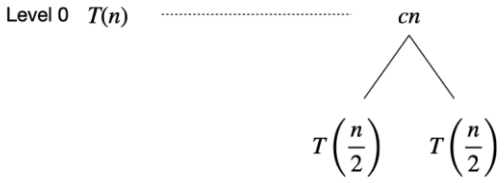
unreliable but it gives us good intuition and a guess for the time complexity. Accurate time complexity can be found by using the substitution method.

Problem 1:

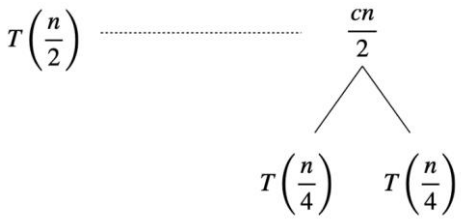
Let us find the time complexity of the following recurrence relation that we have seen in previous explorations.

$$T(n) = \begin{cases} c_1 & \text{for n=1 (when start = end)} \\ 2T\left(\frac{n}{2}\right) + cn & \text{for n>1} \end{cases}$$

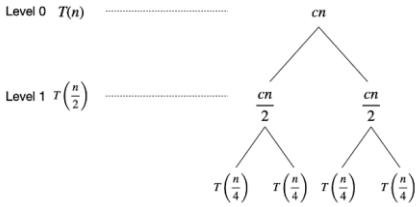
Looking at the recurrence equation T(n) = 2T(n/2) + cn, we can say that when T(n) is executed it spends cn cost of execution for the current execution and then makes two recursive calls expressed by T(n/2). We can represent this in the tree form as:



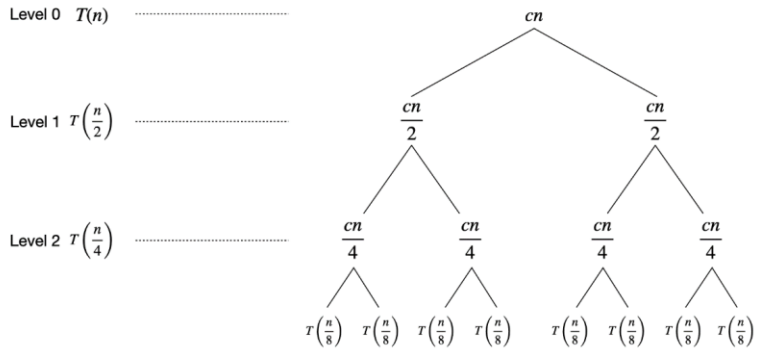
From recurrence relation, we can find out the expansion of T(n/2). It can be written as T(n/2) = 2T(n/2) +c(n/2). This can be represented in tree form as:



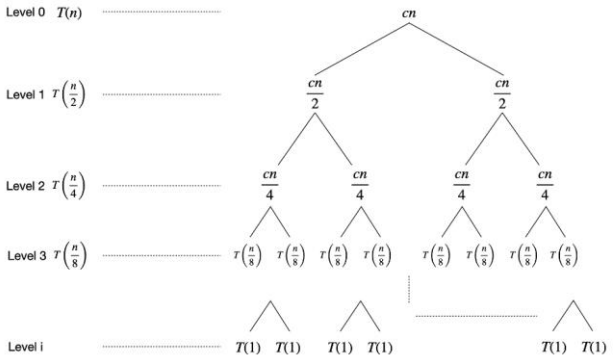
We can use this T(n/2) tree in T(n) tree, to build our recursion tree.



Similarly, we can expand the tree by expanding T(n/4).



Do you get an idea now? So, this way we can build our tree by expanding each recursive call until we reach the base case. We need not do that but that is the idea. So our tree would be:



To find the cost of the whole tree, we will find the cost at each level. For this, we need to observe how many nodes are present at each level. At level 0, there is one node; at level 2 there are two nodes; at level 3 there are eight nodes and so on. Do you see a pattern here? The nodes are increasing by a factor of two. (2^0, 2^1,2^2,2^3,...). This means at ith level there would be 2^i nodes.

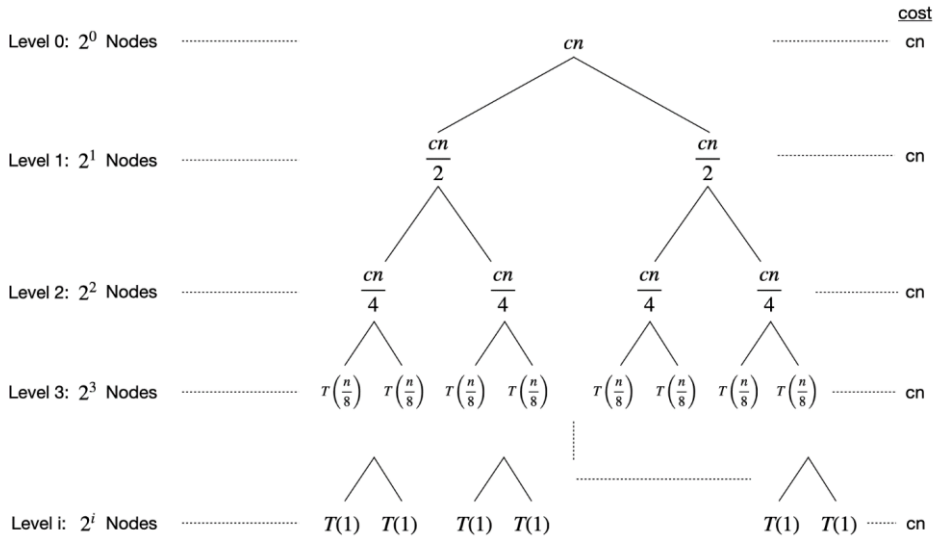
Calculating the cost at each level:

At level 0: cn

At level 1: cn/2 + cn/2 = cn

At level 2: cn/4 + cn/4 +cn/4 + cn/4 = cn

We can find a pattern here as well, the cost at each level is always cn.



So the cost of the total tree will be cn * number of levels in the tree. How many levels do we have? We can find this from the base case. We can see that

At level 0: tree expanded from T(n) ⟹T(n/2^0);

At level 1: tree expanded from T(n/2) ⟹T(n/2^1);

At level 2: tree expanded from T(n/4) ⟹T(n/2^2);

At level 3: tree expanded from T(n/8) ⟹T(n/2^3);

So, at level i: we can write T(1) to be an expansion of T(n/2^i)

Which gives us n/2i = 1, from this we can solve for i, as we have seen before in the substitution method.

We will get, i = log2n

The total cost of the tree will be = cost at each level * number of levels.

= cn * (i+1)

= cn * (log2n +1)

= cnlog2n + cn

So, the total cost of the recurrence relation is cnlog2n + cn, where c is some constant. Ignoring the lower order terms, the growth function of the algorithm represented by this recurrence relation is Θ(nlogn).

Exploration: The Master Method

Introduction

In this section, we will see how to solve recurrence relations using the master method

The Master Method

The master method enables us to directly find the solution of a recurrence relation of the form T(n)=aT(n/b) + f(n)

where a≥1, b>1 and f(n)>0 (i.e. f(n) is asymptotically positive polynomial function)

For example,

T(n) = T(n/2) + c; here a = 1, b =2 and f(n) = c

T(n) = 3T(n/2)+ cn; here a = 3, b = 2 and f(n) = cn

In this method, n^logb a and f(n) values are compared. As you have seen in the previous module, when we compare two functions there are three possible cases: either n^logb a grows asymptotically at a faster rate than f(n), or n^logb a grows at a slower rate than f(n), or both have same order of growth.

For simplicity consider f(n)∈θ(n^d), so that we can easily compare n^d and n^logb a

Case 1

If $f(n)$ grows asymptotically slower than $n^{\log_b a}$

i.e $n^d <<< n^{\log_b a}$

Then, the solution for our recurrence relation will be

T(n) = Θ($n^{\log_b a}$)

The intuition here is: Since f(n) grows slower than $n^{\log_b a}$, the dominating order of growth will be that of $n^{\log_b a}$.

Case 2

If $f(n)$ and $n^{\log_b a}$ have a similar order of growth.

i.e n^d = Θ($n^{\log_b a}$)

Then, the solution for our recurrence relation will be

T(n) = $\theta\left(n^d \log n\right)$

Case 3

If $f(n)$ grows asymptotically faster than $n^{\log_b a}$

i.e $n^d >>> n^{\log_b a}$

Then, the solution for our recurrence relation will be

T(n) = $\theta\left(n^d\right)$

The intuition here is: The time complexity of f(n) dominates in overall time complexity of recurrence relation.

Let us look at a few examples:

Example 1	Example 2	Example 3	Example 4
$T(n) = 4T(\frac{n}{2}) + n$	$T(n) = 4T(\frac{n}{2}) + n^2$	$T(n) = 4T(\frac{n}{2}) + n^3$	$T(n) = 2T(\frac{n}{2}) + 2^n$
$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n$	$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^2$.	$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2$; $f(n) = n^3$	$a = 2, b = 2 \Rightarrow n^{\log_b a} = n$; $f(n) = 2^n$
This satisfies **Case 1**: $n <<< n^2$.	This satisfies **Case 2**: $n^2 = n^2$	This satisfies **Case 3**: $n^3 >>> n^2$.	Which case does it satisfy? Notice that 2^n is not a polynomial function (a condition to apply Master Method), but is an exponential function. So it does not satisfy the condition for the Master Method, hence, we cannot apply it.
So, $T(n) = \Theta(n^2)$	So, $T(n) = \Theta(n^2 \log n)$.	So, $T(n) = \Theta(n^3)$.	

Limitations of Master Method

As you might have noticed that the recurrence relation should satisfy a few conditions before we could apply the Master Method to solve it. This brings us the limitations of the Master method, where it cannot be applied.

- To apply Master Method the recurrence relation should satisfy a>0, b > 1, and f(n) is a positive function.
- Master Method cannot be applied if f(n) is not a polynomial function, Ex: f(n) = sin n or f(n) = 2^n
- Master Method cannot be applied if a or b cannot be expressed as a constant. Ex: T(n) = 2n T(√n);

Examples:

- T (n) = 64T (n/8) – n^2*log n ⇒ Master Method does not apply (since, f(n) is not positive)
- T (n) = 2nT (n/2) + n^n ⇒ Master Method does not apply (since, a is not constant and f(n) is not a polynomial function)

The Master Method cases that are presented above are very simplified versions. If interested you might want to check the CLRS book for the elaborate versions of these.

Special Case:

Let us look at one final case to look at for completeness. If $f(n) \in \theta\left(n^{\log_b a} \log^k n\right)$ for some $k \geq 0$; then T(n) = $\theta\left(n^{\log_b a} \log^{k+1} n\right)$

Example:

T(n)=2T($\frac{n}{2}$)+n logn; Here f(n) is not a polynomial function. f(n) $\in \theta(nlogn)$.

Here a=2 and b=2. $n^{\log_b a} = n$

This follows the special case, $f(n) \in \theta\left(n^{\log_b a} \log^k n\right)$, where k = 1.

So, T(n) = $\theta\left(n \log^2 n\right)$

Exploration: Divide-and-Conquer Technique

Introduction

When you are given a problem that can be solved using recursion, did you wonder if there is any technique that you can use directly to solve the problem? Well, 'Divide and Conquer' technique is one such common approach that can be used to solve an algorithm recursively. We will explore it in this section and solve a few common problems using this approach.

By the end of this section, you will able to implement a divide-and-conquer algorithm (MLO3).

While going through the exploration try to answer these questions:

What is a divide-and-conquer strategy?

How to approach and solve a problem using the divide-and-conquer strategy?

Divide and Conquer

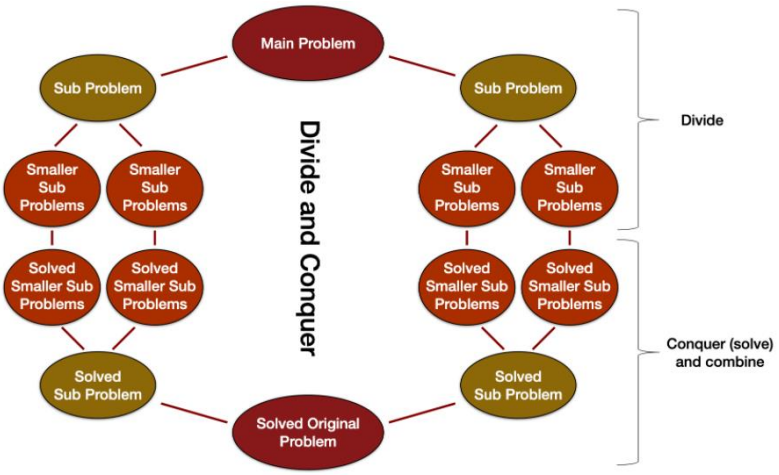
Divide and conquer approach basically divides the problem into smaller subproblems and then conquer (or solve) the subproblems first, then combine the results of the subproblem to get the solution of the original problem.

We follow three steps:

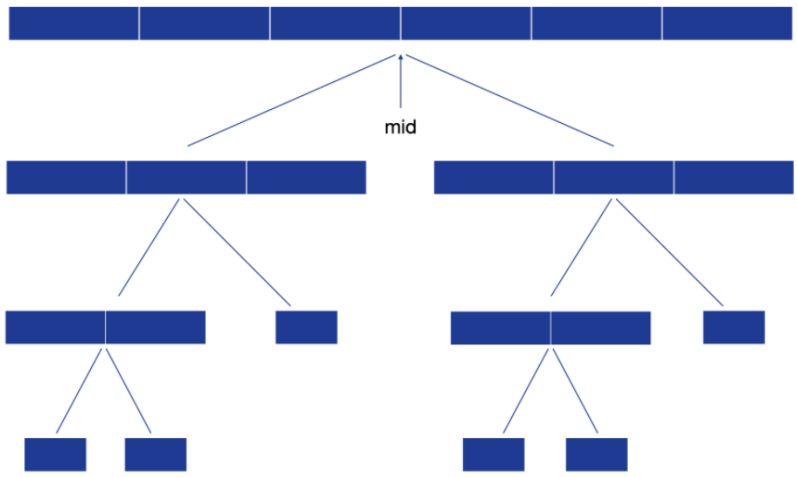
Divide - In the first step we break the problem into smaller subproblems.

Conquer - This step involves solving the subproblems.

Combine - In the last step, we combine the solutions of the subproblems to get the solution for the larger problem.



We will mostly deal with lists/arrays here. When we have an array that we want to Divide-and-Conquer our first task will be to break it into two parts (say two halves). To divide the array into two halves we calculate the middle index. Then for each half, we will find the middle and divide further. This will go on until we cannot divide it any further.



Let us see how to implement this using a sample scenario.

Problem: We need to double the value of each element in a given array.

input: [1,2,3,4,5,6,7]

output: [2,4,6,8,10,12,14]

This is a simple problem, you can do it using a loop. Let's see how to solve this problem using the divide and conquer technique.

Our strategy will be to divide this array into half recursively until we reach a single element, which will be our base case, then double the value of the single element.

To do this we need to keep track of the middle position over each iteration. When we are dividing the array into smaller arrays, we need not create new arrays, it is sufficient if we can keep track of the starting index position and last index position of the sub-arrays. Let's say our recursive function name is 'foo'. Based on our discussion it will need three parameters: foo(arr, start, end)

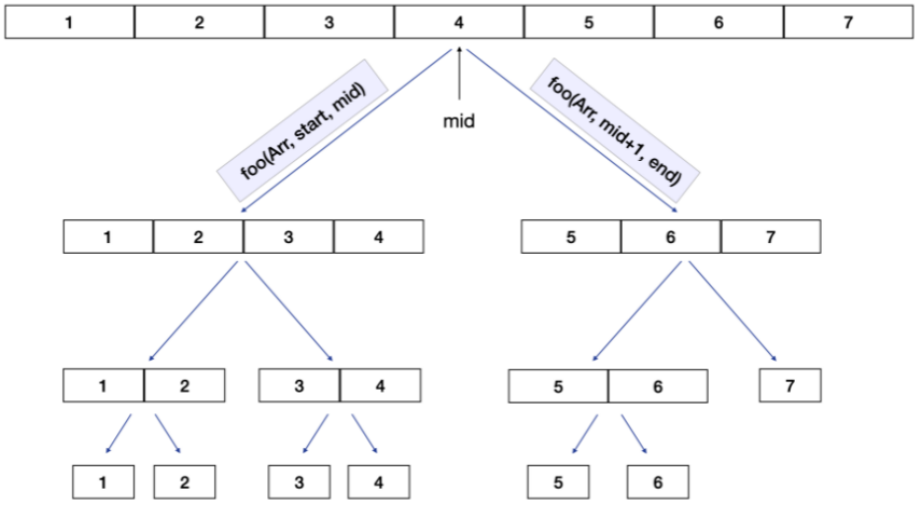
arr: The Array given in the problem

start: The starting index position of the array

end: The last index position of the array

You might think, why not use a helper function? Actually that would be the best approach, but for now, to simplify our code let us consider these parameters to be part of the main 'foo' function itself.

The below figure shows the divide step of the foo function. Note, we are not creating new arrays here.

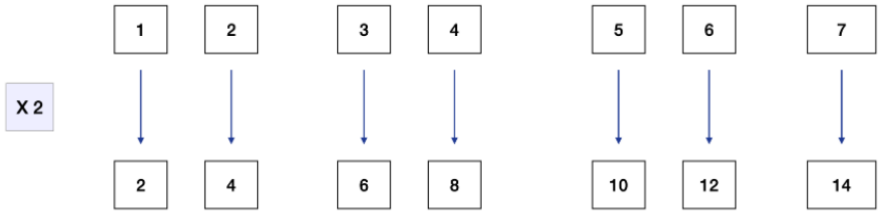


The pseudocode for the divide step will be

```
foo(arr, start, end):
    if(start < end):
        middle = (start + end) // 2
```

```
        foo(arr, start, middle)
        foo(arr, middle + 1, end)
```

Once we have reached the smallest size of our problem (the base case), we can perform the 'conquer' step, i.e solve the smaller sized problem. Now, we can double each item of the array.



The pseudocode for the conquer step will be:

```
foo(arr, start, end):
    if(start < end):
        ...
    else:
        arr[start] = 2 * arr[start]
```

The last step is the combine step, which in this case is automatically done by the recursive function as it returns from the base case.

Below is a graphical representation of how we reach to a single element(1) via recursion. Notice that when we reach the smallest sized array the start index pointer is equal to end index pointer, i.e. the recursion has reached the base case.

The pseudocode for this algorithm will be:

```
foo(arr, start, end):
    if(start < end):
        middle = (start + end)//2
        foo(arr, start, middle)
        foo(arr, middle+1, end)
    else:
        arr[start] = 2 * arr[start]
```

Conclusion: We used two-pointers (start and end) to solve the problem using the divide and conquer technique. By not creating smaller arrays but just tracking the index positions of the sub-arrays we are saving the memory (space) needed to execute our program.

Asymptotic Analysis:

Let us see what is the time complexity of our algorithm.

For this, we will use what have learnt in previous explorations.

First, write the recurrence relation. Then solve it to find the growth function using one of the three methods that we have seen in 'Recurrence Relations' section.

You may attempt this by yourself before looking at the solution below.

The base case and recursive case of the 'foo' function will be:

> The base case runs is independent of the size of the array hence it will take constant time c1.
>
> The recursive case recursively works on two halves of the original problem and a constant time is spent in executing the if condition. Hence the equation for our recursive case will be T(n) = 2T(n/2) + c2. For a detailed explanation check the 'Exploration: Recurrence Relations' where we solved a similar problem.

We get the recurrence relation:

$$T(n) = \begin{cases} c_1 & \text{for n=1 (where start = end)} \\ 2T\left(\frac{n}{2}\right) + c_2 & \text{for n>1} \end{cases}$$

Using the Master Method we can find the time complexity.

a = 2, b= 2, f(n) = c2 (constant)

It satisfies our case 1 where: f(n) has a smaller order of growth than n^logb a

So, T(n) = Θ(n^logb a)

T(n) = Θ (n)

In this section, you saw how to implement a simple recursive function using the Divide and Conquer technique. In the next section, we will look at a couple of examples.

Exploration: Divide-and-Conquer

In this section, we will look at a few algorithms that implement the Divide and Conquer Technique.

While going through the exploration try to answer these questions:

- How to approach and solve a problem using the divide-and-conquer strategy?

Divide-and-Conquer Algorithms

To solve the problem of sorting an array of numbers we have seen insertion sort in the previous module which had a time complexity of O(n2). Let us look at the 'Merge Sort' algorithm that solves this problem using Divide and Conquer Technique.

Merge Sort

Problem: Sort an array of n numbers that are not in sorted order.

To solve this problem using the Divide and Conquer technique, we will first break the problem into two subproblems. We can divide the array of n numbers into arrays of size (n/2). It will be easy to sort an array of size (n/2) than an array of size of n. We can divide the (n/2) sized array further into two arrays of half the size. This can be done until we reach the smallest size of the array which has just one element. One element is by default sorted. Since it is easy to arrange individual elements we can now conquer and combine these individual elements. So, after first combining, we will have smaller arrays of two elements each that are in sorted order. We can arrange these to form bigger sorted arrays. Subsequently sorting the n elements of the whole array. The below animation demonstrates this process.

We are performing merger sort using Divide and Conquer by following these steps:

Divide: Break the array into two halves, we calculate the middle index position of the array and obtain two sub-arrays.

Conquer: Sort the two subarrays obtained from the divide step.

Combine: Merge the two sorted subarrays to get a single sorted array.

Let us see the implementation, we can implement this in two steps

1. Divide the array until we reach single elements, single elements are by default sorted.
2. Combine the subarrays obtained from the previous step.

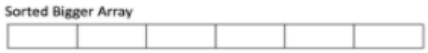
Step 1 can be achieved by this pseudocode:

```
merge_sort(Arr,start,end):
    if(start<end):
        mid = (start+end)//2 #Computes floor of middle value
        merge_sort(Arr,start,mid)
        merge_sort(Arr,mid+1,end)
```

Does this look familiar? Yes, it is similar to the 'foo' function that we looked in 'Recurrence Relations' Exploration. This code just goes on dividing the array until we reach the single element.

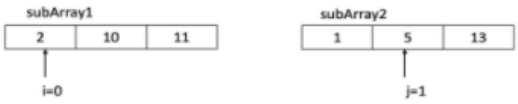
To merge the two smaller arrays after dividing them (remember the smallest sub-array that we get from merge_sort function is already sorted since it has a single element, which is default sorted, out next task is to merge them), we will write a merge function so that we can keep our code clean. To implement the merge function on two sorted subarrays we will check the first element of each subarray and copy the smallest element to our bigger array, then look for the next smallest element among the two subarrays. We can do this with the help of two pointers that iterate through two subarrays. See below interactive slides to understand how this is done (variables i, j in are used to iterate through the array elements)

Which is smaller? subArray1[i] or subArray2[j]

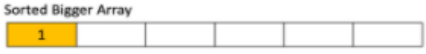


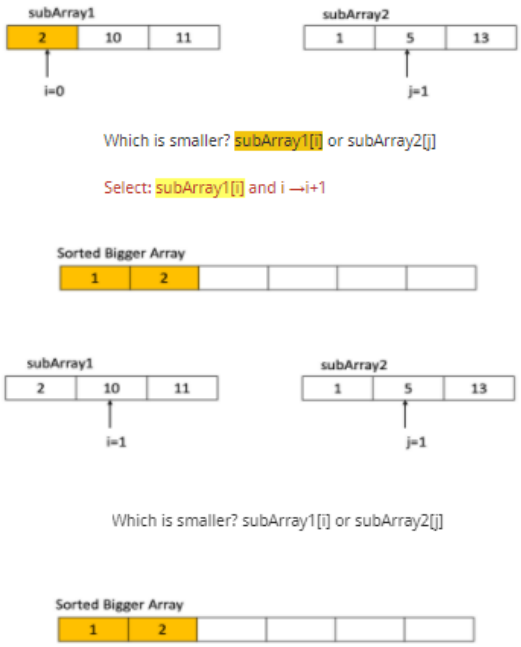
Which is smaller? subArray1[i] or subArray2[j]

Select: subArray2[j] and j →j+1



Which is smaller? subArray1[i] or subArray2[j]





Once we reach the end of any of the subarrays we can copy remaining elements from other subarrays that are not yet copied. One thing to note here is that in the divide step we were not actually creating new arrays but tracking the starting and ending index positions of our subarray. Hence before copying we need to make temporary arrays of our subarrays and then copy to the bigger array.

The pseudocode for merge procedure:

```
merge(Arr, start, mid, end):
  #temporary arrays to copy the elements of subarray
  leftArray_size = (mid-start)+1
  rightArray_size = (end-mid)

  leftArray = [0]*leftArray_size
  rightArray = [0]*rightArray_size

  for i in range(0, leftArray_size):
    leftArray[i] = Arr[start+i]

  for i in range(0, rightArray_size):
    rightArray[i] = Arr[mid+1+i]

  i=0
  j=0
  k=start
```

```
  while (i < leftArray_size and j < rightArray_size):
    if (leftArray[i] < rightArray[j]):
      # filling the original array with the smaller element
      Arr[k] = leftArray[i]
      i = i+1
    else:
      # filling the original array with the smaller element
      Arr[k] = rightArray[j]
      j = j+1
    k = k+1

  # copying remaining elements if any
  while (i<leftArray_size):
    Arr[k] = leftArray[i]
    k = k+1
    i = i+1

  while (j<rightArray_size):
    Arr[k] = rightArray[j]
    k = k+1
    j = j+1
```

Analysis of Merge sort implementation:

The time complexity of merge function: The while loop inside the merge function contributes to the maximum execution time. It runs at most n times i.e. the size of the array. Hence it will have a running time of c1n.

The time complexity of merge_sort function: We have seen this before in 'Recurrence Relations' exploration for the 'foo' function. Its recurrence relation will be T(n) = 2T(n/2) + c2

The recurrence relation for our merge sort algorithm will be:

T(n) = 2T(n/2) + c1n + c2

We can apply the Master Method and obtain the time complexity for this recurrence relation as Θ(n log n).

Exploration: Dynamic Programming Fundamentals

Introduction

In the last module, we have seen an algorithmic paradigm to solve a problem that can be broken into smaller subproblems, further optimization can be achieved by using a technique called Dynamic Programming. In this section, we will section, we will explore the Dynamic Programming technique to solve algorithms.

As you go through this section try to answer these questions:

- What is Dynamic Programming?
- When to use Dynamic Programming?
- What are the two approaches of Dynamic Programming?

Dynamic Programming

Do you remember the Fibonacci Number problem?

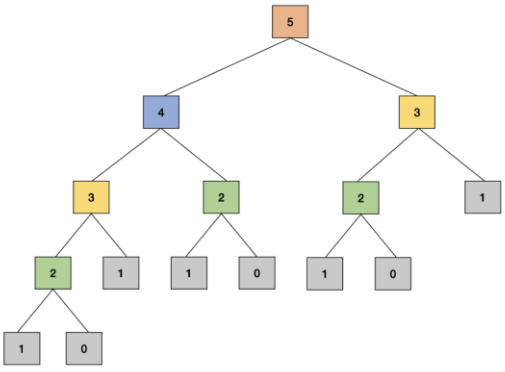
To find the nth Fibonacci number the recurrence 'formula' can be given as following. It is little different from recurrence relation, here we represent the values that our function returns for each value of n. When n = 0 or n = 1, F(n) will return 1; and for other values of n the function F(n) will return F(n-1) + F(n-2). Here F(n) represents our function itself unlike T(n) in recurrence relation in the previous exploration, where T(n) represents the time taken to solve problem of size n.

$$F(n) = \begin{cases} 1 & \text{If n = 0} \\ 1 & \text{If n = 1} \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

The pseudocode to solve this problem using Divide-and-Conquer:

```
Fib(n):
    if n = 0:
        return 1
    else if n = 1:
        return 1
    else:
        return Fib(n-1) + Fib(n-2)
```

Let's see the recursion tree representing the recursive calls for n = 5.



It is good that we are dividing the big problem of calculating for n = 5 into smaller problems to get the final problem. The time complexity of this algorithm is 2^ n/2 (Exponential – which is very slow). Did you notice that we are calculating the same smaller problem multiple times? Notice in the recursion tree that we are calculating for n = 3 twice and for n = 2 thrice. We are repeating our work for the same

subproblem, aren't we? So, when we ask ourselves if we can make this algorithm to run faster, the answer will be yes. We just have to avoid redoing the work of the first time. So, after we find the value for n=3, we will store this result and when we need it again, we will use the stored value rather than recalculating it. Similarly, for n=2. Tis is the idea behind Dynamic Programming.

Two Approaches to Dynamic Programming

Dynamic Programming is a powerful algorithm design technique that can be used to solve problems that have strictly overlapping subproblems. There are two approaches that can be used to solve an algorithm using Dynamic Programming: the top-down approach and the bottom-up approach. Let us solve the Fibonacci Number problem to learn about these approaches.

Top-Down Approach

As we have discussed, we can optimize the solution of Fibonacci Number problem (Fib(n) pseudocode) by storing the results of the overlapping subproblems. To achieve this we will create a new variable, FibMemo, of the dictionary datatype, where we can store the results of the subproblems that we calculated. By default, we can also include the base case values into this dictionary variable.

FibMemo = {0:1 , 1:1}

FibMemo[0] = 1

FibMemo[1] = 1

After we calculate the value of a sub-problem we can update our FibMemo variable. Let's say we calculated the value for n = 2, our updated FibMemo will be {0:1, 1:1, 2:2}

The code for this top-down implementation will be (Why is it called top-down will be answered in a while):

```
FibMemo = {0:1, 1:1}
def Fib_top_down(n):
    if n in FibMemo:
        return FibMemo[n]
    FibMemo[n] = Fib_top_down(n-1) + Fib_top_down(n-2)
    return FibMemo[n]

print(Fib_top_down(5))
```

The below animation demonstrates how the recursion tree grows during each recursive call for n = 5. Notice that we did not work on the entire tree as in the case of Fib(n) algorithm. As you see below we are starting with the n value and we are going down the tree to find the solution for the problem, hence the name top-down approach.

So, Fib_top_down(n) recurses only for distinct values of n for each subsequent recursive sub-call. We memoize (remember) and reuse the subproblem solutions to help solve the problem. This technique of storing temporary results in the top-down approach is called '**memoization**; (without ' r'), which is similar to creating a memo.

Time Complexity Analysis:

For every k, that belongs to [1,n] we calculate Fib_top_down(k) once. So, our recurrence relation will be:

T(n) = T(n-1) + theta(1)

This would give us a time complexity of theta(n)

Another way of analyzing this would be:

We spend time to calculate only non-memoized values of n, n-1,...1. Memoized calls take constant time theta(1) since wee have to only look up the value in the FibMemo structure.

Time Complexity = number of Subproblems * times/subproblem

= n*theta(1) [since we spend constant time during each call which is independent of n]

= $\epsilon\theta(n)$

We can slightly modify our code as below to get rid of the global variable FibMemo.

```
def fib_top_down2(n, fibmemo=None):
    if fibmemo is None:
        fibmemo = {0:1, 1:1}
    if n not in fibmemo:
        fibmemo[n] = fib_top_down2(n-1, fibmemo) + fib_top_down2(n-2, fibmemo)

    return fibmemo[n]
```

Bottom-Up Approach

In this approach, we will solve smaller sub-problems first and then combine the results until we solve our actual bigger problem. For the Fibonacci Number problem, we will start with base cases: Fib(0), Fib(1), then we will find Fib(2), Fib(3) and so on until we reach our n value.

It is similar to the iterative approach and can be achieved using a loop. The algorithm for this approach will be:

```
def fib_bottom_up(n):
    fib_table = [0]*(n+1)
    fib_table[0] = 1
    fib_table[1] = 1

    for i in range(2,n+1):
        fib_table[i] = fib_table[i-1] + fib_table[i-2]
    return fib_table[n]
```

This approach is also called 'tabulation' as we are constructing the table of results of all sub-problems until we reach the value of n for our problem.

Time complexity analysis: We are executing a for loop for range n. Hence our time complexity will be theta(n).

Comparing Top-Down and Bottom-Up Approaches

In the Top-Down approach, we start with the **bigger problem** and go down to the base case (go top-down). As we go down we store the solution of the subproblems in extra memory space to avoid re-computation. This is mostly implemented using recursion.

In the Bottom-Up approach, we start with the **base case** and solve bigger problems as we progress until we reach our actual problem (go bottom-up). This is mostly an iterative approach that builds up the solution of the sub-problems and stores the solution in extra memory space.

Exploration: Dynamic Programming - Change Making Problem

Introduction

Let us look at Change Making Problem in this section and solve it using Dynamic Programming. As you go through this section try to observe the approach we are taking to solve the problem.

Change Making Problem

Suppose you are programming software for a vending machine. Your vending machine can give change only in the form of coins. Your target is to make sure that your machine gives fewest possible coins. The coin denominations are customizable so that it can be used in any country.

Suppose your coin denoimations are 1c, 3c, and 5c. We want to make a change for 5c. We can do this in the following ways:

1. One 5c coin = 1 coin
2. One 3c coin and two 1c coin = 3 coins
3. Five 1c coin = 5 coins

In our problem, we would be provided with the denominations of the coins and the amount for which we have to make the change. We would return -1 if change cannot be obtained.

Before we proceed with the dynamic programming approaches. Do you want to pause and think about how you would solve it without dynamic programming using brute force or a naive way?

One of the most commonly thought incorrect approach is to start with the maximum denomination of coins. This might not always give the correct solution. For example, when denominations are: 1, 5, 7 and 10, and the amount we need change for is 14. If we start with a coin worth 10 then we would have a total of five coins (10,1,1,1,1). But, the minimum number of coins that can be used are two coins (7,7). This technique of picking the best possible value is called a greedy approach, we will see this in later modules.

Recursive Solution

```
Def makechangeBF(coins, amount):
  If (amount == 0): return 0
  Result = amount +  1  # no denomination higher than amount
  For I in range(len(coins)):
    If (coins[i] <= amount):
```

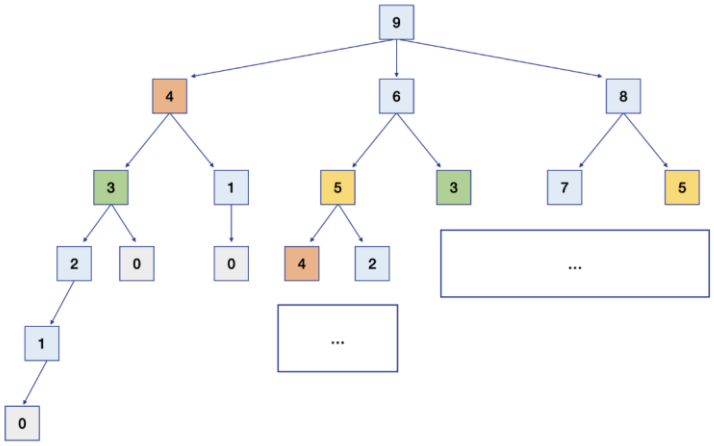
Result = min(result, makkechangeBF(coins, amount – coins[i]) + 1)

Return result

What will be the time complexity of this solution? Look at the recursion tree, in the worst case the amount A might get branched into the total number of possible denominations that are given, which is (n). Hence the time complexity will be exponential, A^n.

Top – Down Approach solution

Look at the recursion tree for the sample input values of coins (1, 3, 5) and an amount value of 9. We can notice that we are solving subproblems that are overlapping (few of them have been color-coded for easier identification. If you compete the tree you will find that there are more overlapping sub-problems).



As the amount value increases the overlapping sub-problems would increase. Hence we can use dynamic programming memoization to solve this problem.

To do this, we will need to keep a memo (which can be an array, for example, countmemo[]) of a minimum number of coins needed for an amount value that we calculate as we proceed.

We will follow the same steps that we did when performing recursion. For a given amount value we will calculate the possible number of coins needed when using each denomination.

```
import sys


def makechange_topdown( coins, amount):
    if amount == 0:
        return 0

    return makechange_topdown_helper(coins, amount, [0] * (amount + 1))
```

```
def makechange_topdown_helper( coins, amount, countmemo):
    if (amount < 0):
        return -1

    if (amount == 0):
        return 0

    if (countmemo[amount] != 0):
        return countmemo[amount]

    inf = sys.maxsize
    minimum_coins = sys.maxsize  # set to some maximum value
    for coin in coins:
        temp_coincount = makechange_topdown_helper(coins, amount - coin,
countmemo)

        if (temp_coincount >= 0 and temp_coincount < minimum_coins):
            minimum_coins = 1 + temp_coincount

    countmemo[amount] = -1 if (minimum_coins == inf) else minimum_coins  # if we
found a new minimum use it

    return countmemo[amount]
```

Code Explanation

We are using a helper function (makechange_topdown_helper) for ease of coding. Our memoization table is an array of size of the amount+1 since we can have at most 'amount' number of values to calculate. This you can observe in the tree, the depth of the tree will be the same as the value of amount if coin worth 1¢ is used. This is initialized to 0.

```
countmemo = [0] * (amount + 1)
```

In makechange_topdown_helper function, we iterate through all possible values of coins that we can use and recursively call the helper function with the remaining amount after using a coin. This helper function returns the best possible count of coins found so far.

```
for coin in coins:
    temp_coincount = makechange_topdown_helper(coins, amount - coin, countmemo)
```

For each iteration, we set minimum_coins to system.maximum number (some large number)

```
minimum_coins = sys.maxsize
```

When we have a valid coin that we can include to make the change, temp_coincount will have a valid value. If temp_coincount value is better than previously calculated minimum_coins (0 ≤ temp_coincount < minimum_coins) then we replace the minimum_coins with the new count.

```
if (temp_coincount >= 0 and temp_coincount < minimum_coins):
    minimum_coins = 1 + temp_coincount
```

It is better if you can run the code with the PyCharm debugger and see how the variables are changing to get a better understanding of the code.

Time Complexity

If the amount for which we are calculating change is A, and the total number of denominations that we have is n, then our tree will have a maximum depth of A in the worst case (i.e. when the minimum denomination is 1, then we have nodes representing, A, A-1, A-2.... 1). And for each node, we will check n possible branches. Since we have the results cached, we will do a maximum of n iterations for each denomination of the coin. So, our time complexity will be O(An)

We need an additional space of size A (our countmemo array), hence our code will require O(n) space complexity.

Bottom-Up Approach Solution

For the bottom-up approach, we will start with the base case and create an array of the number of coins needed for amount values starting from 1 to A (amount value).

```
def makechange_bottomup(coins, amount):

    min_count_table = [amount+1]*(amount+1) # setting array elements to some
large value that is not possible answer

    min_count_table[0] = 0 # setting the base case

    for i in range(1, amount+1):  # iterate through all possible amount values
from base case
        for j in range(0, len(coins)): #find the number of coins needed for each
coin denomination
            coin_val = coins[j]
            if(coin_val <= amount and (i-coin_val)>=0): # if denomination value
is less than amount then we can use the coin
                # replace min_count_table[i] with minumum value of coins possible
                min_count_table[i] = min(min_count_table[i] , min_count_table[i-
coin_val]+1)

    # we have a valid count of coins if min_count_table[amount] is valid
    if min_count_table[amount] > amount: result = -1
    else: result = min_count_table[amount]
```

```
    return  result
```

Time Complexity

We are computing A (Amount) number of subproblems, and for each value of A we are iterating through n (number of coin denominations) number of times. Hence the complexity of our code is O(A*n).

We need an additional space of size A (our countmemo array), hence our code will need O(n) space complexity.

In the next section, we will consolidate the steps that we can take when solving a problem using the dynamic programming technique.

Exploration: Dynamic Programming - Longest Common Subsequence Problem

Introduction

In this section, we will look at the steps that can be taken to solve a problem using dynamic programming and summarize how to identify if a problem is solvable using the dynamic programming technique. We will in turn also solve one of the classic dynamic programming problems - Longest Common Subsequence problem. While you read try to answer the following questions:

- What are the steps to solve a problem using Dynamic Programing?
- When to use Dynamic Programming?

Where to use Dynamic Programming and Steps

For a problem to be solvable using dynamic programming it must have two characteristics: it should have **overlapping subproblems** and it should have **optimal structure**.

- Overlapping Subproblems: We have visited the point regrading overlapping subproblems. It is to avoid re-computation of the same subproblems. For example, the Fibonacci Number problem, and the change-making problem have many overlapping subproblems. They can be solved using dynamic programming
- Optimal Substructure: When the problem requires us to find the optimal (minimum or maximum) value and the optimal solution to the problem can be obtained by solving for the optimal solution for the problem's subproblems; then we can say that the problem has optimal substructure. In other words when the problem asks – what is the optimum (min or max) solution. Can you find the optimum value in a subproblem that would ultimately lead to the optimum solution for the whole problem? For example, in the change-making problem in the last section, we could obtain the minimum number of coins needed for an amount by solving for the minimum number of coins needed for all values lesser than the amount, starting from the base case.

When you read a problem and if you find that it needs optimization (finding the minimum or maximum of something), identify whether the problem has optimal substructure and overlapping subproblems. If your answer is yes, then you can likely use dynamic programming to solve it.

Steps to Approach Dynamic Programming

1) Identify the parameters that impact the problem
2) Identify the subproblem
3) Define the recursive formula (recurrence relation) including the base case
4) Implement a naive recursive solution (This is optional, though it will help you get a good understanding of dynamic programming)
5) Turn the recursive formulation into dynamic programming algorithm
    a. Optimize the recursive solution to use cache (memoization)
    b. Implement using the top-down or bottom-up approach

Longest Common Subsequence

Given two strings str1 and str2, return the length of their longest common subsequence of characters between the two strings. A subsequence is a sequence that appears in the same relative order, but are not necessarily contiguous. If there is no common subsequence, return 0.

Assume that an empty string is not a subsequence of any string.

Example:

Input: str1 = "abcdef", str2 = "apceg"

output: 3

Explanation: The longest common subsequence is "ace" and its length is 3.

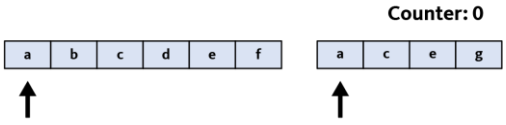
Before thinking about the dynamic programming approach, what is your naive approach to solve this problem? Take a pen and paper and think about how would you solve this problem if you were to write an algorithm for this.

Note about this problem: This problem has many real-world applications, for instance when Git needs to merge two files it needs this algorithm, in biometrics -it is used to detect similarities between two genetic codes, and many more.

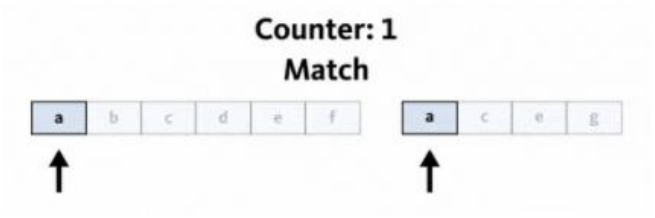
Recursive Solution

When we are given two strings (str1 = "abcdef", str2 = "aceg") our minds can look at it and directly tell the length of the longest common subsequence. Let's decipher the steps that our mind is taking.

We start with a counter in our minds that is initially set to 0.



We then look at the first character in the first string ("a") and the first character in the second string ("a"), if it is a match we increment the counter by 1.



And move on to the second character of each string ("b" and "c"). They don't match. So we would keep our pointer on either of the strings static (let's say first string) and increment to the next character in the second string to see if we find a match. Next character is "e", which does not match, so we move the pointer forward to compare the next character. This way we do this until we find a match or reach the end of the second string. In this case, there is no match.

Then we come back to the first string and increment the pointer to the next character ("c") and compare the second string from the character that was our first case of no-match in our previous step, the character "c" (We don't want to start from "a" since it has already been matched).

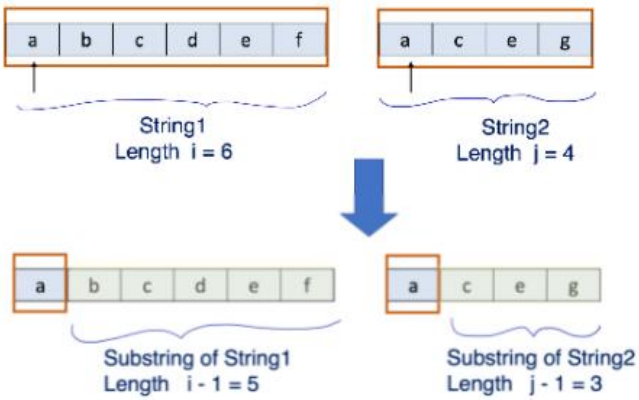
Similarly, we move to our next characters in both the strings.

This gives us an idea about how to solve this problem recursively.

Let us look at the steps now.

Step1: Identify the parameters that impact the problem: Length of string1 and string2 impact our problem.

Step2: Identify the subproblem: In our solution analysis, observe that after we have compared the first characters of the strings, the remaining subproblem is again a longest common subsequence problem, on two shorter strings. The below figure explains this. First, we look at the problem for strings of lengths i, j (i=length of string1, j=length of string2). Since it matched, we are looking for a solution to the problem of size i-1 and j-1.



We can define our subproblem to be:

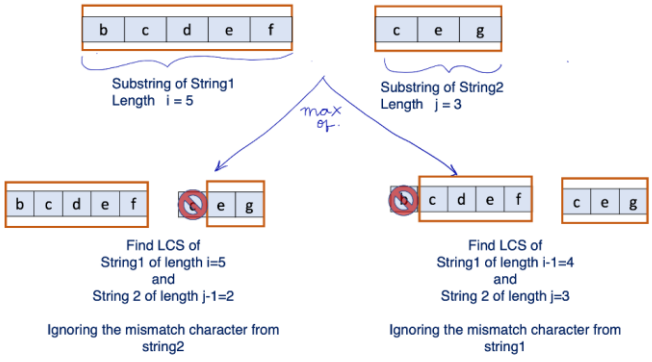
LCS[i,j]: Longest common subsequence of strings of lengths i and j.

Step3: Define the recursive formula: Based on the understanding from step1 and step2, we can see that when we are matching the first characters of two strings, they would either match or not match, which gives us two cases.

As in the above figure when we are solving for LCS[i,j] when the first characters match we will count it as one match and add 1 to the result of substrings of lengths i-1 and j-1. This can be written mathematically as:

LCS[i,j] = 1 + LCS[i-1, j-1] When the first characters of str1 of length i and str2 of length j match.

When they don't match, we have two options. The first option is to find the LCS of str1 of length i and str2 of length j-1 (here we are excluding the mismatching character from string2). The second option would be to find the LCS of str1 of length i-1 and str2 of length j (excluding the mismatching character from string1). Then, take the maximum of the two. Below figure demonstrates this.



We can write this in the form of an equation as:

LCS[i,j]= max{ LCS[i, j-1] , LCS[i-1 , j] } when the first characters of str1 of length i and str2 of length j do not match.

Finally, the base case, when the length of str1 or str2 is 0 then the LCS value will be 0.

LCS[i,j] = 0 when i = 0 or j = 0

Combining all the cases our recursive formula will be:

$$LCS[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + LCS[i-1, j-1] & \text{if the first characters of string1(length i) and string2 (length j) match} \\ \max\{ LCS[i, j-1] , LCS[i-1, j]\} & \text{if the first characters of string1(length i) and string2 (length j) DO NOT match} \end{cases}$$

Step4: Implement a naive recursive solution

```
def lcs_BF_helper(s1, s2, m,n):
    if m < 0 or n < 0:
        return 0;
    elif s1[m] == s2[n]:
        return 1 + lcs_BF_helper(s1, s2 , m-1, n-1)
    else:
        return max(lcs_BF_helper(s1, s2, m-1 , n), lcs_BF_helper(s1, s2, m, n-1))
def lcs_BF(str1, str2):
    return lcs_BF_helper(str1,str2, len(str1)-1, len(str2)-1)
print(lcs_BF("abcde", "ac"))
```

Time Complexity: This is an exponential problem. For every character we have two possible choices, it can either be in the subsequence or it cannot. So worst-case time complexity will be O(2n), where n is the length of the longer string.

Step5: Turn recursive formulation into the dynamic programming algorithm. Let us implement the dynamic programming solution.

Bottom-Up Approach

Since this problem can be solved by solving the subproblems and the problem asks for a maximum possible length of the sequence (an optimal substructure), this problem can be solved using dynamic programming.

Based on the explanation try to write the pseudocode for this approach by yourself.

We would need a two-dimensional array to store the results, we need extra space to store the results of the base case. ie. LCS[0,1], LCS[1,0], LCS[0,2] and so on. i.e when we are comparing strings with an empty string. If our strings in our problem are of lengths m and n, we need a 2-D array of size [m+1][n+1]

	"" 0	B 1	A 2	C 3
"" 0	0	0	0	0
A 1	0			
B 2	0			
C 3	0			
D 4	0			
T 5	0			

After we have our 2-D array initialized as:

```
cache = [[0 for x in range(n+1)] for x in range(m+1)]
```

We can loop through this "cache" and fill in the values using the recurrence equation that we wrote.

```
for i in range(m+1):
    for j in range(n+1):
        if i==0 or j==0:
            cache[i][j] = 0
        elif str1[i-1] == str2[j-1]:
            cache[i][j] = cache[i-1][j-1] + 1
        else:
       Top-Down Solution     cache[i][j]= max(cache[i-1][j] , cache[i][j-1])
```

We can write a recursive solution to this approach. Similar to the bottom-up approach we can create a two-dimensional table and return 0 when we approach the base case, i.e. when either of the lengths of the strings is 0.

Dynamic Programming Approaches Notes

"The first approach is top-down with memoization. In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table). The procedure now first check to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the

value in the usual manner. We say that the recursive procedure has been memoized; it “remembers” what results it has computed previously.

The second approach is the bottom-up method. This approach typically depends on some natural notion of the “size” of a subproblem, such that solving any particular subproblem depends only on solving “smaller” subproblems. We sort the subproblems by size and solve them in size order, smallest first. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions. We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

These two approaches yield algorithms with the same asymptotic running time, except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems. The bottom-up approach often has much better constant factors, since it has less overhead for procedure calls." [1]

Exploration: Dynamic Programming: Rod Cutting Problem Exercise

Introduction

In this section, you will get some practice on the topic of solving problems using the Dynamic Programming technique. Attempt the problems by yourself before you look into the solution.

When we use term Dynamic Programming, the word “Programming” refers to a tabular method, not to writing computer code.

Problem 1: Product Sum

Given a list of n integers v_1, v_2,.... v_n, the product sum is the largest number that can be formed by either multiplying or adding the adjacent numbers in the list. You can only multiply the adjacent numbers in the list.

For example,

For the list: (1, 2, 3, 1), the product sum is: 8 = 1 + (2 × 3) + 1

For the list: (2, 2, 1, 3, 2, 1, 2, 2, 1, 2), the product sum is 19 = (2 × 2) + 1 + (3 × 2) + 1 + (2 × 2) + 1 + 2.

$$(n) = \begin{cases} 0 & \text{if } i = 0 \\ v_i & \text{if } i = 1 \\ \max\{f(i-1) + vi \ , \ f(i-2) + vi\text{-}1*vi\} & \end{cases}$$

Problem 2: Rod Cutting Problem

Given a rod of n meters and a list of prices (price[]) for different lengths of the rod, a rod of length i will have a price of price[i-1]. We want to sell the rod and make the maximum possible revenue. We can cut

the rod into lengths of different sizes before selling. The price of each piece will be determined by the list of prices provided. What is the optimal amount of revenue we can get out of the rod given length, n and prices?

Example: Rod length: 4, price = [1,5,8,9,10]

The optimal revenue: 10

Explanation:

From the price array, the price of lengths 1 = 1, price of length 2 = 5, price of length 3 = 8, price of length 4 = 9, price of length 5 = 10

Length of pieces after the cut	Revenue
4 (no cut)	9
1, 3	(1+8)=6
2,2	(5+5)=10
1,1,2	(1+1+5)=7
1,1,1,1	(1+1+1+1)=4

Week 4

Exploration: Dynamic Programming – Unbound Knapsack Problem

In this section and the next one we will discuss one of the common problems that can be solved using dynamic programming, it is called knapsack problem. There are multiple versions of this problem we will look at two of them in this module.

At the end of this section, you will be able to solve an unbound knapsack problem.

As you go through this section, try to answer the following questions;

- What is a knapsack problem?
- How to solve an unbounded knapsack problem?

Knapsack Problem

Suppose you woke up on a mysterious island and there are numerous valuable items on the island. You have a knapsack (a bag) with you which you can use to take back these items with you. But, the problem is there is a limit to the weight that your knapsack can carry. So, our questions would be what is the best way to cram items into our knapsack to maximize the overall value of items that you can carry back with you.

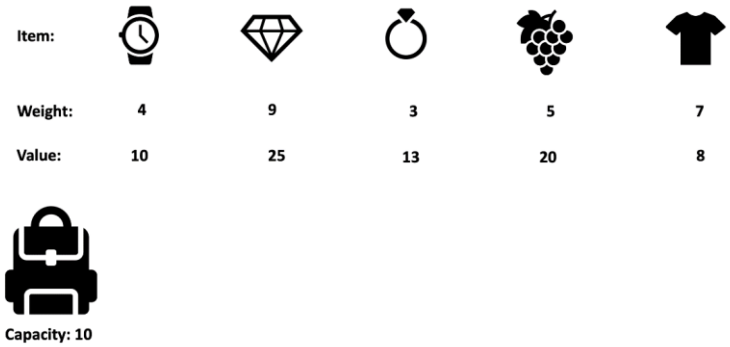
W have n items with their weights w_i and values v_i. We are given the total capacity of the knapsack, W.

For example,

Consider items with weights w = [4, 9, 3, 5, 7]

And their corresponding values v = [10, 25, 13, 20, 8]

The knapsack that we have cannot carry weight more than W = 10.



There are two versions of the knapsack problems:

1. Unbounded knapsack
    a. In the case of the unbounded knapsack, we have unlimited copies of all the items.
    b. What is the best way to maximize the value of our knapsack in this case?
2. 0-1 knapsack
    a. In the case of the 0-1 knapsack, we have only one copy of each item. We can either chose a given item or we don't choose an item to maximize the value of our knapsack. We will look more into this in the next section.

Based on the approach that you used to solve the unbounded knapsack activity, do you think we can solve this problem using dynamic programming?

Yes, the problem has overlapping subproblems. We pick one item and see the best solution that can be obtained with the remaining capacity of the knapsack. We repeat this procedure until we find the optimal solution. Our approach has both overlapping subproblems and optimal substructure.

Unbounded knapsack

We have infinite copies of all the items and we have to maximize the value of our knapsack.

Let us follow the steps that we have seen before to solve a dynamic programming problem.

Step 1: The parameters that impact the problem: The capacity of the knapsack is the only parameter that would change with every iteration.

Step 2: Identify the subproblem: We will find the solution for a smaller knapsack first, then for a bigger one and then an even bigger one until we reach W.

The subproblem will be, the optimal solution for a knapsack of weight x. The value of x cannot be greater than W. We can write as f(x).

So, our subproblem will be

f(x) = optimal value for a knapsack of capacity x.

Step 3: Define the recurrence formula including the base case

To solve our knapsack of capacity x: if our bag has a capacity of x, we can try putting in the bag each item of weight w_i, and find an optimal solution for a subproblem for a knapsack of weight x-w_i (i.e. the remaining capacity after placing i-th item).

If W = 5, W = [1,2,3,4], V = [10, 20, 5, 15]

f(x) would look like a one-dimensional array of size W + 1. When x = 0, the optimal value will be 0 (base case).

We shall try all weights from w[] to find the best value for x = 1 subproblem.

So, for each item i, we will place it in our knapsack and find the value of the choice

= v_i + [best solution for knapsack of capacity x – w_i]

This brings us our recurrence formula:

$$f(x) = \begin{cases} 0 & \text{if there is no i such that } w_i \leq x \text{ or if } x = 0 \text{ (base case)} \\ \max \{f[x-w_i] + v_i\} & \end{cases}$$

0<i<= n; w_i < x

V_i = value of i-th item w_i = weight of I-th item

Step 4: Turn the recursive formula into dynamic programming algorithm

Since our recursive formula is a function of a single variable (x), we will store the answers to our subproblems in a 1-dimensional array, this will be our cache.

This can be better solved with a bottom-up approach. If we are able to find the best solution for a knapsack of smaller size then we find the optimal solution for the bigger size knapsack.

```
def unbound_knapsack(W, n, weights, values):
    dp = [0]*(W+1)

    for x in range(1, W+1):
```

```
        for i in range(n):
            wi = weights[i]
            if wi <= x:
                dp[x] = max(dp[x] , dp[x-wi] + values[i])

    return dp[W]


print(unbound_knapsack(10,5,[4,9,3,5,7], [10,25,13,20,8]))
```

As you might have noticed in the code,, we are filling our cache of all subproblems in dp[] array with maximum possible subsolution in the statement:

```
dp[x] = max(dp[x] , dp[x-wi] + values[i])
```

Time complexity analysis: We are iterating through two for loops, one for all possible weight values until the capacity W of the knapsack, and the second loop to iterate over all possible n items. This gives us a time complexity of O(n*W).

Applications of the Knapsack algorithm

The Knapsack algorithm can be used in a lot of real-world decision making situations like internet download managers, the data to be downloaded is broken into chunks and server uses this algorithm to pack these chunks to use the maximum available limit.

Exploration: Dynamic Programming - 0-1 Knapsack Problem

Introduction

In this section, we will solve the 0-1 knapsack problem.

The 0-1 knapsack Problem

We have seen that in the 0-1 knapsack problem we have only one copy of each item. When finding the optimal solution we have only two options when we consider one item: either we chose it or we don't chose it, hence the name 0-1.

For example,

Consider items with weights w= [4,9,3,5,7]

and their corresponding values v = [10, 25, 13, 20, 8]

The knapsack that we have cannot carry weight more than W = 10

Item:					
Weight:	4	9	3	5	7
Value:	10	25	13	20	8

Capacity: 10

What is the best way to maximize the value of our knapsack if we have only one copy of each item.

Let us follow the steps that we have seen before to solve a dynamic programming problem to approach the solution of the 0-1 knapsack problem.

Step 1: The parameters that impact the problem: As with the unbound knapsack problem, the capacity of the knapsack is a parameter that impacts the solution of our problem. Additionally, the available items also impact our solution. For instance, if we included an item A, we cannot consider it again since we have only one copy of it. Hence, these will be our two parameters that would impact our problem.

Step 2: Identify the subproblem: Along with the capacity, our subproblem should also contain information about the items that we have used. Here, we are speaking about the two parameters defined above. Hence, our subproblem will be of form f(x,i).

f(x,i) = optimal solution for a knapsack of capacity x using first i items ($1 \leq I \leq n$)

So, for each ith item, we will consider

What will be the optimal solution if we **do** include it

What will be the optimal solution if we **don't** include it.

We will evaluate these two cases and consider the one that results in the best value.

Step 3: Define the recurrence relation including the base case.

Here the recurrence formula will be:

$$f(x,i) = \begin{cases} 0 & \text{if } i=0 \text{ or } x=0 \\ \max \{v_i + f[x-w_i, i-1] \;,\; f[x, i-1]\} & \text{otherwise} \end{cases}$$

$1 <= i <= n$

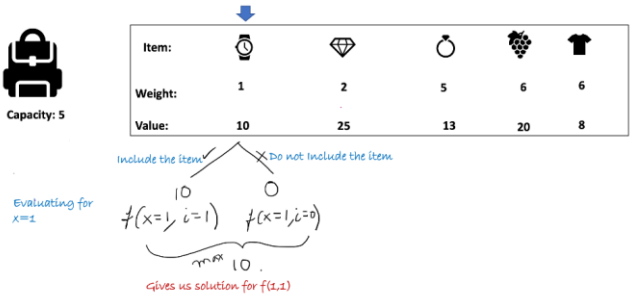
Step 3: Turn the recursive formulation into the dynamic programming algorithm.

Before we approach the solution let us look at the cache structure to store the subproblem results.

We need to track two parameters as listed in our recurrence relation equation, the value of x and i, so we need a 2-dimensional array. Let us visually see why we need a 2-dimensinoal array.

If our w = [4, 2, 5, 6, 7]; v = [10, 25, 13, 20, 8] and W = 7

When we are evaluating a subproblem, we are iterating through each item and we are calculating the value obtained by including it and by not including it; and we are taking the maximum of the two. The below figure shows us the case when we are evaluating for the subproblem of capacity 1, and we are considering the first item (i=1). This gives us the value of f(1,1) (the first 1 represents the capacity and the second 1 represents the ith item, the first item in this case).



Similarly, by evaluating the second item for x = 1, we will obtain the value for f(1,2). By evaluating the third item, we will obtain f(1,3) and so on.

Once we are done calculating all the values for our x=1 subproblem we shall calculate for the next subproblem of capacity x=2.

Evaluating the first item for x = 2 capacity will give us f(2,1). This leads us to a two-dimensional table as shown below. We obtain the solution for the problem when we reach the last cell in the table, where we reach the value for our W and n; which is f(7,5) in the current example.

Weight:	1	2	5	6	7
Value:	10	25	13	20	8
	i=1	i=2	i=3	i=4	i=5
x=1	f(1,1)	f(1,2)	f(1,3)	f(1,4)	f(1,5)
x=2	f(2,1)	f(2,2)	f(2,3)	f(2,4)	f(2,5)
x=3	f(3,1)	f(3,2)	f(3,3)	f(3,4)	f(3,5)
x=4	f(4,1)	f(4,2)	f(4,3)	f(4,4)	f(4,5)
x=5	f(5,1)	f(5,2)	f(5,3)	f(5,4)	f(5,5)
x=6	f(6,1)	f(6,2)	f(6,3)	f(6,4)	f(6,5)
x=7	f(7,1)	f(7,2)	f(7,3)	f(7,4)	f(7,5)

Our base cases value would be 0 if either x=0 or i=0.

Code Practice

With this understanding of the cache, try to write the pseudocode for solving the 0-1 knapsack problem using the bottom-up approach of dynamic programming.

Pseudo Explanation:

First, we are creating a two-dimensional array. Then, we are looping through each item for each value of smaller capacity bags (0 to W). Then we are calculating the optimal solution for f(x,i) using the recurrence formula that we discussed before.

By default, we are assigning a current value of cache (in below line) to be same as the best solution obtained with all the previous items, which refers to the case of not including the ith item.

```
dp[x,i]=dp[x,i-1]
```

Time Complexity analysis: We are iterating through two for loops, one for all possible weight values until the capacity W of the knapsack, and the second loop to iterate over all possible n items. This gives us a time complexity of O(n*W).

Exploration: Dynamic Programming - Find Optimal Solutions

Introduction

Until now we have looked at different problems that can be solved using the Dynamic Programming approach. For instance, we looked at the longest common subsequence problem - we solved for the length of the longest common subsequence between two strings, but we did not find the subsequence itself. In this section, we will look at the way of finding an optimal solution for some of the dynamic programming problems.

Optimal Solution

We have seen that we can solve optimization problems using dynamic programming. These programs have many possible solutions as opposed to a single solution.

For instance, let's say we want to find the optimal solution for the longest possible subsequence between two strings "AXYBC" and "ABCXY". What is the length of the longest possible subsequence? It is 3. And what is the longest possible subsequence? It can be either "ABC" or "AXY". Both are correct answers. This is what we mean when we say that optimization problems have 'an' optimal solution vs 'the' optimal solution. Our target in solving the problems in this section would be to find an optimal solution.

Longest Common Subsequence problem

In the previous module, we have seen the code to obtain the length of the longest common subsequence of two strings.

Below is equation and code for the bottom-up approach.

$$LCS[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ 1 + LCS[i-1, j-1] & \text{if the first characters of string1(length i) and string2 (length j) match} \\ \max\{ LCS[i, j-1] , LCS[i-1, j]\} & \text{if the first characters of string1(length i) and string2 (length j) DO NOT match} \end{cases}$$

```
def lcs_bottomup(str1, str2):
    m = len(str1)
    n = len(str2)

    # create a 2-D dynamic programming table of size m+1 X n+1
    cache = [[0 for x in range(n+1)] for x in range(m+1)]

    # building the matrix
    for i in range(m+1):
        for j in range(n+1):
            if i==0 or j==0:
                cache[i][j] = 0
            elif str1[i-1] == str2[j-1]:
                cache[i][j] = cache[i-1][j-1] + 1
            else:
                cache[i][j]= max(cache[i-1][j] , cache[i][j-1])

    return cache[m][n]
```

In this solution, we returned the length of the longest common subsequence between two strings str1 and str2. If we want to obtain the longest common subsequence itself, and not just the length, how would we solve it?

For this, we can use the cache table to build the solution. We will start at cache[m][n] and trace backwards. If the str1[m] and str2[n] are same then we know that it is part of the longest common subsequence. If str1[m] and str[n] are not the same we will take our step backwards towards the largest adjacent entry between cache[m][n-1] and cache[m-1][n].

So, we can use the recurrence equation and the cached two dimensional array to obtain an optimal solution.

```
def optimalsolution_LCS(str1, str2, cache):
    # Cache is already filled out 2-D array solution
    m = len(str1)
    n = len(str2)
    lcs = []

    while m>0 and n>0:
        if(str1[m-1] == str2[n-1]):
            #append str1[m] to lcs
            lcs.append(str1[m-1])
            m -=1
            n -=1
        elif(cache[m][n] == cache[m][n-1]): n= n-1
        else: m = m-1

    #lcs has the solution in reverse order,
    # since we traversed from the last characters of two strings
    # We will get our solution by reversing the value of lcs
    return ''.join(reversed(lcs))



def lcs(str1, str2):
    m = len(str1)
    n = len(str2)

    # create a 2-D dynamic programming table of size m+1 X n+1
    cache = [[0 for x in range(n+1)] for x in range(m+1)]

    # building the matrix
    for i in range(m+1):
        for j in range(n+1):
            if i==0 or j==0:
```

```
                cache[i][j] = 0
            elif str1[i-1] == str2[j-1]:
                cache[i][j] = cache[i-1][j-1] + 1
            else:
                cache[i][j]= max(cache[i-1][j] , cache[i][j-1])

    return  optimalsolution_LCS(str1, str2, cache)


print(lcs("ABCDGH", "AEDFHR"))
```

Code explanation: In the first if condition, we are comparing the corresponding characters of the string from the 2-D array.

```
if(str1[m-1] == str2[n-1])
```

If the characters don't match we just go back to the "current" cell in the 2-D array and accepting that answer.

```
elif(cache[m][n] == cache[m][n-1]):
```

As discussed in the video we need to reverse the final string that we obtained. The below line of code is one of the ways of doing it.

```
''.join(reversed(lcs))
```

Time complexity: In the worst case if we compare and retrace all the characters of both of the strings "optimalsoution_LCS" subroutine will take an extra O(m+n) time complexity.

Change Making Problem

Let us revisit the change-making problem that we saw in the previous module.

Problem: We were given some denominations of coins and an amount value, A. We have to find the minimum number of coins necessary to make the change for the given amount. This time, we don't want to return the minimum number of coins but we want to print the coin denomination that can be used to make the change. We still need to satisfy the condition of returning the minimum number of coins.

As a refresher, here is the recurrence equation and the code that we discussed before. If you find it difficult to recollect, it is advisable to study the exploration Dynamic Programming - Change Making Problem and come back to this section.

OPT[amount] = min{OPT[amount-di]+1 } ; di ranges over all possible denominations

```
def makechange_bottomup(coins, amount):

    min_count_table = [amount+1]*(amount+1) # setting array elements to some large va
lue that is not possible answer

    min_count_table[0] = 0 # setting the base case

    for i in range(1, amount+1):  # iterate through all possible amount values from b
ase case
        for j in range(0, len(coins)): #find the number of coins needed for each coin
denomination
            coin_val = coins[j]
            if(coin_val <= amount): # if denomination value is less than amount then
we can use the coin
                # replace min_count_table[i] with minumum value of coins possible
                min_count_table[i] = min(min_count_table[i] , min_count_table[i-coin_
val]+1)

    # we have a valid count of coins if min_count_table[amount] is valid
    if min_count_table[amount] > amount: result = -1
    else: result = min_count_table[amount]
    return  result
```

Explain of the approach to solve the problem: We saved the minimum possible number of coins for each amount value starting from 1 to amount in the 'min_count_table' as shown in the below animation.

Let us say the amount = 6; coin denominations = [1,3,5].

amounts	1	2	3	4	5	6
min_count_table	1	2	1	2	1	2
Coins	1	1 1	3	1 3	5	3 3

Now we need to keep track of the coins that were used to form the amount value. For example, for amount 2 the coin of denomination 1 is used. We can store this in the coin_used array for value 2. coin_used[2] = 1 (1 represent denomination 1). To find all the coins that were used to value 2: we can say denomination 1, and the solution for the remaining amount, i.e. 2-1 = 1. So we will look at the solution for the coins used for value 1, which is again a denomination 1.

Similarly, to find the solution for amount = 4, we can use coin of denomination 3 and the solution for the remaining values of 1 (4-3), which can be found by looking at the solution for amount=1.

amounts	1	2	3	4	5	6
min_count_table	1	2	1	2	1	2
Coins	1	1 1	3	1 3	5	3 3

Coin_used	1	1	3	3	5	3

To find the coins that form our solution we will look at the coin_used[6], which gives us 3 as one of the coins that can be used. To find the remaining coins we check for the remaining amount that is left after

using a coin of denomination 3, it gives us 6-3 = 3. So to find the coins used for amount 3, we will again look at the coin_used array for value 3 (coin_used[3]), this gives us a coin of denomination 3. After considering the second coin, the remaining amount will be 3-3=0, indicating no more coins to be used. Hence our solution will be coins of denomination 3,3.

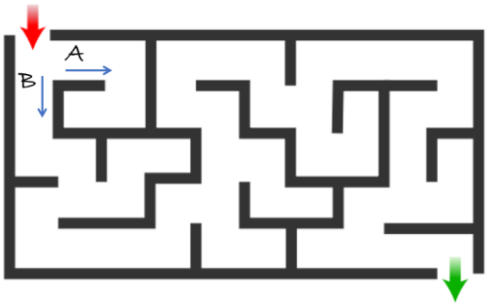
Exploration: Backtracking

Introduction

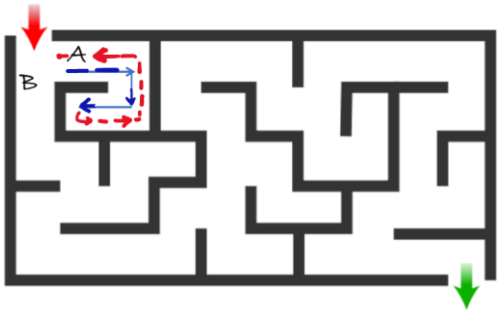
In this section, we will look into an interesting algorithm solving technique called backtracking. We will cover an overview of the idea behind this technique and solve a simple problem using the backtracking technique.

Backtracking Technique

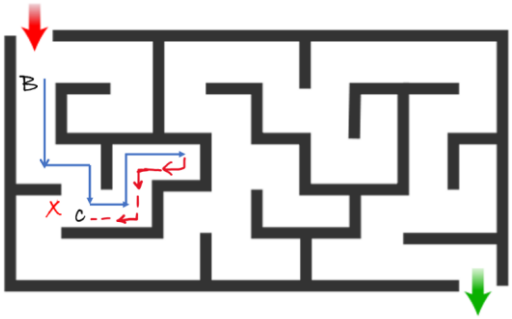
Suppose we want to solve a maze puzzle. We will first start at the entry point (marked with red arrow). From there we have two possible choices that we could take path A or path B.



Let us say we decide to take choice A, and we walk along the blue line. Once we reach the dead end, we realize that we have not taken a correct choice and we return back to the point where we could take path B.



After walking further on path B, at the junction X we again have two choices and suppose we take path C. After walking a little we realize that it does not lead us to end. So we again backtrack and we follow this process until we reach our goal.



Did you realize how we take a possible path (a choice), we stop exploring that path once we realize that it does not lead us to our goal and we backtrack our steps till the point where we can explore a new path? This is the exact technique of backtracking algorithms.

In this technique, you evaluate every possibility. We stop the evaluation if we meet any constraints and take your steps back until you are exhausted with all possible paths.

So, in the backtracking technique might appear we go on solving until we meet an end (or a constraint) and we sense that it would not lead us to the solution.

How do we identify if using the backtracking technique is appropriate for a given problem? Basically, if the problem asks for any of the following then it can be solved using backtracking:

- We want to 'generate all' possible answers
- We want a valid solution rather than an optimum solution based on some value (When asking for an optimum solution we want to think about Dynamic Programming)

Permutations Problem

Problem description: You are given a string of characters. You have to print all possible combinations of those characters (all permutations). You should not repeat any character. Assume that you will be given distinct characters in the string.

Example:

Input: "ABC"

Output:

ABC, ACB, BAC, BCA, CAB, CBA

Note: if there are n characters in the string, the possible permutations would be n!. In the above example, there were 3 characters in the string, hence we have 3! = 6 possible permutations in the result.

Solution using backtracking

Let us solve the example of input: ABC.

There are many ways to approach this problem, let us look at the backtracking technique.

We start with the available characters. Our first combination can start with character A or B or C. If our first character starts with A(A _ _), we have two possible choices to make from B and C. If we choose B. We have only once choice to make, that is C. Now, we are left with no more choices and our resulting string is of length 3 and is one of the possible combinations. Since we have no more choices to make now, we backtrack till A _ _, and this time we choose C. And repeat our steps, until we find all possible permutations.

```
def permutations(result, str):
    #base Case, print the result when we obtain the result using all characters
    if(len(result) == len(str)):
        print(''.join(result))

    for i in range(len(str)):
        current_choice = str[i]
        # If the choice was not already made we chose it to include in our result
        if(current_choice not in result):
            result.append(current_choice)
            #recursively calling permutations function until we obtain our result
            permutations(result, str)
            #Once we have exhausted all possible paths we backtrack
            result.pop()

def permuations_backtracking(str):
    permutations([],str)

permuations_backtracking("ABC")
```

Code Explanation;

We are looping through each of the available characters using the for loop. We add our choice to our result if it was not chosen previously. Then we recursively call our “permutations’ function to complete our results with available choices.

```
if(current_choice not in result):
    result.append(current_choice)
    permutations(result, str)
```

Once we have exhausted all the possible choices, we backtrack by removing the last characters before exploring other available choices for the recently emptied space.

Time Complexity: If you look at the tree structure in the video, in the first level we have n possible nodes, and in the next level each node expands to n-1 leaves, and further down, next level has n-2 leaves per node. Hence, we are performing a total of

n*(n-1)*(n-2)*….*(1) – n!

This gives us a complexity of O(n!)

Exploration: Backtracking – N Queens Problem

Introduction

In the previous module, you have seen the basic idea behind the backtracking algorithm. In this section, we concretize our understanding of approach to solve problems using the backtracking algorithm. We will solve the N-Queens Problem in this section.

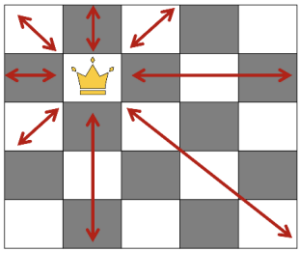
N-Queens Problem

A refresher from the previous module: In the backtracking technique, we find all possible solutions to a problem. We do this by incrementally building up the solution and discarding (or backtracking) from those solution paths that don't lead us to the goal.

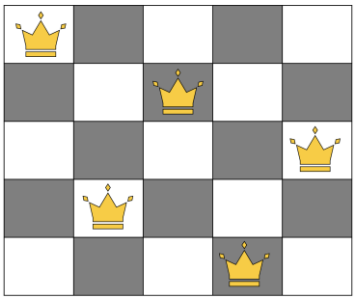
There are three elements to the backtracking technique

1. Possible choices
2. Constraints if any
3. The goal

N-Queens Problem: The problem is to place N queens on an NxN chessboard such that no two queens are attacking each other. According to the rules of chess, a queen can be attacked horizontally vertically and diagonally. The below figure shows the attacking positions of a queen.



The below figure shows the solution for a 5-Queen problem.



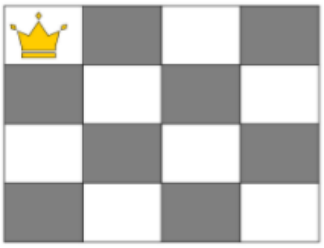
To solve this problem our goal is to place all the N queens on the board such that no two queens are in the attacking position.

This problem can be solved incrementally by placing a queen in the first available cell and the second queen in the next non-attacking position and the third queen in the third non-attacking position and so on, until we are unable to find a non-attacking position for our queens. Then we backtrack to our last queen and try to solve it.

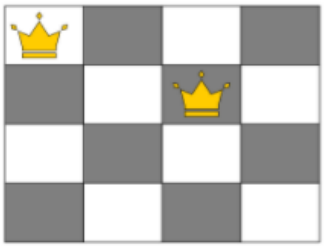
This problem can be solved using the backtracking technique since it has clearly defined constraints and we evaluate the available choices of the solution and if we exhaust all of our choices then we backtrack and explore other choices.

Let us look at the backtracking approach for the 4-Queen problem.

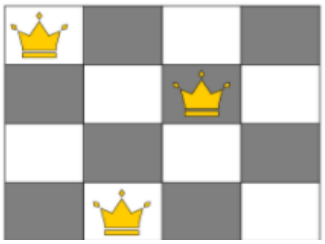
Place Queen 1 in [1,1]



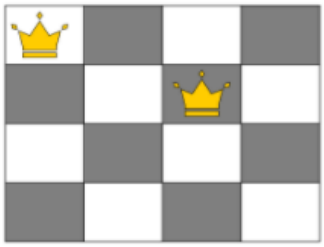
Possible Choice for Queen 2 is [2,3]



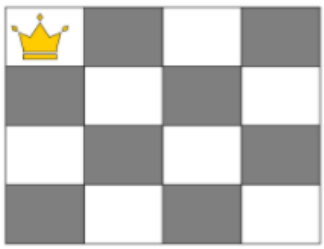
Possible Choice for Queen 3 is [4,2]



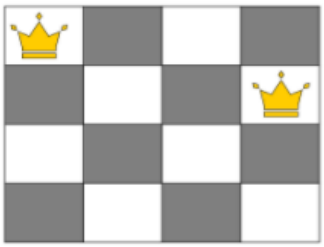
No More valid choices for Queen 4, So backtrack!



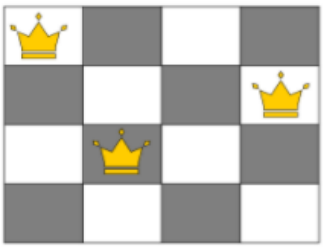
No More valid choices for Queen 3, So backtrack!



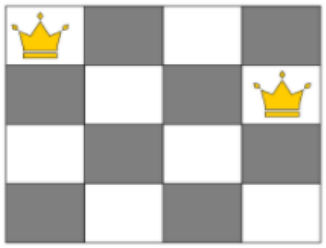
Place Queen 2 in [2,4]



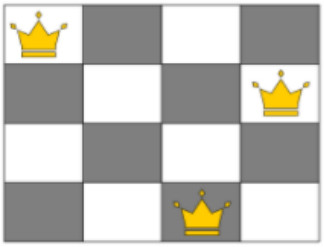
Place Queen 3 in [3,2]



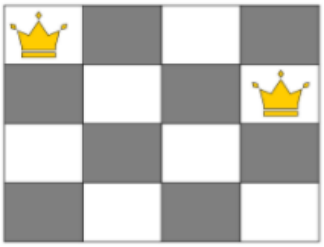
No More Possible Choices for Queen 4, so backtrack!



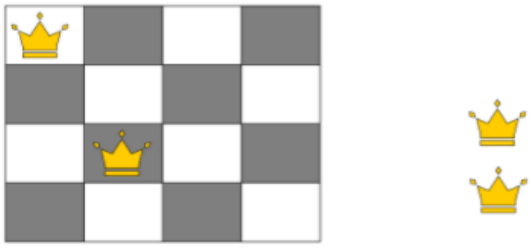
Place Queen 3 in [4,3]



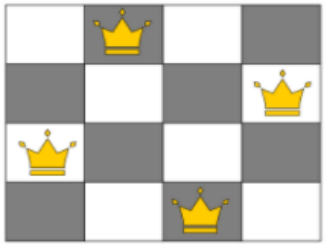
No More valid choices for Queen 4, so backtrack!



Place Queen 2 in [3,2]



And So on….. Until we solve the Entire puzzle and reach solution:



To solve this problem, we can represent the board with a 2-dimensional array (board[N][N] and default values to 0. If a queen is placed in a position we mark it as 1. The solution for the 4-Queen problem demonstrated above would be:

[[0, 1, 0, 0],

[0, 0, 0, 1],

[1, 0, 0, 0],

[0, 0, 1, 0]]

```
def n_Queens(N):
    '''
    Returns an NxN board with N-queens placed in correct positions.
    '''
    board = [[0 for x in range(N)] for x in range(N)]
    solve_n_Queens(board, 0, N, N)
    return board

solve_n_Queens(board, row, N, remaining):
    #base case
    if(remaining == 0) : return True
    for col := 1 to N
        if(is_attacked(row, col, board, N)):
            # skip the cell and continue
```

```
        else:
            # Place the queen and recursively solve for remaining queens
            board[row][col] = 1
            if(solve_n_Queens(board, row+1,N, remaining-1)):
                return True
            #backtrack if any placement results in no solution
            board[row][col]=0
    return False

'''
Check if row and col positions are valid for a queen placement
'''
is_attacked(row, col, board, N):
    # check row
    for i in range(N):
        if (board[row][i] == 1):
            return True
    #Similary check column and Diagnally
```

Explanation

We are passing the board, the first row value, the number of queens and the remaining number of queens to be solved, to our recursive function

```
solve_n_Queens(board, row, N, remaining):
```

When the remaining queens to be solved are 0, we have reached our base case

```
if(remaining == 0) : return True
```

We are iterating through each cell in a row and checking if it can be attacked.

```
for col := 1 to N
    if(is_attacked(row, col, board, N)):
```

If the position is not attacked, we place our queen in the cell, and solve for the remaining queens. We do so, by incrementing our row since we cannot place another queen in the same row.

```
board[row][col] = 1
if(solve_n_Queens(board, row+1,N, remaining-1)):
```

If the above function call returns false, this implies we cannot place a queen in the row and column, then we backtrack to the previous queen by resetting the cell value to 0

```
board[row][col]=0
```

“is_attacked(row, col, board, N)” is a constraint checking function, which validates that no other queen is present in the row, column and in diagonal positions.

Time Complexity;

The outer for loop executes for n times. “is_attacked(row, col, board, N)” has a time complexity of O(n). The if condition where is_attacked() function is called would have a time complexity of O(n**2).

The recursive call is called n-1 times. Since it is in the for loop that is executed n times, iti would have a running time of nT(n-1), if T(n) is the running time of the solve_n_Queens() function.

We can write the recurrence relations as

T(n) = O(n**2) + nT(n-1)

Solving this recurrence relation would give the worst-case time complexity of O(n!).

This can be analyzed at a high level as, the first queen can be filled in n positions in the first row, the second queen can be filled in n-1 possible positions, the third queen in n-3 possible [positions and so on. Resulting in time complexity of O(n * (n-1) * (n-2) ….) = O(n!)

Exploration: Backtracking – Combination Sum Problem

Introduction

More backtracking problems.

Combination Sum Problem

Problem: Given a sorted array of positive integers nums[] and a sum x, find all unique combinations of integers from the nums[] array whose sum is equal to x. Any integer in the array can be chosen an unlimited number of times.

Additional requirements:

Elements in the combination (e1,e2,e3,.....,ex) must be in the ascending order. (i.e. e1≤e2≤e3≤.....≤ex)

The combinations should be sorted in ascending order, i.e. the combination with the smallest first element should be printed first.

The solution should not contain duplicate combinations.

Return [], if there is no solution.

Example 1:

nums[] = [2 , 3 , 6 , 7]; x = 7

Output:

[2,2,3]

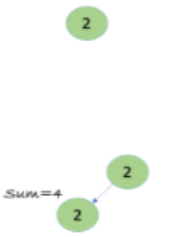
[7]

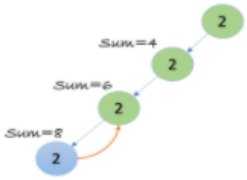
Example 2:

nums[] = [1] ; x= 1

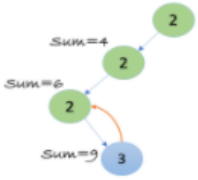
Output: [1]

Solving this problem will be similar to the permutations problem that we saw in the previous module. We will iterate through each element in the array from the start of the array. When the sum of the combination of the choices is grater than the target value, we backtrack to the last selection and try a different choice. The below slideshow shows how the first result is built for example 1.

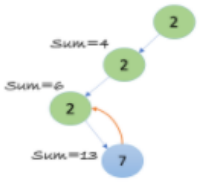




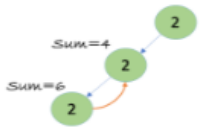
Sum is greater than target sum, so backtrack.



Sum is greater than target sum, so backtrack.



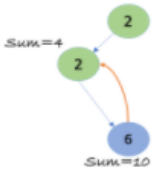
Sum is greater than target sum, so backtrack.



Result: [[2,2,3]]

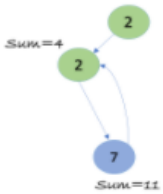
Sum = target sum. Add the combination to result.

Then, proceed to next choice.



Result: [[2,2,3]]

Sum is greater than target sum, so backtrack.



Result: [[2,2,3]]

Similarly, try other combinations with remaining numbers of the array.

Backtracking when the sum is greater than the target sum.

```
def combination_sum_helper(nums, start, result, remainder, combination):
 #Base case
 if(remainder == 0):
    result.append(combination)
    return
 elif( remainder <0):
    return # sum exceeded the target


 for i := start to len(nums)-1:
    combination.append(nums[i])
    combination_sum_helper(nums, i, result, remainder-nums[i], combination)
    #backtrack
    combination.pop()
def combination_sum(nums, target):
 result = []
 combination_sum_helper(nums,0, result, target,[])
```

Explanation of the code

We are making a call to the helper function with the initial array, starting index, and an array to store the result, the target sum, and an empty array where we will store running combinations.

```
combination_sum_helper(nums, start, result, remainder, combination)
```

We are keeping track of the remainder of the sum after making a choice in the remainder variable. If the remainder is 0, we have obtained a result. If the remainder is less than 0, the sum has exceeded the target hence we discard it. These will form our base cases to break out of recursion.

```
if(remainder == 0):
    result.append(combination)
    return
elif( remainder <0):
    return # sum exceeded the target
```

We loop through the array and compute the result by making the choice of keeping each number in the array. Running choices are collected in the “combination” variable.

```
for i := start to len(nums)-1:
    combination.append(nums[i])
    combination_sum_helper(nums, i, result, remainder-nums[i], combination)
```

We backtrack by removing the last section from the combination array (pop operation).

```
combination.pop()
```

Time complexity: This is an exponential time program. If n is the length of the array in the question and k is the target sum. Time complexity will be O(n**k)

Exploration: Greedy Algorithms – Activity Selection Problem

Introduction

In this section, we will explore the Greedy technique to solve some algorithms. We will see the required characteristics of a problem to apply the Greedy Technique. We will also explore the limitations of Greedy Algorithms.

Greedy Algorithms

The idea between Greedy Algorithms is to make the best (Greedy) choice at each step of solving the problem with the hope that this will ultimately lead to the optimal solution.

Let us understand this with an example. Do you remember the Change Making Problem from previous modules? Given some currency denominations say [1, 5, 10, 25, 100] we need to make a change for a given amount with the minimum number of coins.

Let us say the amount is 30.

When applying the greedy approach, we will look for the best possible choice at each iteration. The first best choice will be a coin of denomination 25. This will result in the remaining amount of 5, for which the best possible choice will be selecting a coin with the denomination of 5. This will give us a solution of [25, 5]. Isn’t the greedy algorithm straight forward? But there is a limitation to this approach which we will look in a bit.

Firstly, let us look at the characteristics of problems that can be solved using Greedy technique.

1. Greedy Choice Property: A globally optimal solution can be obtained by making a locally optimal choice. That is, for sub-problems, if we make the best possible choice (locally greedy choice) this would result in the optimal solution for the bigger problem (Global optimal solution).
2. Optimal Substructure: We have seen this term in the dynamic programming section. A problem is said to have an optimal substructure if the optimal solution for the problem can be obtained by taking the optimal solution for the sub-problems.

Observe how the change-making problem satisfies these two properties.

For the change-making problem, the sub-problem is to find the greedy choice of amount 30 (solution = {25}), then to find a greedy choice for amount 5 (solution = {5}) (Optimal Structure). We are able to find the solution to the whole problem by making locally best choices (Greedy Choice Property)

To make a greedy algorithm first identify the optimal substructure in the problem and devise a way to iteratively go through the sub-problems to find the solution.

Activity Selection Problem

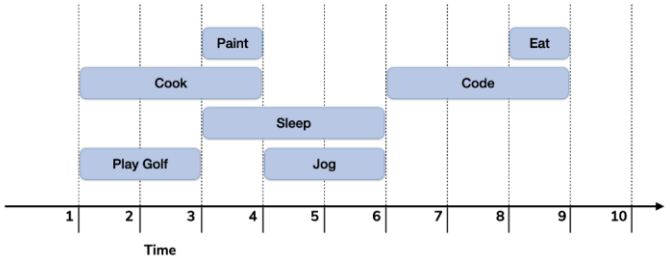
You are given a list of activities {a_1, a_2,...a_n} with their start times [s_1, s_2,...s_n] and end times [e_1,e_2,....e_n].

Example activities: {Play Golf, Paint, Cook, Sleep, Jog, Write Code, Eat]

start times: [1, 3, 1, 3, 4, 6, 8]

end times: [3, 4, 4, 6, 6, 9, 9]

You are equally interested in all of them. Your goal is to maximize the number of activities that you can perform. You cannot choose overlapping activities, for example, 'cook' and 'paint'. You can choose activities that immediately start after the previous one ends, for example, 'Play Golf' and 'Paint'.



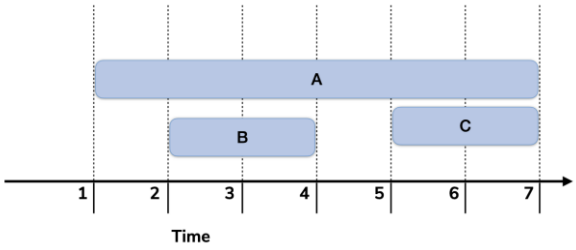
Take a few minutes to think about how you would solve this problem. Suppose you don't know any of the programming techniques and try to think greedily to solve the problem.

Think greedily to pick the best choice we think of these approaches:

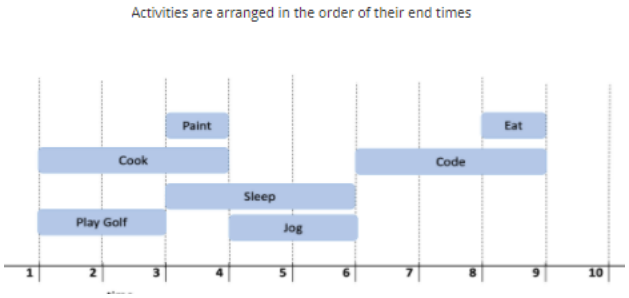
Approach 1: We can arrange the activities in the order of start times and pick one after another such that non overlaps. But would this give the maximum number of non-overlapping activities? Here is a counter example:

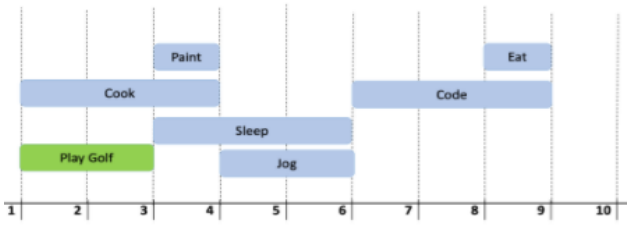
For below activities, our approach would give us the solution to be : {'A'}

But, the optimal solution comprises of two activities: {'B', 'C'}



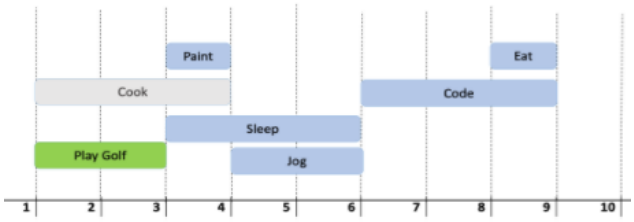
Approach 2: Arrange the activities in the order of the end times and pick the ones that end first followed by the next possible non-overlapping activity that ends first and so on. This sounds like a reasonable approach.





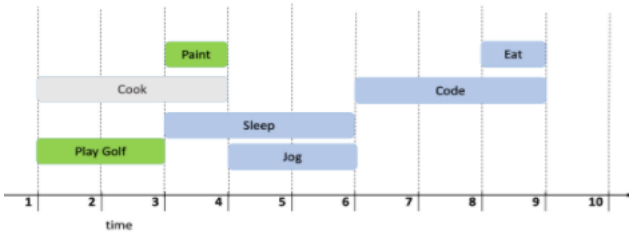
Now for next selection

Cannot pick the ones that overlap with already picked activity (Greyed in figure below)



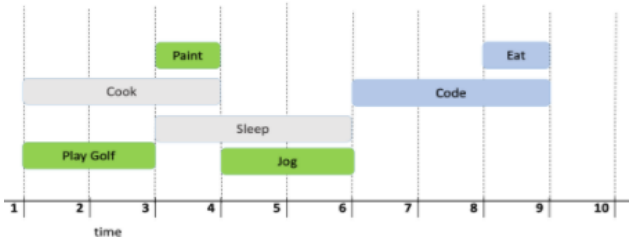
Picking next activity with smallest finish time

Cannot pick the ones that overlap with already picked activity (Greyed in figure below)



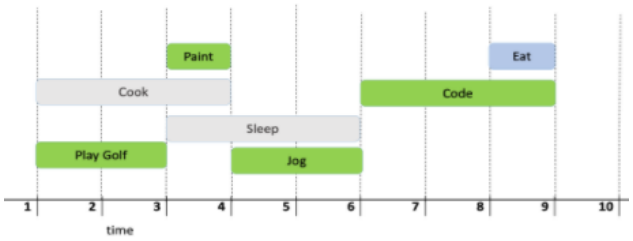
Picking next activity with smallest finish time

Cannot pick the ones that overlap with already picked activity (Greyed in figure below)



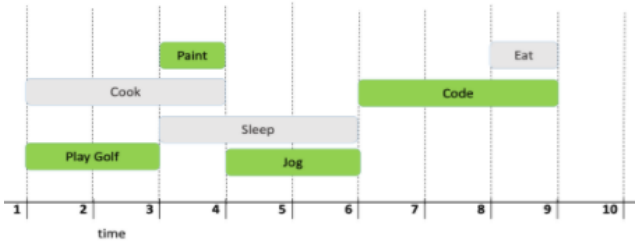
Picking next activity with smallest finish time

Cannot pick the ones that overlap with already picked activity (Greyed in figure below)



Picking next activity with smallest finish time

Cannot pick the ones that overlap with already picked activity (Greyed in figure below)




```
activity_selection(activities, start_times,end_times):
  result = []
  blocked_time = 0
  for i := 1 to len(activities):
    if(start_times[i] >= blocked_time):
      result.append(activities[i])
      blocked_time = end_times[i]
return result
```

Pseudo Explanation

Since the activities are sorted, we iterate through them in order. To begin, we choose the first activity that has the earliest finish time amongst all of the activities. We update the “blocked_time” with the end time of the chosen activity so that we can pick our next activity that starts immediately after the “blocked_time”. We repeat until we cover all the activities.

Time Complexity: The running time of our solution will be O(n). Space complexity will be O(n).

Correctness Proof

Let us prove the correctness of our algorithm. There are different ways of proving a greedy algorithm. We will prove our algorithm's correctness using two methods.

1. Greedy Stays Ahead Method (This is done in two parts):

Part 1: At each iteration the algorithm finds the best possible solution. i.e. the algorithm has found the best possible solution, which is optimal.

Part 2: The solution of the problem is optimal. i.e. The number of non-overlapping problems found by the Algorithm is an optimal solution.

The Activity Selection Problem has many real-life applications. One classic application is scheduling a room for multiple competitions in an event so that the maximum number of competitions can be fit in.

The advantages of Greedy algorithms are: These are often efficient compared to other algorithms. These are easy to implement.

Limitations

Apply Greedy technique that we discussed denominations: [1,7,10,25,50] to make a change for an amount of 14, using a minimum number of possible coins.

{10, 1, 1, 1, 1}

Explanation

For the amount 14, the best choice of the denomination is {10}, then for the remaining amount of 4 best choice is {1,1,1,1}. But this the optimal for this change-making problem? No, {7, 7} is the solution with a minimum number of coins. This brings us to the limitations of the greedy algorithms.

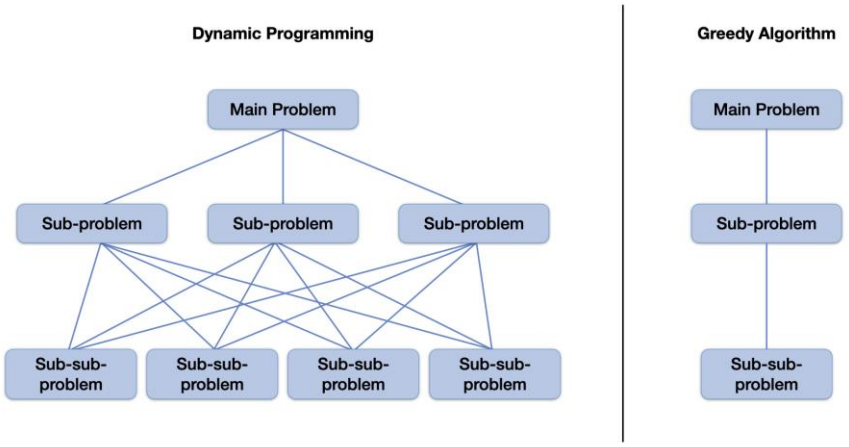
The greedy algorithm does not always give us the correct solution, rather we get an approximate solution most of the time. To get a correct solution it is very important to first identify the optimal substructure and check that in each step we consider the solution that would be closest to an optimal solution. If a problem cannot be solved by greedy technique, we can solve it using the dynamic programming technique.

The limitations of the Greedy Algorithms:

1. Greedy solution would not always result in an optimal solution. It is difficult to verify the correctness of the solution
2. Hard to design: Sometimes it is challenging to find the right greedy approach to solve a problem.

Greedy Algorithm Vs. Dynamic Programming

As discussed above the Greedy technique works with problems with optimal substructure, similar to dynamic programming problems. But the difference is that in Greedy algorithms we can right away find the optimal solution in each subproblem and move on. Another difference is that in Greedy algorithms, each problem depends only on one subproblem unlike in dynamic programming where we had to check the best possible solution from previously solved subproblems.



Exploration: Greedy Algorithms – Minimizing Lateness Problem

Introduction

In this section, we will look into yet another problem that can be solved using the Greedy technique

Minimizing Lateness Problem

Problem: You have a list of assignments [A, B, C, D]. You are provided with the time that it takes to solve these assignments, timetaken[t1, t2, t3, t4] and the deadline of each of the assignment, deadline[d1, d2, d3, d4].

Where t1 corresponds to the time taken to complete Assignment A, which is due on d1.

Your goal is to complete the assignments in such a way that you minimize the total lateness for the assignments. You need to return the minimum (Maximum lateness).

Assume units for timetaken and deadline are days.

For example:

If assignments = [A, B, C, D]; timetaken=[1, 2, 3, 4] and deadline=[2,4,5,7].

	A	B	C	D
timeTaken	1	2	3	4
Deadline	2	4	5	7

One way of doing the assignments is to start with assignment A. It will take 1 day to finish it and will be submitted on Day 1, resulting in 0 lateness. Next, we start with assignment B. It will take 2 days time and will finish on Day 3, again resulting in 0 lateness since the due date is Day 4 for assignment B. The below

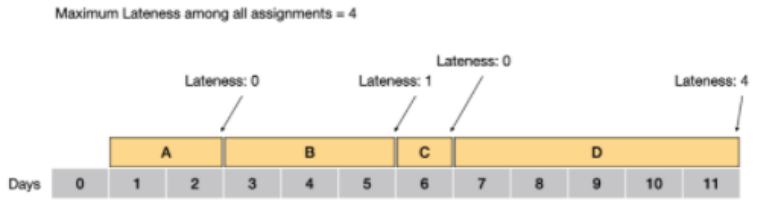
animation demonstrates this. Overall we will obtain lateness of 0 for assignment A, 0 for assignment B, 1 for assignment C and 3 for assignment D. So the maximum lateness, if we do assignments in this order, is 3. We can do the assignments in a different order and obtain a different amount of maximum lateness.

Consider another example:

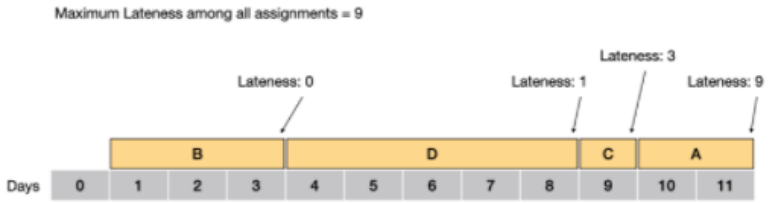
assignments = [A, B, C, D]; timetaken=[2, 3, 1, 5] and deadline=[2, 4, 6, 7].

Following are two random ways of doing these assignments. Observe how the lateness is calculated in each case.

Option 1:



Option 2:



Solution

This problem can be solved in different ways. When making a selection of the assignments we can first pick the assignment that would take minimum amount time to perform, or we can pick an assignment which is due first.

Let us analyze each of the techniques for solving this problem.

Approach 1

First, pick assignment with the shortest time to perform

This would not always give us an optimal solution, one counterexample is:

	A	B
Time Taken	1	100
Deadline	101	100

This approach will result in the selection of Assignment A, followed by Assignment B. It will result in maximum lateness of 1.

But the optimal solution is to first do assignment B, then do assignment A. This will result in maximum lateness of 0.

Approach 2

Another approach would be to pick assignment with minimum slack time, i.e. those which would have no extra time based on the amount of work needed (deadline – TimeTaken)

This would not always give us an optimal solution, one counterexample is:

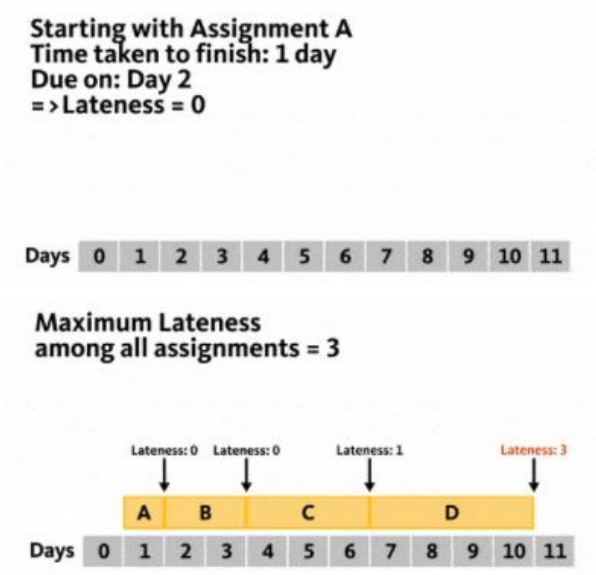
	A	B
Time Taken	1	10
Deadline	2	10

This approach will result in the selection of assignment B, followed by assignment A, which would result in the lateness of 9.

But, the optimal solution will be to first perform assignment A, followed by assignment B, which would result in the lateness of 1.

Approach 3

The third approach will be to first select assignments based on their due date, and the minimum deadline assignment would be selected first. The following animation demonstrated this approach for assignments = [A, B, C, D]; timetaken=[1, 2, 3, 4] and deadline=[2, 4, 5, 7].This seems to be a reasonable one to proceed with.



For this, we would first sort the assignments in the increasing order of their deadlines and then calculate the minimum lateness. Pseudocode for this approach is:

```
min_Lateness(Name, TimeTaken, DeadLine):
    sort Name, TimeTaken, Deadline based on deadlines such that d₁ <= d₂ <= d₃ <=...<= d
ₙ

    start_Time = 0
    max_Lateness = 0
    schedule = []
    finish_Time = 0

    for i:= 1 to n:
        finish_Time ←← start_Time + TimeTaken
        if(finish_Time > dᵢ):
            max_Lateness ←← max( max_Lateness, finish_Time - dᵢ)
        start_Time ←← finish_Time
        schedule.append(Nameᵢ)
    return max_Lateness, schedule
```

Complexity Analysis

The time complexity for this algorithm will be dominated by the time take to sort the deadlines. Which would take O(n log n) if we use the efficient sorting technique. Space complexity will be Θ(n)that is used to store the final schedule

Exploration: Greedy Algorithms - Huffman Coding Problem

Introduction

In this section we explore one popular problem called Huffman Codes that is solved using a Greedy Algorithm. As you go through this section, analyze the approach taken towards solving and proving a greedy algorithm.

Huffman Codes

We know that all the text, numbers etc are represented in binary format in our computers. Consider the problem of representing some text, say - "this is a big secret message" in binary, but the condition is that we need to use minimum possible binary bits and it should be easy decode the text back from binary. Huffman Coding one of the commonly used techniques to solve this problem (this problem is called lossless data compression). Let us explore this technique.

Let us assume our imaginary message to be "aaaaaaaabbbcdeeee" (8 a's, 3 b's, 1c, 1d, 4 e's)

Approach 1 to solve the problem - assign a binary representation for each character:

Let us assign each character with some binary representation.

Characters	Binary Code
a	000
b	001
c	010
d	011
e	100

The binary text will be: 000 000 000.....100. Our message length will be 51 bits. Is it optimal? Maybe we can shorten it.

Approach 2 – assign shorter code to the most frequent character:

Can we optimize the length of the message by making it shorter, by using a different binary code representation in such a way that the length off the binary code is variable based on the frequency of the occurrence of the character? We can present the characters that appear a maximum number of times (high-frequency characters) with a shorter code and characters that appear least number of times (low-frequency characters) with longer code. So, the idea is to use a variable-length code.

Characters	Frequency	Binary Code
a	8	1
b	3	00
c	1	001
d	1	101
e	4	01

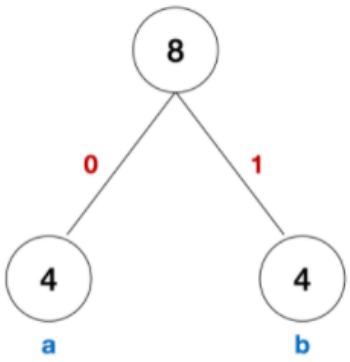
Now using this code, our message will be 111111110000000110101010101. Our message length is now 24 bits. But can we decode these bits back to our message? When decoding should we read the last bit as d, or read last two bits as e? Both appear correct based on the code.

The approach appears optimal but decoding seems to be challenging. So we need to design a better approach to represent the character based on the frequency in such a way that we don't have ambiguity.

Approach 3 – Huffman code:

The idea is to represent the characters in the form of a binary tree, such that the characters with the least frequency will have the maximum depth. Here is an example:

If there are two characters a, b both with the same frequency of 4. Our binary tree will be as shown below. Each node represents the frequency of occurrence of the character. We assign a binary symbol (0 or 1) to each branch.



Based on this tree, we can represent A:0 and B:1.

The below video shows the method to construct a binary tree and how to decode the message. We first take the two characters with the lowest frequency and form a tree with two nodes. The root node will

be the sum of the frequency of the lowest two nodes that we considered. Then we follow the same procedure with the remaining characters but we also include the frequency of the root node to build the tree.

When constructing the tree, at each step we want to pick two smallest frequencies from the available frequencies including the one in our tree's current root. For the characters and their frequency of occurrence given below, construct a binary tree to generate Huffman codes. Based on your tree answer below questions.

Characters	Frequency
S	3
T	5
V	6
W	4
Z	2

Algorithm

Now that we understand the approach to solve the problem, let us write our algorithm to implement it. For this, our knowledge of data structures will come in handy. As you might have noticed that while building the tree we are repeatedly asking - what is the minimum frequency among the list of frequencies. This gives us an idea that we need to sort the frequencies. We would be generating new frequency values (by combining two minimum frequencies) and this new value needs to be in sorted order with the previous list of frequencies so that we can get two minimum frequencies repeatedly, this would be done until we build the complete tree. Hence, we need to represent our frequency values in a way to perform these operations with the best time complexity. We can use the min-heap data structure for this purpose.

Consider these characters and their corresponding frequencies.

Let us see how the pseudocode works.

Character	Frequency
P	3
S	5
T	6
M	4
N	2

```
Create node for each character
for i <-- 1 to n:
    create min-heap Q with each node frequency as a key
```

The first step is to create a node for each character.

Next, when we create a min-heap (also called min Priority Queue) all the characters will be sorted in the order of their frequencies.

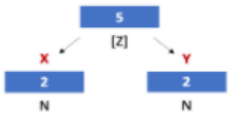

```
while(len(Q)>1):
    x = pop minimum value from Q
    y = pop minimum value from Q
```

We will take the least two frequency characters and remove them from the queue.


```
create new node z
z.key = x.key + y.key
z.leftChild = x
z.rightChild = y
```

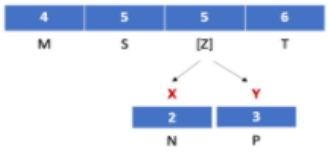
We will create a new node [z] that holds the sum of frequencies of x and y as it's priority.

The nodes x and y are added as left child and right child of this new node.



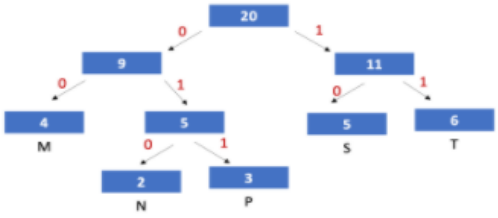
```
insert z into Q
```

Now, we will insert this new node into the Queue Q.



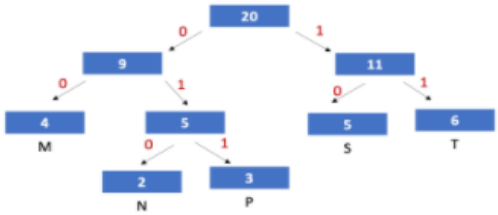

```
x = pop minimum value from Q
y = pop minimum value from Q
create new node z
z.key = x.key + y.key
z.leftChild = x
z.rightChild = y

insert z into Q
```

We will repeat this procedure until the Q is left with only one node (ie. The root node)




```
return the root value from Q
```

We end by returning the node from Q that has the root value.



Pseudocode:

```
BuildHuffman(characters[1..n], frequencies[1..n]):
    Create nodes for each character
    for i <-- 1 to n
        create min-heap Q with each node frequency as a key

    while(len(Q)>1):
        x = pop minimum value from Q
        y = pop minimum value from Q
        create new node z
        z.key = x.key + y.key
        z.leftChild = x
        z.rightChild = y

        insert z into Q

    return the root value from Q
```

Time Complexity:

Insert operation into Q takes log n time and we are performing this 2n-1 times, each time taking out two nodes and adding a new node.

Exploration: Graph Introduction and Traversal

Introduction

A majority of the real-world problems can be divided into subproblems which are represented as nodes and can be constructed to form a graph. For example, if Amazon wants to optimize their delivery routes, they can be represent their delivery locations as nodes and form a graph connecting these nodes and solve their problem using the knowledge of solving graph problems.

You might have learned about graphs in your Data Structures course. In this section, we will take a quick look at the terminology and look at the ways of traversing a graph. As you go through this section try to answer the following questions.

Graph Introduction

Strong knowledge in graph fundamentals will enable us to better approach the graph-related problems. Read the content presented in the below slides to get familiar with the terminology involving graphs.

Graph is a mathematical representation of a set vertices (or nodes) connected with edges.

Note that a graph is represented as (V, E)

V: set of vertices

E: set of edges

For Example:

Mathematical presentation of undirected graph shown will be:

V = {A , B , C, D}

E = {(A,B) , (A,C), (B,C), (C,D)}

|V| represents cardinality of V. |V| = 4

|E| represents cardinality of E. |E| = 4

Undirected Graph:

Edges have no directions

V = {1, 2, 3, 4, 5}

E = {(1,2), (2,3), (3,4), (1,4), (1,5), (4,5)}

Note: Order does not matter in each tuple in E.

Directed Graph (also called Digraph):

Edges have directions

V = {1, 2, 3, 4, 5}

E = {(1,2), (2,3), (3,4), (1,4), (1,5), (4,5) , (4,1)}

Note: In E order matters in each tuple,

Observe we have both (1,4) and (4,1) to show edge between 1 and 4

Indegree: Number of edges coming into a node (we need to observe the direction of the arrow)

Indegree of V = 3

Outdegree: Number of edges going out from a node (We need to observe the direction of the arrow)

Outdegree of U = 2

Degree: Number of both incoming and outgoing edges.

Degree of U = 2

Subgraph

G' is a subgraph of Graph G if

The edges of G' are subset of edges of G

The vertices of G' are subset of vertices of G

The graph with green nodes is a subgraph

Spanning Subgraph

A subgraph that contains all the vertices of G

(it may not contain all the edges)

Directed Acyclic Graph (DAG) is a graph that has no cycles.

This is DAC

This is not DAC (there is cycle between (1,2,3,4))

Tree is an acyclic graph.

Note, not all acyclic graphics are trees.

Representation of Graphs

Graphs can be represented in two ways

1. Adjacency Matrix
2. Adjacency List

Adjacency Matrix

A graph can be represented of a matrix of size |V| x |V|. Cell (I,j) is marked as 1 if there is an edge between I and j vertices. Below is an example. Cell (1,2) is marked 9 while cell (2,1) is marked as 1 since there is a directed edge between 2 and 1.

This representation requires a space of theta(|V|^2) to store the graph and access time to check if a node takes theta(1).

Adjacency List

In adjacency list, we have vertices in the form of a list, and each element (a vertex) in the list contains another list (or array or linked list) with the elements that are connected to the vertex.

The array has |V| elements and each element has its adjacent vertices as a list, which are in total |E|. Hence it will need a space of theta(|V| + |E|), this would be for directed graph. In the case of an undirected graph, each edge will be directed, making it a total of 2|E| edges.

Graph Traversal

To traverse each vertex in the graph, tree based algorithms are used. The two ways of traversing in a graph are:

1. Depth First Search
2. Breadth First Search

In Depth-First Search we start at the root and go down an edge as far as we can, then we trace back to cover other depths.

In Breadth-First Search we start at root and traverse across each level.

DFS uses the stack data structure for implementation and BFS uses queues, both have a time complexity of O(|V|+|E|)

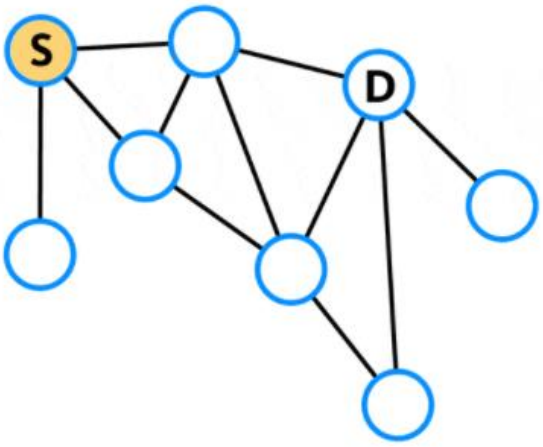
Breadth-First Search Applications

BFS is used to solve many real-world problems: in analyzing networks, in GPS navigation systems, in crawlers in web engines, and finding the shortest path.

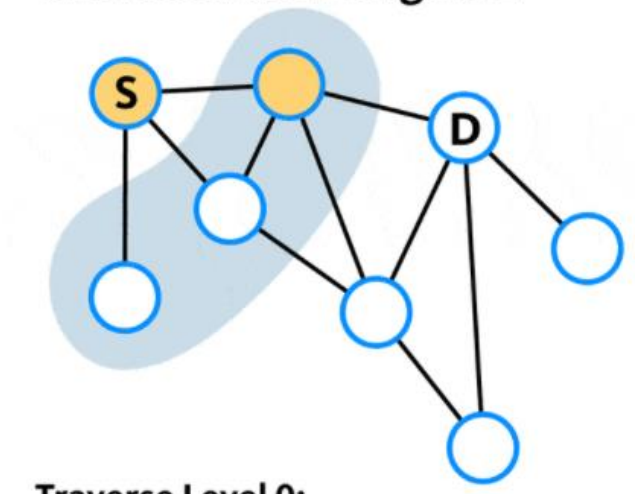
Finding the Shortest Path: We are given a graph and the source node (S) and a destination node (D). We need to find the length of the shortest path between the source and the destination.

We can start at the source and traverse the graph in a breadth-first fashion. We increment the length of the path until we reach the level where the destination node is located.

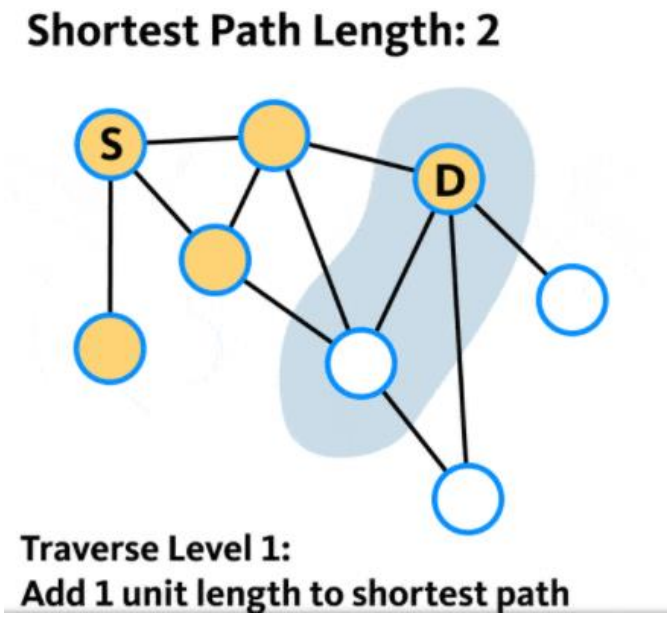
Shortest Path Length: 0

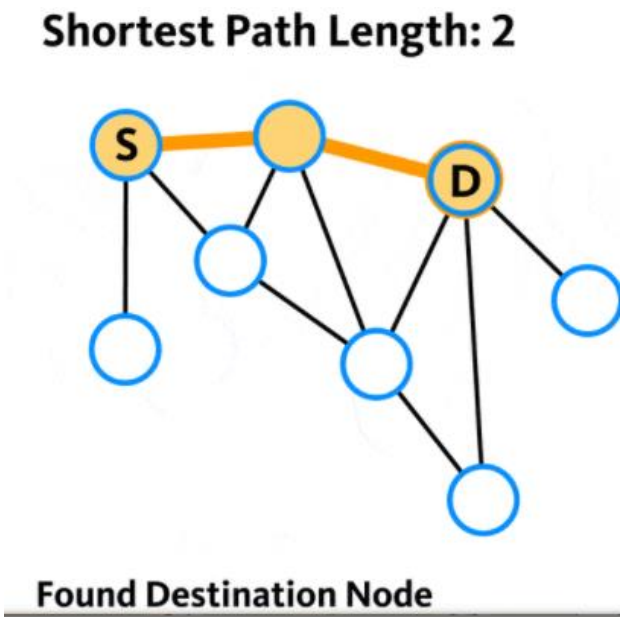


Shortest Path Length: 1



Traverse Level 0:
Add 1 unit length to shortest path





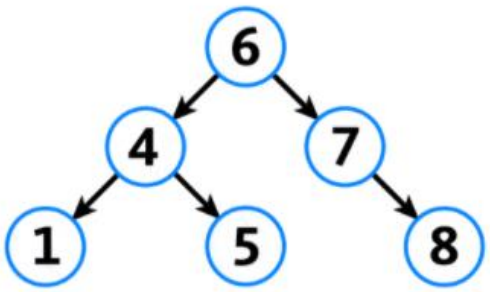
This process will take a time complexity of O(|V| + |E|)

Depth-First Search Applications

DFS is used in topographical sorting, in the traveling salesman problem, in solving maze puzzles, etc.

In-order traversal of a Binary Search Tree: Given a binary search tree print the nodes in the sorted order.

The binary search tree is a binary tree that holds the property that the left children in the tree will be smaller the root node and the right children will be larger than the root node. Traversing binary search tree in DFS order will return the nodes in the order.



Result: 1, 4, 5, 6, 7, 8

This process will take a time complexity of O(|V| + |E|)

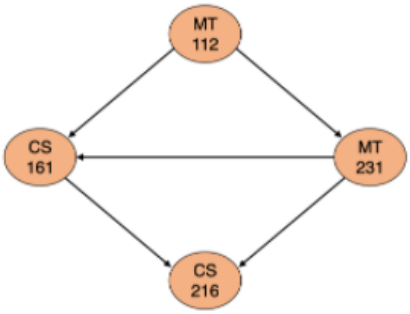
Exploration: Topological Sort

Introduction

You might be aware that most of the courses that you take here at OSU have some other course as prerequisites. For example, the current course CS 325 has CS 261 and (CS 225 or MTH 231) as prerequisites. When we pictorially represent the courses in the form of nodes and connect them with their dependencies we get a graph. In this section, we will look at the algorithm to find the ordering of the courses with respect to their prerequisite.

Topological Sort

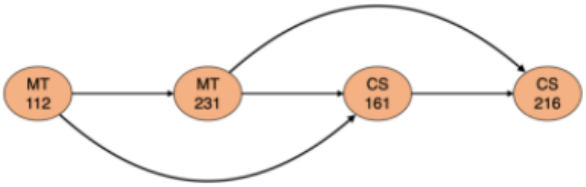
The problem of ordering the courses based on their prerequisites is a form of topological sort problem. A sample course dependencies graph is shown below.



Course CS 161 is dependent on course MT 112, similarly, course CS 216 is dependent on CS 161 and MT 231 courses.

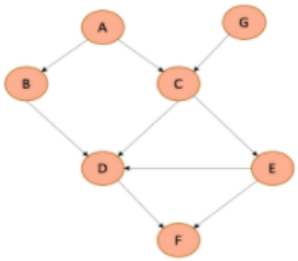
We need to find the order in which the courses can be taken.

The below graph shows one such possible order. This is the topological sort of the nodes in the graph. Note that all the arrows are pointing forward.

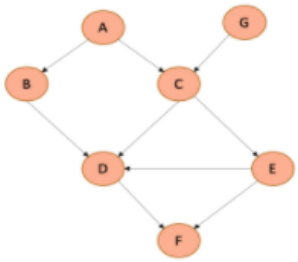


Below is not a topological sort of this graph. There should not be any arrows pointing backwards.

At a high level, we can see that to get the topological sort we are going down the depth of the graph to find the dependency flow. This can give us an idea to use DFS.

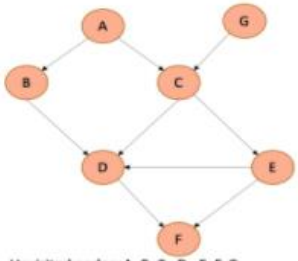


Let us see the algorithm for this graph.



We will perform DFS on this graph.

We will keep track of Visited Nodes and add the node that has been processed to a stack.

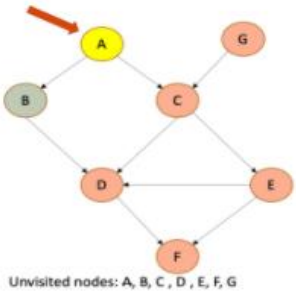


Unvisited nodes: A, B, C , D , E, F, G

Visited set:

Result (Stack):

Visited CurrentNode



Unvisited nodes: A, B, C , D , E, F, G

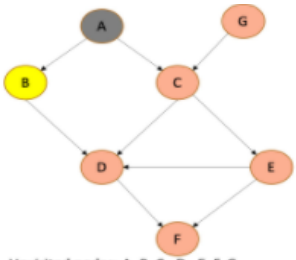
Visited set: A

Result (Stack):

Visited CurrentNode

We can start at any node, let us start at node A.

And explore the children of A are not been visited yet.



Unvisited nodes: A, B, C , D , E, F, G

Visited set: A

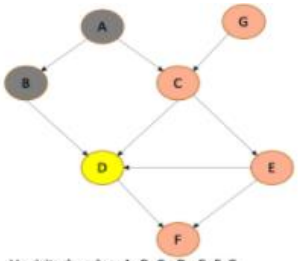
Result (Stack):

Visited CurrentNode

Visiting first child of A, B

Note: A is marked as visited.

Next we will explore B's child Nodes



Unvisited nodes: ~~A, B~~, C , D , E, F, G

Visited set: A, B

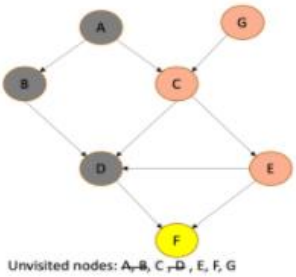
Result (Stack):

Visited CurrentNode

Visiting B's child D

Note: B is marked as visited.

Next we will explore D's child Nodes



Unvisited nodes: ~~A, B~~, C ~~, D~~ , E, F, G

Visited set: A, B

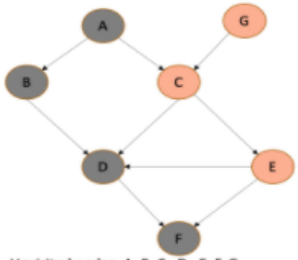
Result (Stack):

Visited CurrentNode

Visiting D's child F

Note: D is marked as visited.

Next we will explore F's child Nodes



Unvisited nodes: ~~A, B~~, C ~~, D~~ , E, ~~F,~~ G

Visited set: A, B, F

Result (Stack): F,

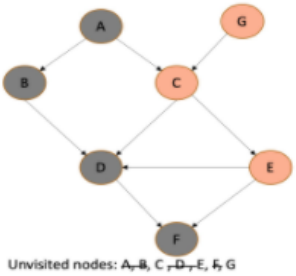
Visited CurrentNode

Visiting F's child –

F has no child nodes to process, hence add it to Result

Note: F is marked as visited.

Next we will explore F's child Nodes



Unvisited nodes: ~~A, B~~, C ~~, D~~ ~~, E~~, ~~F,~~ G

Visited set: A, B, F

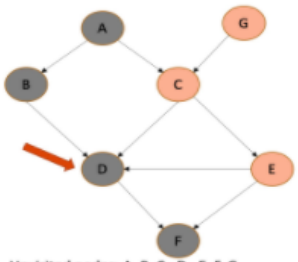
Result (Stack): F,

Visited CurrentNode

Visiting F's child –

F has no child nodes to process, hence add it to Result

Next we will go back to previous node D



Unvisited nodes: ~~A,~~ ~~B~~, C ,~~D ,~~ E, ~~F,~~ G

Visited set: A, B, F, D

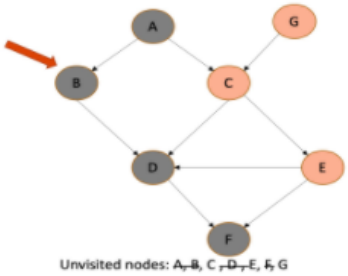
Result (Stack): F, D

Visited CurrentNode

We are at D.

D has no pending child nodes to process, hence add it to Result

Next we will go back to previous node B



Unvisited nodes: ~~A,~~ ~~B~~, C ,~~D ,~~ E, ~~F,~~ G

Visited set: A, B, F, D

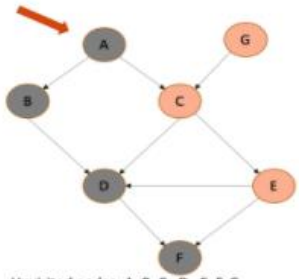
Result (Stack): F, D, B

Visited CurrentNode

We are at B.

B has no pending child nodes to process, hence add it to Result

Next we will go back to previous node A



Unvisited nodes: ~~A,~~ ~~B~~, C ,~~D ,~~ E, ~~F,~~ G

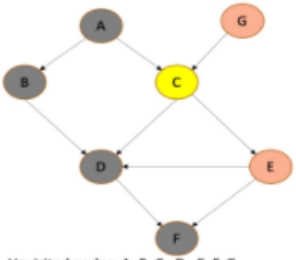
Visited set: A, B, F, D

Result (Stack): F, D, B

Visited CurrentNode

We are at A.

Next we will explore A's child Nodes (that are unvisited)



Unvisited nodes: ~~A, B~~, C ~~, D~~, E, ~~F~~, G

Visited set: A, B, F, D

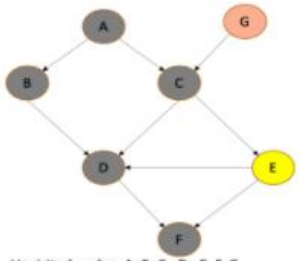
Result (Stack): F, D, B

Visited CurrentNode

We are at C.

Adding C to visited nodes.

Next we will explore C's child Nodes



Unvisited nodes: ~~A, B, C~~ , ~~D~~ , E, ~~F~~, G

Visited set: A, B, F, D, C,

Result (Stack): F, D, B

Visited CurrentNode

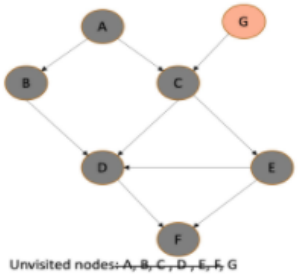
We are at E.

Adding C to visited nodes.

Next we have to explore E's child Nodes. Since it has none.

It can be marked as visited and will be added to Result.

And we will go back to previous node – C.



Unvisited nodes: ~~A, B, C~~ , ~~D~~ , ~~E, F~~, G

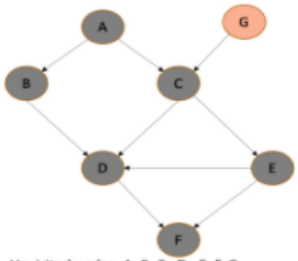
Visited set: A, B, F, D, C, E

Result (Stack): F, D, B, E, C

Visited CurrentNode

We are at C.

It has no more children to process so we will add it to result.



Unvisited nodes: ~~A, B, C , D , E, F, G~~

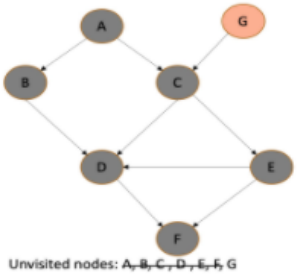
Visited set: A, B, F, D, C, E

Result (Stack): F, D, B, E, C, A

Visited CurrentNode

We will go back to A from C.

A has no more unvisited child nodes. Hence, add A to result.



Unvisited nodes: ~~A, B, C , D , E, F,~~ G

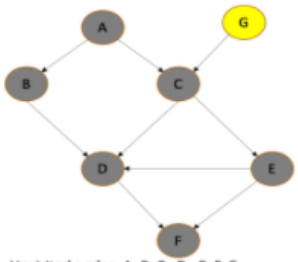
Visited set: A, B, F, D, C, E

Result (Stack): F, D, B, E, C, A

Visited CurrentNode

We covered all the nodes that are reachable from A.

Now we will check for the unvisited node – G.



Unvisited nodes: ~~A, B, C , D , E, F, G~~

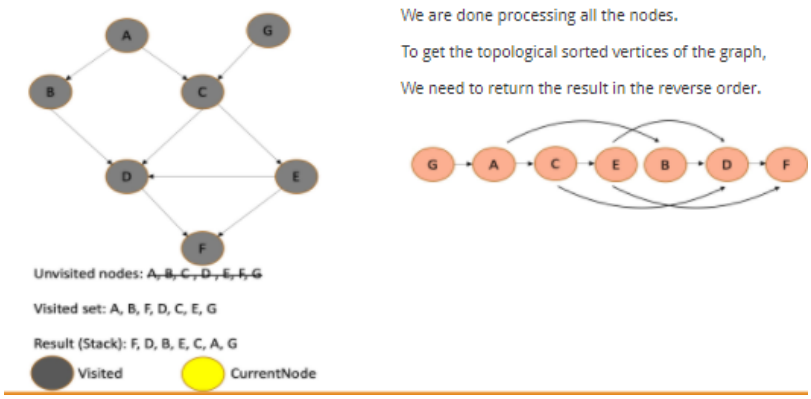
Visited set: A, B, F, D, C, E, G

Result (Stack): F, D, B, E, C, A, G

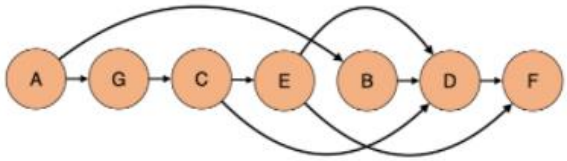
Visited CurrentNode

Now, we are at G.

G has no more children to visit, hence we will add it to Result.



Note that we can have a different order of the sorted vertices, so the solution returned by the algorithm is not the only solution. Below shows an order of vertices that can also be a solution.



The pseudocode for the algorithm:

```
topological_sort(G):
    Stack = []

    while(unvisited Nodes):
        helper_toposort(currentNode in G):
            mark currentNode as visited

            for node in currentNode.neighbours:
                if(node is not already visited):
                    helper_toposort(node)
            Stack.insert(currentNode)

    return Result in reverse order
```

We are building the result in a stack. We are using a helper_toposort() helper function to apply recursion on each vertex until we reach the depth of the graph. Once each node is processed we are adding it to the stack.

Time Complexity: Similar to DFS, this algorithm will have O(V+E) time complexity.

We can obtain topological sort for 'directed acyclic graphs' but not for undirected graphs or graphs with cycles.

Why? In an undirected graph when we pick two nodes connected with the edge we cannot say which node is dependent on the other one, so order doesn't come into the picture. In a graph with cycles, for example if a--> b -->a, then it is difficult to say which node takes priority.

We can solve this problem using a naive approach by iterating through each node in the graph. But that will not be efficient. The problem wants us to cover all the nodes in the graph. Let us see how to solve it using graph traversal technique.

Applications:

Topological sort is in many Computer Science applications. When you are installing a package which have a couple of modules that are dependent on each other. Then the topological sort algorithm is used to build a graph of the modules and decide the order in which the modules should be installed. Other applications are in problems like job scheduling.

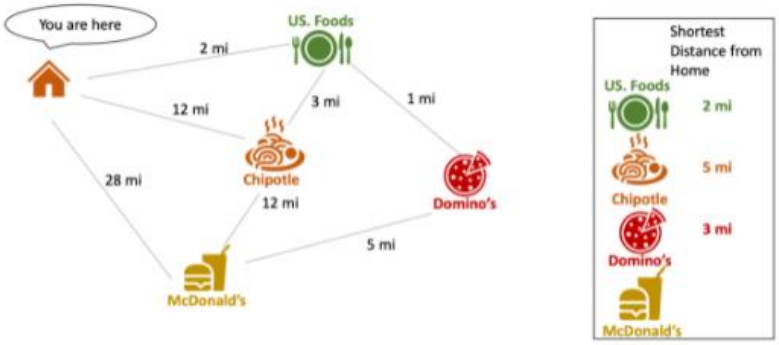
Exploration: Dijkstra's algorithm

Introduction

In this section, we will explore an algorithm that can be used in applications like Uber, Lyft to find the shortest path from a source to destinations, this is called Dijkstra algorithm. It is a Dutch name, where j is either silent or is pronounced like y. Dijkstra is pronounced as Dyk(as in bike) -stra.

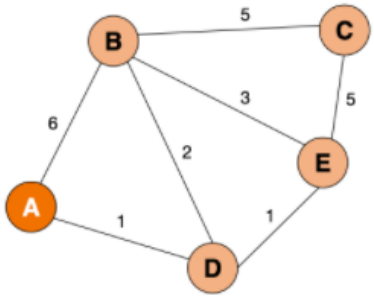
The Dijkstra Algorithm

Suppose you want to solve a problem to find 'all food places near me'. Your algorithm should return all the food places and the distance from your location.



To solve this problem we can use Dijkstra's algorithm. Did you notice that the graph is a weighted graph? Dijkstra's algorithm is used to solve similar problems that have a weighted graph but the weights should be **non-negative**.

Let us look at the Dijkstra's algorithm and find the shortest path from node A to all other nodes in the below graph. Let A be the source node and our goal is to find the shortest path to all other nodes from A.



To begin with, let us look at the method to the shortest path from A to B. If we simply use BFS it will return (A-->B). But the actual shortest path is (A-->D-->B). To solve this let us the algorithm technique that we learnt before, the greedy algorithm.

The general idea is to pick the locally best choice to obtain a globally optimal solution.

The relation that we use to find the local minimum can be written as (consider that we are calculating the distance from node u to node v):

dist[v] = min{ dist[v] , weight[u,v]+dist[u] }

```
def dijkstra(Graph, Source):

  dist[v] = infinity for all v in V

  dist[Source] = 0


  while length(unvisited)>0:

      currentNode = From unvisited nodes select node u where dist[u] is smallest for
v in currentNode.neighbours:

      dist[v] = min {dist[v] , weight(currentNode, v) + dist[currentNode] } updated c
urrentNode as visited and remove

      from unvisited list

  done
```

Explanation

We are setting the initial distance of all nodes to infinity and source node distance to 0.

```
dist[v] = infinity for all v in V
dist[Source] = 0
```

Then we are looping through all the unvisited nodes and picking the unvisited node that has the least distance from the source node.

```
while length(unvisited)>0:
    currentNode = From unvisited nodes select node u where dist[u] is smallest
```

Then, we are checking the neighbors of the selected node and calculating the shortest distance using the relation that we discussed above.

```
for v in currentNode.neighbors:
    dist[v] = min {dist[v] , weight(currentNode, v) + dist[currentNode] }
```

Finally, we have updated the visited node.

To discuss the time complexity of this algorithm we need to decide upon the data structures we want to use in the code. We are repeatedly looking out for the smallest distance node, this sounds like what a priority queue implemented as min-heap can do for us.

If we use the priority queue the pseudocode will be:

```
def Dijkstra(G,s):
    #Add nodes to min-priority queue Q with initial distance infinity
    for v in V:
        Enqueue(Q, v,'infinity')

    #in Q make source priority = 0
    DecreaseKey(Q, source, 0)

    visited = []
    #while length of Queue is valid
    while(length(Q) > 0):
        currentNode = Dequeue(Q) #pick and delete the min distance node

        visited.append(currentNode)

        for v in currentNode.neighbours:
            dist_V = min{dist_V, weight(currentNode, v) + dist_currentNode}

            DecreaseKey(Q, v, dist_V )
```

Time complexity

The while loop is executed E times for each edge and the inner for loop will be executed V times for each vertex since we are covering each vertex only once. Decrease key and Dequeue takes log V time complexity. Hence, this algorithm will have O((V+E)logV) time complexity.

Correctness Proof

Lemma: For each x ∈ VisitedNodes, dist(x) = smallestDistance(x)

dist(x) is the distance calculated by the algorithm, let us represent it as A(x).

smallestdistance(x) is the correct smallest distance from the source node to x, let us represent it as C(x)

Proof by Induction:

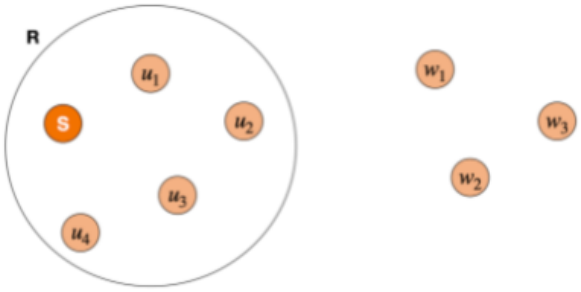
Base Case: When |VisitedNodes| = 1, The first node that is visited is the source node(S). C(x) =0, Our algorithm sets the distance from the source node to 0. So, A(x)=C(x).

Inductive Hypothesis: All the k nodes (u1, u2, ... uk) that are already in the VisitedNodes set have the distance calculated A(x)=C(x).

Inductive Step:

We need to show that the next node that is added to VisitedNode set (say w*) has the smallest distance from S.

Let us mark the VisitedNode set by region R

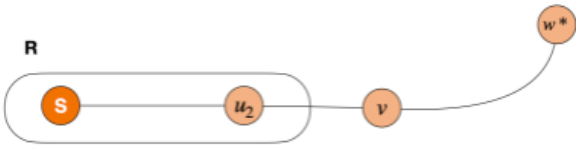


We will prove this by contradiction.

There must exist a path P from S to w* and this path would cross the region R at some point.

The length of path P, len(P) = A(u) + len(u,w*)

Assume that there is another node v outside R on path P that gives us the shortest distance from s to w*.



len(P) = A(u) + len(uv) + len(vw*)

$\implies$ A(u) + len(u,w*) = A(u) + len(uv) + len(vw*)

Since dijkstra is for non negative weights. len(vw*) >=0

$\implies$ A(u) + len(u,w*) >= A(u) + len(uv)

$\implies$ len(uw*) >= len(uv)

Now, let us invoke the Greedy property used by Dijkstra algorithm.

If there exists a Node v, that has a shorter distance than w*, v would have been visited next not w*. So, our assumption that there exists a node v on outside region R that has shorter distance than w* is false.

Hence, w* is the next node with shorter distance outside region R.

By our inductive hypothesis, all the nodes inside R have the best shortest distance. A(u) already has the optimal distance calculated.

A(u)+len(uw*) will give us optimal distance of w* from S.

Limitations of the Dijkstra Algorithm

The Dijkstra algorithm does not work with negative edges.

Exploration: Minimum Spanning Tree – Prim's Algorithm

Introduction

In this module, we will see what the minimum spanning tree is. We will explore Prim's algorithm that is used to solve the minimum spanning tree problem.

Minimum Spanning Tree

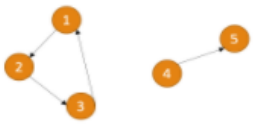
Let us look at some terminology related to graphs that will help us explore this section.

Connected Graph:

All the nodes are connected.

Disconnected Graph:

At least two vertices are not connected.



Not all vertices are connected. Vertex 4 and 5 are not connected to any of the other vertices {1,2,3}

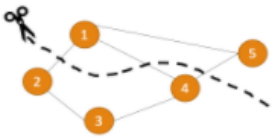
Tree is a connected acyclic graph.

Forest is a unconnected acyclic graph.

Graph that is not connected and but is composed of trees.

Above figure looks like it has two graphs but it is a single disconnected graph.

Cut: Dividing the graph into two non empty parts.

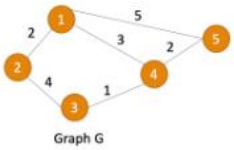


This cut divides the graph into two parts:

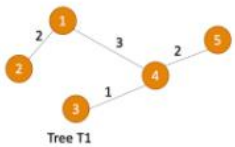
{1, 5} and {2, 3, 4}

This concept will be helpful to us in proofs of the algorithms.

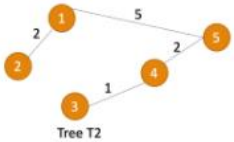
By now you know what a spanning tree for a graph is. Let's see what a minimum spanning tree is. Given a weighted undirected graph G, the minimum spanning tree is that spanning tree of G whose cost (the sum of weights of edges) is the least among any other spanning tree of G.



Graph G



Tree T1



Tree T2

A graph can have more than one minimum spanning trees.

Our problem is to find a minimum spanning tree for a given graph. This problem has numerous applications like in network design in telephone, electric, hydraulic sectors. For example, in telephone network design we want to connect all the nodes but minimize the length of the wire. In image processing, it is used to find the connected regions in an image. The algorithm is a starting step in solving other advanced graph problems.

If we take a graph that has V vertices, a total of V^(V-2) trees are possible. Hence, if you try to implement it using a brute force approach it will be exponential time, which is very slow.

There are two greedy algorithms that solve this problem:

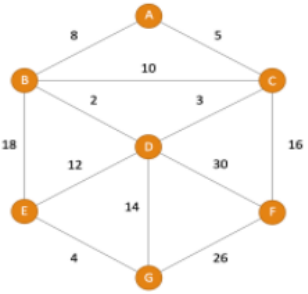
1. Prim's algorithm
2. Kruskal's algorithm

PRIM'S ALGORITHM

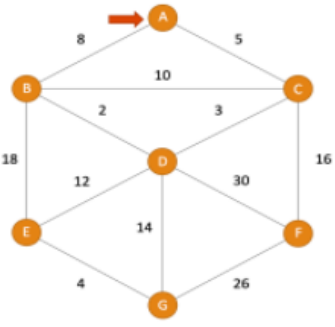
Prim's algorithm picks a random node in the graph and builds the spanning tree by using the greedy approach of selecting the edge with the last weight.

Go through the below demonstration to understand the algorithm.

Let us see the steps of Prim's algorithm on this graph.

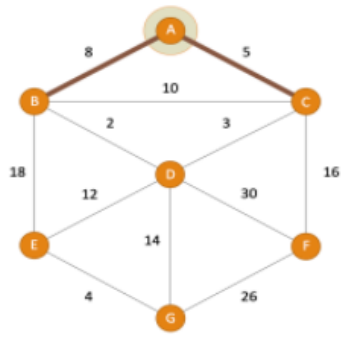


Let us see the steps of Prim's algorithm on this graph.



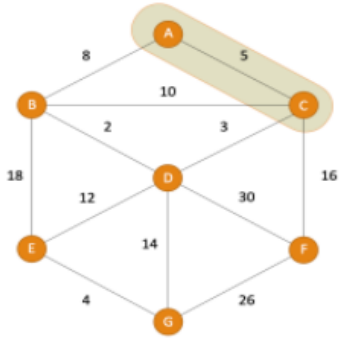
Pick any source.

Let us start at A



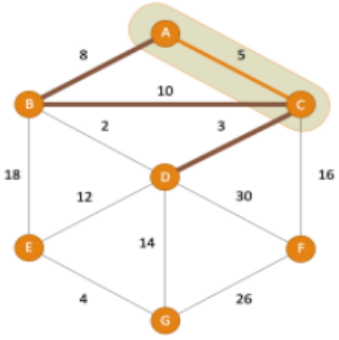
Shading the region of selected nodes with

Check the nodes that are coming out of our shaded region



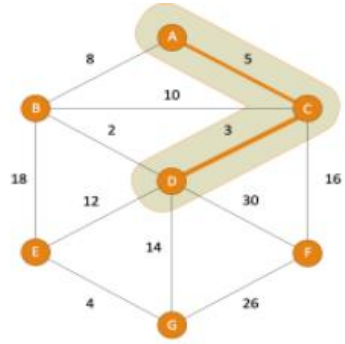
Add the minimum node to the Minimum Spanning Tree (MST).

This will expand the region of our tree.



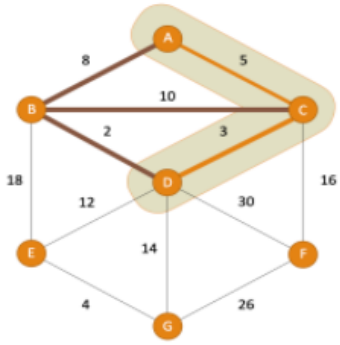
Check the nodes that are coming out of our shaded region to find the minimum node to expand the tree

(Greedy approach!)

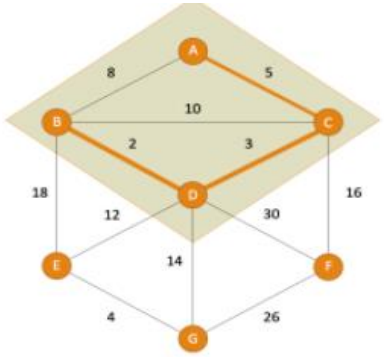


Add the minimum node to the MST.

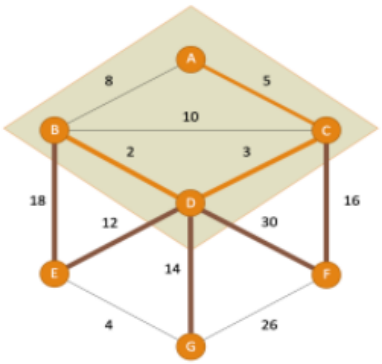
This expands the tree region further.



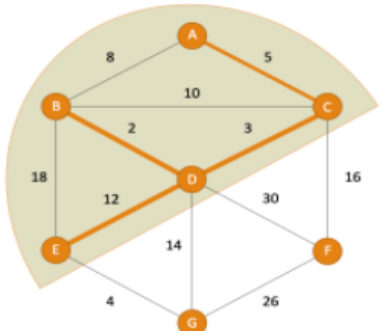
Check the nodes that are coming out of our shaded region and find the minimum node.



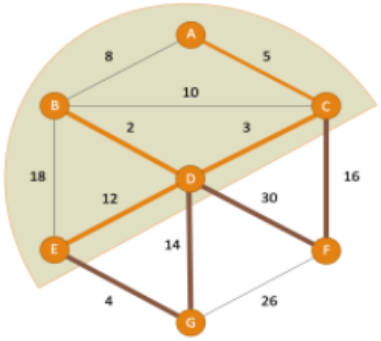
Add the minimum node to the MST.



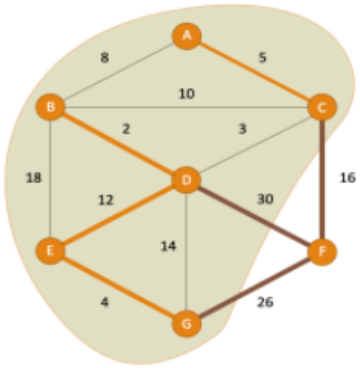
Check the nodes that are coming out of our shaded region and find the minimum node.



Add the minimum node to the MST.



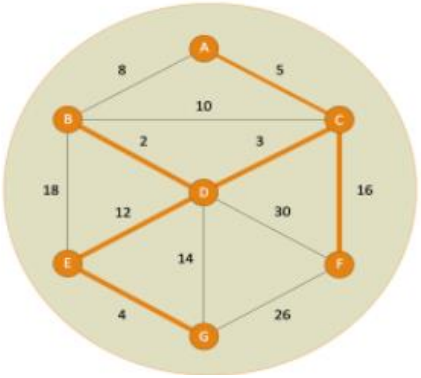
Check the nodes that are coming out of our shaded region and find the minimum node.



Add the minimum node to the MST.

And

Check the nodes that are coming out of our shaded region and find the minimum node.



Add the minimum node to the MST.

This completes building the MST

The pseudocode for this algorithm:

```
def prims(G):
    Result = {}
    visited = {} #pick one vertex from V

    while(len(visited)< V):
        find (a,b) where
            (a is in visited and b is not in visited) and (Edge(a,b) is min)

        Result.add((a,b))
        visited.add(b)

    return Result
```

If we implement it in a naïve way, it will have a time complexity of O(VE). The while loop is executed V times and if we manually find the edge with the minimum value we will need to search E number of times for each V.

Can we make this faster? Yes, using our knowledge of data structures. We can use a min-heap using priority queue and make this faster. Watch the below video for the explanation.

```
def prims(G):
    s <- pick a source vertex from V

    for v in V:
        dist[v] = infinity
        prev[v] = Empty

    #initalize source
    dist[v] = 0
    prev[v] = s

    #update neighbouring nodes of s
    for node in s.neighbours
        dist[v] = w(s,node)
        prev[v] = s

    while(len(visited)<len(V)):
        CurrentNode = unvisited vertex v with smallest dist[v]
        MST.add((prevNode, CurrentNode))
        for node in CurrentNode.neighbours:
            dist[node] = min(w(CurrentNode, node), dist[node])
            if dist[node] updated: prev[node] = CurrentNode
        visited.add(CurrentNode)
    return MST
```

CORRECTNESS PROOF

Let us prove that Prim's algorithm computes an MST.

We will prove this by induction.

Let us denote the graph with G and the edges as (e1,e2....ek.....).

The loop invariant is that after every iteration of the while loop the tree T built by the algorithm is a subgraph of the minimum spanning tree. This will give us our induction hypothesis.

Induction hypothesis: After every iteration, the tree T built by the algorithm is a subgraph of the MST.

Base case: At the start of the loop the T has only one node s which is part of the MST (Since MST should have all the vertices otherwise it will not be called a MST). The base case is true.

Induction case: We will prove this by contradiction.

At some point in the algorithm, we would have built a tree T that is part of the MST. The algorithms next adds edge e that has one end connecting to vertex u which is in the region of T. Assume that e does not belong to MST. If we add e to MST it will create a cycle, since each vertex is already connected to every other edge.

In the MST of the graph, there must be another edge e* that is connecting vertex u.

Since from vertex u among e and e*, edge e was picked by the algorithm, this implies weight(e) <= weight(e*)

Now, if we remove e* from tree M and add edge e. The overall cost of MST might reduce further. Which contradicts our definition of MST. Hence the loop invariant is maintained during the execution of the loop.

Termination: The loop would terminate when all vertices are included in the tree and the tree T built is an MST.

Exploration: Minimum Spanning Tree – Kruskal's Algorithm

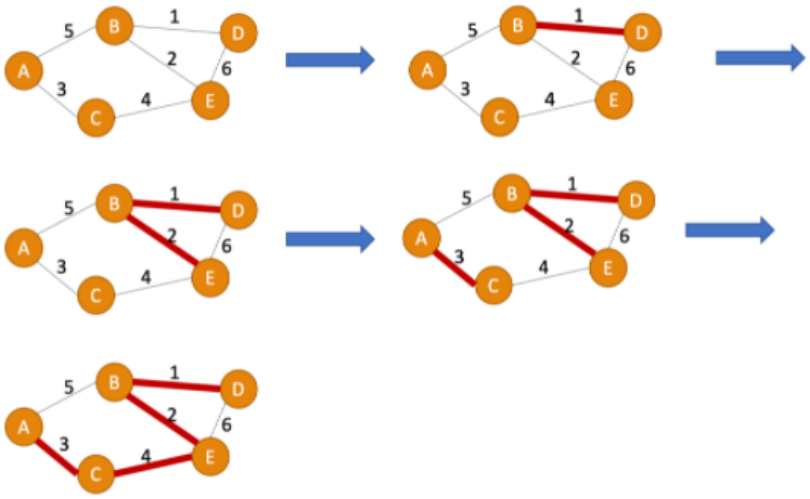
Introduction

In this section, we will explore another algorithm that is used to compute a minimum spanning tree. This algorithm is called Kruskal's Algorithm. It is slightly different form Prim's algorithm.

KRUSKAL'S ALGORITHM

Kruskal's algorithm uses a greedy approach to find a minimum spanning tree. The idea is to look for the edge with minimum length if the edge does not create a cycle then add it to the tree.

The below deck demonstrates the construction of the tree.



A. Simple pseudocode for Kruskal's algorithm can be given by:

```
def Kruskal(V,E):
    sorted_E = sort E by increasing weight
    MST ={}

    for e in sorted_E:
```

```
        if MST and e don't cycles:
            add e to MST
    return MST
```

The time complexity to implement this code using a straight forward approach: Cost to sort E will be O(E logV). The cost to find if edge e would form any cycles with MST, which can be done using BFS or DFS on the graph MST by comparing each edge. This will take a time of O(V). This is done E times. The total time complexity for the loop will be O(EV). Hence, the algorithm will have a time complexity of O(EV).

Let us see if we can improve this running time. The costly operation that we are performing here is to check if e forms cycle with MST, the disjoint set data structure.

Disjoint Set Data Structure

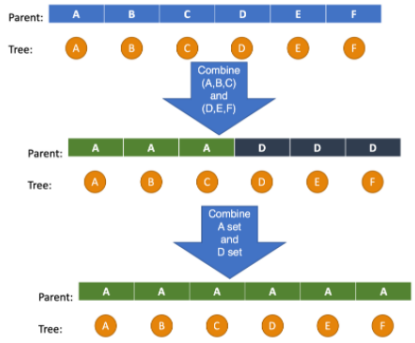
Let us look briefly about disjoint set data structure. Disjoint set data structure stores a collection of a disjoint set of items. In other words, it stores a partition of set of objects in disjoint sets. Among all the operations that it supports two that would be useful to us is to find the set S where an object x belongs to (Find(x)) and to merge two sets and form a single set (Union(x,y))

Find(x) = returns the set to which x belongs to.

Union(x,y) = will merge the two sets x and y.

Suppose, to begin with, we have a graph with vertices (A, B, C, D, E, F).

Using the Disjoint set data structure each object is represented with a parent. When we have a single object in the set each one is its own parent. When we merge two disjoint sets we update the objects to point to a common parent/leader. When we merge vertices A, B, and C, we can choose one of these as the leader, let us say we pick A. Similarly when we merge D, E, F we pick one leader among these vertices, let's say D. We now have two disjoint sets. Given a vertex, we can find which set it belongs in O(1) time by just querying for its leader. The below figure demonstrates the merge and operations on the vertices (A, B, C, D, E, F).



The union operation can be performed in many ways to optimize the implementation. Rather than always appending the second set to the first set, we can always append the set that is smaller to the bigger set. To accomplish this we would have to maintain a rank of each tree, which might refer to the number of objects in the tree.

The union operation can be accomplished in logarithmic time complexity in the worst case. You can refer to this data structure in the supplementary help section on this page.

EFFICIENT IMPLEMENTATION

For ease of implementation, we can write the pseudocode of Kruskal's as below. Where, to begin with, we will create a forest of trees with each vertex having its own tree. Then we greedily add the minimum weighted edge e that connects from u - v, such that u and v belong to different trees so that there are no cycles.

```
def Kruskal(V,E):
    sort E by increasing weight

    for v in V:
        create a tree for each V

    MST = {}

    for i in Range(|E|):
        (u,v) <- lightest edge in E

        if u and v not in same tree:
            MST.add((u,v))
            merge u and v trees

    return MST
```

Firstly, we create a forest for each tree.

```
for v in V:
    create a tree for each V
```

Then we greedily add the minimum weighted edge e, whose vertices do not belong to the same tree. This is to make sure that no cycles are created.

```
(u,v) <- lightest edge in E

if u and v not in same tree:
    MST.add((u,v))
```

Then, we merge the two trees.

The operations: to verify 'u and v do not belong to the same tree' and 'merging u's tree with v's tree' can be accomplished by using sets.

We can use a disjoint set data structure to efficiently implement the algorithm.

The pseudocode can be written as:

```
def Kruskal(V,E):
    E_sorted = sort E by increasing weight

    for v in V:
        make_set(v)
    MST = {}

    for (u,v) in E_sorted:
        if(Find_set(u) != Find_set(v)):
            MST.add((u,v))
            Union(u,v)
    return MST
```

make_set(v) — Create a set containing only the vertex v in O(1) time. This would be performed a total of V times.

Find_set(v) — Return an set containing v in O(1). This would be performed 2E times, twice for each edge.

Union(u,v) - performs |V|- 1 operations (one for each edge in the minimum spanning tree) in amortized O(logV) time. This is performed a total of O(VlogV) times.

The sort operations dominate the running, which would be O(ElogE) to sort the edges. In graphs we have edges |E| at most of |V|2 (That will happen when all the vertices are connected with each other), So we can write the time complex as O(ElogV).

Exploration: P, NP, NP-Complete, NP-Hard

Introduction

Having come to almost the end of this class, now you know numerous problems of different time complexities ranging from a linear search that takes O(n) to an N-Queen problem that takes O(n!) time. When solving a problem, we always want to ask can we do better to improve the time complexity? Now, let us see the world of problems on which still research is going on and scientists are trying to find a better solution. In a few cases, they are working to prove that they cannot find a better solution. We will explore all those in this and the next sections.

COMPLEXITY CLASSES

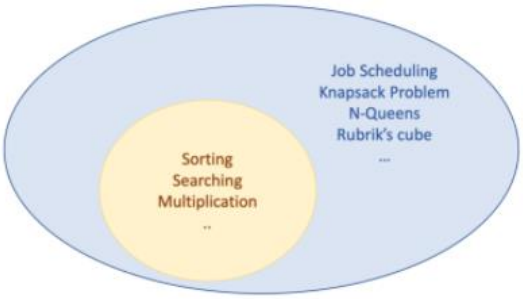
When we say that we are trying to make a problem faster do we have a definition of what is most efficient? Until merge sort and quick sort algorithms were invented, we thought n^2 is the maximum efficiency that we can achieve for sorting problems. There are a few problems like 0/1 knapsack problems that take 2^n time. If you remember from the first module, we said anything that takes 2^n or more is extremely slow. To refresh your memory, let us compare n^2 with 2^n. A problem with n = 100, when solved with n^2 time complexity algorithm would take less than a second, but the same would

take 10^17 years to solve using 2^n time complexity algorithm. With this, we can conclude that "polynomial" execution time is certainly a lot efficient than "non-polynomial" execution time.

For ease of understanding and analyzing problems, computer scientists have classified problems based on how much time is required to solve the problems, this classification is called complexity classes. We shall not cover this in detail but look at it to the extent needed for our topics of discussion.

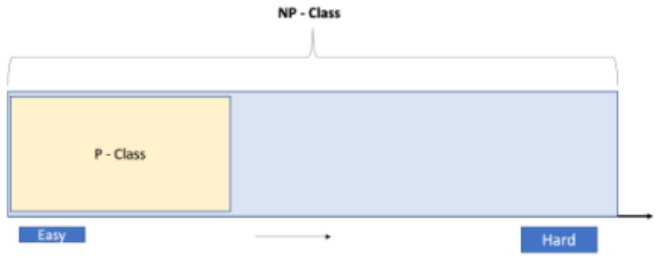
To begin with, if we look at the problems in the world that can be solved in non-polynomial time, for now, ignore the fact that we can solve a few of them in more efficient polynomial time. We can group all these problems and add them to a class, let's say NP class. Say, we have not found an efficient polynomial-time algorithm for any problem yet. The problems that we know – searching problems, sorting problems, knapsack problem etc. belong to this NP class.

Among those problems for which we were able to find a polynomial-time solution to can be grouped together and added to a subclass, let us call it P class.



This is a broad general picture of P class and NP class.

Let's say we were to arrange these problems on a scale that increases in terms of the time complexity: Easy refers to those that are solvable by efficient polynomial-time algorithms, these are placed on the left of the scale and hard refers to those that are difficult, and we don't know if there is a polynomial-time algorithm to solve these. Our graph would look as shown below. Keep this graph in mind we will add more items it as we go.



REDUCTION

Take an example game called number game, that you are playing with your friend. You are given numbers 1 to 9. Each player has to select a number on their turn. When all the numbers picked by a player adds up to 15 then that player wins the game.

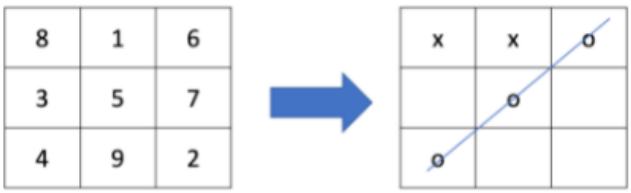
1	2	3	4	5	6	7	8	9

Say, you picked 5, you friend picks 8;

Then you pick 5 and your friend pricks 1;

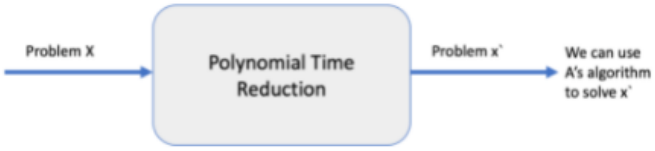
Then you pick 6 and you win the game 😊.

This game is little difficult to play as is, but let us say we arrange the numbers in the form of grid such that in the row, column and diagonally the numbers add up to 15, as shown below. Then the game is easy to play. To win you just have to select a number in a row/column/diagonal and stop your friend from doing the same.



So, if we have an algorithm to solve the tic-tac-toe game, we can easily solve the number game. This is an abstract idea of converting one problem to another to solve it easily. We call it a reduction. The process of reducing one problem to another should be in polynomial time, otherwise, there is no point in reduction. Suppose we have an algorithm for some problem A. We want to solve a problem X. Let's assume X is reducible to problem A in polynomial time. Then we will first perform the polynomial-time reduction to get problem X and solve X using A's algorithm. This is demonstrated in the below figure. This can be written as X <=p A (X can be reduced to A in polynomial time). In other words, X is no harder

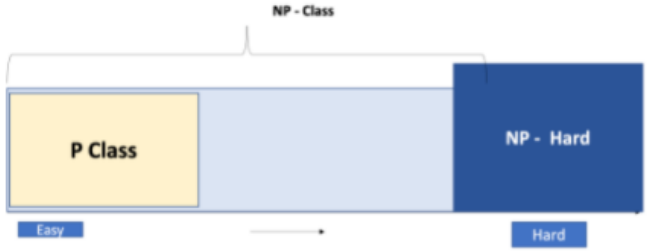
than A. Note that reduction does not mean reducing a harder problem to an easier one, but in our context, it means transforming one problem to another.



NP-HARD and NP-COMPLETE

Suppose you have a problem 'myProblem', that you are trying to solve in polynomial time, but you were not able to. After struggling for a week you are reminded of the concept of reduction. You decide to reduce one of the NP problems in NP to 'myProblem'. After thinking hard, you pick some problem NPx from the set of NP-Class (not in P-Class) and reduce NPx to 'myProblem' in polynomial time. Since you were able to do this, you can declare that your problem is not solvable in polynomial time but it belongs to NP-class.

For this purpose, can you choose any problem from the NP-class? Well, no. You should pick from 'the' most difficult problems from NP class. But what are those? Scientists have picked such problems from NP-class such that other problems in NP can be reduced to these and called this class of problems as NP-Hard. These problems are the hardest problems in NP-Class.



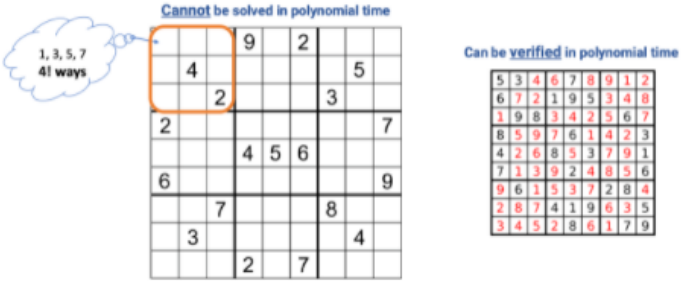
Now coming to 'myProblem' if you are able to reduce an NP-Hard problem to 'myProblem', with full confidence you can say that your problem is not solvable in polynomial time as you have proved that it is similar to one of the hardest problems in the Complexity Class. Now, it is the work of scientists to solve your problem, or you become a scientist and solve it.

The NP-Hard class extends beyond the NP-Class. To understand why? Let us define our NP-Class.

NP does not mean Non-polynomial, it means Non-Deterministic Polynomial. To simply put it, as we discussed before it includes those problems that cannot be solved in polynomial time, but there is an additional property of these, these can be **verified in polynomial time**.

To understand this let us take the Sudoku game. Rules of the game are that we fill numbers from 1 to 9 in the grids such that no number repeats itself in a row and a column, additionally, no number should repeat in each grid of 9 squares marked by thick lines. To solve this game, let us first look at the first

grid, which already has numbers {4,2}. So the first cell can contain numbers {1,3,5,7} excluding 6, because it is already in the column. This can be done in 4! ways, similarly, we can check the next cell. The number of possibilities grows the time complexity of the solution to a non-polynomial degree. But if we are given a filled Sudoku puzzle, it can be verified in N2 time (polynomial time). Sudoku belongs to NP-Class but not to P-Class. You might think, well computers can solve Sudoku quickly why is this in NP? At this point, we should not forget that we are speaking about asymptotic growth of the problem as the input size increases i.e. the size of the Sudoku puzzle increases.

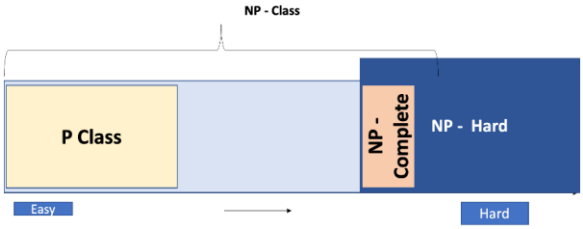


Why the word Non-deterministic? 'In computer science, a nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm'. This means we have a magic machine that somehow can give us the right answer for each input. If we had such a magic machine, we give it each cell of Sudoku and it will give us the right answer in a constant time, then we can solve our Sudoku in polynomial time. 😊 Hence the name Non-deterministic polynomial.

NP-Hard problems **include** those that cannot be verified in polynomial time. But a special point to note about the NP-hard problems is that every problem in NP can be reduced to these problems.

Now, there can be some problems in NP-Hard that really are hard among the NP-class, that is every problem in NP can be reduced to these problems but these can be verified in polynomial time. These form the set of overlap between NP-Class and NP-Hard, these are called NP-Complete problems. A problem is NP-Complete if;

- It is in NP-Hard (i.e. every problem in NP can be reduced to it in polynomial time)
- It is NP (i.e. it can be verified in polymodal time but cannot be solved in polynomial time)



P =? NP

We will refer to P-class as P; and NP-Class as NP.

Is it true that we cannot solve any of the problems in NP in polynomial time? Maybe or maybe not! It is not proved that it is not possible. In fact, there is a million-dollar prize on this question: Is P = NP? Clay Mathematics Institute listed 'Millennium problems' for million dollar price for solving each one of them, P vs NP is one among them.

If we are able to solve one NP-Hard problem in polynomial time then all the problems in NP can be solved in polynomial time because all the problems in NP can be reduced to NP-Hard problems. This would collapse NP into P. We will leave the P vs NP question with this :).

For now, remember that if we are able to solve one NP-Hard problem in polynomial time then all the problems in NP can be solved in polynomial time. This would mean P=NP, which is not true (as of now). We will use this statement to prove NP-Hard problems in the next section.

Decision Problems

Decision problems are those which can be answered in yes or no.

Examples:

- Is the given list of integers sorted?
- Given a graph G, is there a path connecting vertices U and V?

Other classification problems can be an optimization problem, where we are finding the best/optimum possible solution.

Examples:

- Given a list of numbers, what is the maximum number?
- Given a graph G what is the minimum distance between vertices U and V?

Each optimization problem can also be converted into a decision problem.

Example:

- Optimization problem: Given a knapsack of capacity W and N[] items worth value V[], what is the optimum way to fill the knapsack with these N items to maximize the value of the knapsack.
- Decision problem: Given a knapsack of capacity W and N[] items worth value V[]. Does a solution exist to fill the knapsack such that total worth is at least k.

Why are we discussing decision problems? Historically for mathematical simplicity the discussion of P and NP is done in terms of decision problems. Here are more accurate definitions of P and NP.

P Class: P is a complexity class that represents the set of all decision problems that can be solved in polynomial time. That means any problem that is in P can produce a YES or NO answer in polynomial time.

NP Class: NP is a complexity class that represents the set of all decision problems for which the instances where the answer is "yes" have proofs that can be verified in polynomial time. So, to reword this, if we were given a solution to a problem and the answer sheet for that specific problem, we can confirm that the solution to the problem is in fact correct in polynomial time.

In the examples above we have shown that we can convert any optimization problem or any problem in general into a decision problem. And since the main difference between P and NP is whether or not a solution exists for the problem that can be found in polynomial time, it is important to phrase problems as decision problems, so that the appropriate comparison/analysis can be made.

Exploration: NP-Completeness

Introduction

In this section, we will look at few NP-Complete and NP-Hard problems. We will also see how to show a problem to be NP-Complete.

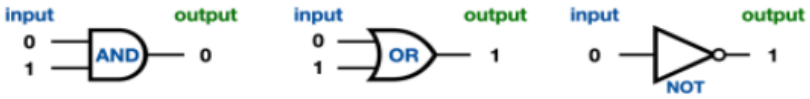
CIRCUIT SAT PROBLEM

Let us look at a popular problem called Circuit SAT.

Problem 1: Circuit SAT

Basics of a circuit: Circuit contains few elements that takes either 0/1 and outputs 0/1 depending on type of circuit.

Few basic elements that we are interested in:



Similar to AND operation

input		output
0	0	0
0	1	0
1	0	0
1	1	1

Similar to OR operation

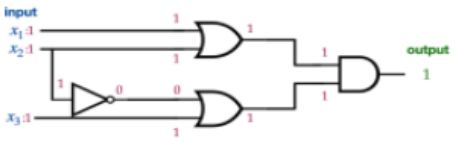
input		output
0	0	0
0	1	1
1	0	1
1	1	1

Flips input

input	output
0	1
1	0

Problem 1: Circuit SAT

A circuit: A circuit is a combination of these elements.



input			output
1	1	1	1

input			output
1	1	0	?

What will be the output of this circuit for the above input?

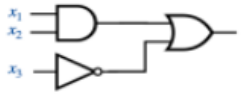
- 1
- Cannot decide
- ✔ 0

1 / 1

Problem 1: Circuit SAT

Problem Definition: Given a Boolean circuit, is there a set of inputs that makes the circuit output TRUE (i.e 1)

If such a assignment exists we say that the circuit is satisfiable (SAT in short).



- ✔ Yes

  Correct! Explanation: one possible input values: x1 = 1, x2 = 1, x3 =0
- No
- Cannot decide

1 / 1

Problem 1: Circuit SAT

Problem Definition: Given a Boolean circuit, is there a set of inputs that makes the circuit output TRUE (i.e 1)

If a circuit has N inputs and we are want to find if the circuit is satisfiable or not, how many possible inputs do we have to try. Consider we are using brute force approach?

- Cannot decide
- N^2
- 2N
- ✔ 2^N

  Correct! Explanation: Each input can be either 0/1, we check this possibility for each input i.e 2*2*2*...N times = 2^N

Problem 1: Circuit SAT

Problem Definition: Given a Boolean circuit, is there a set of inputs that makes the circuit output TRUE (i.e 1)
Let us see if this satisfies our NP-Class definition

Can we **verify** the circuit SAT problem in **polynomial time**? In other words, given a set of input values can we verify if a Boolean circuit is satisfiable in polynomial time?

✔ Yes

Correct! Explanation: To be able to tell if circuit is satisfiable, we just have to run the inputs through the circuit elements and check if the output is 1 or 0.

Cannot decide

Can we **solve** a circuit SAT problem in **polynomial time**? In other words, given a Boolean circuit can we tell there exits a set of inputs that would make the circuit output One in polynomial time?

Cannot decide

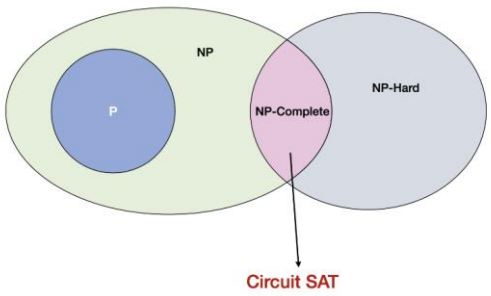
Also, correct, Explanation: Maybe we can, but as of now no one has discovered a clever algorithm to solve Circuit problem in Polynomial time.

✔ No

The Cook-Levin Theorem

Two Computer Scientists, Cook and Levin, independently proved a remarkable theorem that showed the Circuit (Boolean) SAT problem is NP-Hard. Which means any problem in NP can be reduced to this in a polynomial time. We will not go into the details of this theorem or the proof.

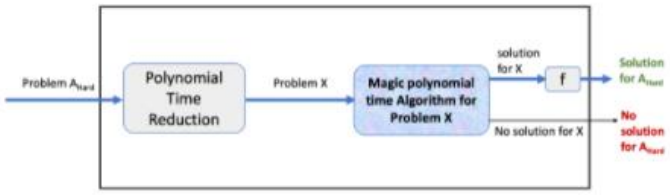
This implies that the Circuit SAT problem is NP-Complete; that is, it is NP-Hard and is in NP-class.



PROVING NP-COMPLETENESS

We have seen before that we can use reduction to transform one problem into another. If we want to show that our problem X is NP-Hard:

- We pick a known NP-Hard problem, say A_Hard
- We will show that we can reduce problem A_Hard to problem X. There are two parts to reduction:
    - To show that we can transform problem A_Hard to problem X in polynomial time.
    - To show that if there was some magic algorithm to solve problem X, then we can obtain the solution for the problem A_Hard using some polynomial transformation(f) if needed. If the algorithm returns that there is no solution for X, then we can conclude that there is no solution for A_Hard.



Now, we can argue that if there exists a polynomial-time algorithm to solve problem X, then problem A_Hard could be solved in polynomial time. If the problem A_Hard could be solved in polynomial time then P=NP. This implies problem X is NP-Hard.

The magic algorithm box that we saw in our reduction step is called a BlackBox. We don't know how it does the job but we know that it would give us our required output.

3SAT Problem

The below deck explains what a 3SAT problem is and how to show it to be NP-Complete.

Basic Concepts: Boolean Formula

You might be aware of a couple of Boolean operators: Negation, Conjunction (AND, ∧), Disjunction (OR, ∨), equivalence (⇔)
If we have variables (or literals) a, b, c, d…. Then some Boolean formulas could be:

- (a ∧ d) ∧ (b ∧ c)
- (b ∧ $\bar{c}$)
- (a ∧ b ∨ c) ∧ (c ⇔ a ∧ d) ∨ ($\overline{a \wedge d}$)

Note: We don't have to worry about processing or solving these formulas we are interested only in AND, OR, and Negation :)

Conjunctive Normal Form (CNF): It is a type of Boolean formula that contains only Conjunction (∧) and Disjunction (∨) of literals.

The literals could be variable (x) or the negation of the variable $\bar{x}$.
Examples:

- (a ∨ d) ∧ (b ∨ c) ∧ ($\overline{a \vee d}$)
- (a ∨ b ∨ c) ∧ (b ∨ c)

Basic Concepts: Conjunctive Normal Form (CNF)

Conjunctive Normal Form (CNF): Each of the clauses contains Disjunction (∨) and all the clauses are joined by Conjunction (∧) hence the name conjunctive form.

2CNF: Clauses have 2 literals

$(a \vee b) \wedge (b \vee c)$

3CNF: Clauses have 3 literals

$(a \vee b \vee c) \wedge (b \vee c \vee \bar{c})$

Problem: 3CNF or 3SAT

Definition: Given a 3 CNF Boolean formula, is the formula satisfiable?

Example:

Is $(a \vee b \vee c) \wedge (b \vee c \vee \bar{c}) \wedge (d \vee e \vee \bar{c})$ satisfiable? i.e. does there exist a value of the literals that will make the resultant value of the formula true?

This problem is also called **3SAT.**

Show that 3SAT NP-Hard

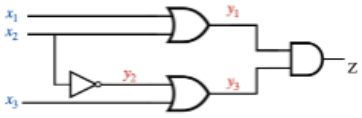
To show that 3SAT is NP-Hard:

- STEP 1: We will pick a known NP-Hard problem: we know Circuit SAT is NP-Hard
- STEP 2: We will reduce Circuit SAT to 3SAT in polynomial time

Problem: 3CNF or 3SAT

Reduce Circuit SAT to 3SAT in polynomial time

Let K be an arbitrary Boolean circuit.



We will transform this into 3CNF form as follows:

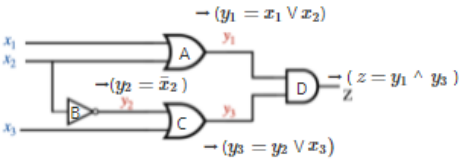
1. Firstly we will form a Boolean formula ϕ. For this we will label every interior wire with a label y_i and the output with a variable z.
2. Then, we will convert this Boolean formula into 3CNF from.

Problem: 3CNF or 3SAT

Reduce Circuit SAT to 3SAT in polynomial time

Reduce Circuit SAT to 3SAT in polynomial time

Form a Boolean formula ϕ: Write an expression for each of the circuit element and the whole circuit can be converted into conjunction of each element.



ϕ: A ^ B ^ C ^ D ^ Z

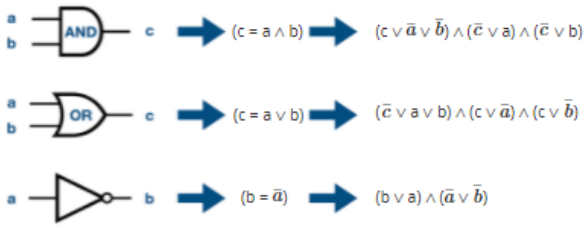
$\phi: (y_1 = x_1 \vee x_2) \wedge (y_2 = \bar{x}_2) \wedge (y_3 = y_2 \vee x_3) \wedge (z = y_1 \wedge y_3) \wedge z$

A B C D

Problem: 3CNF or 3SAT

Reduce Circuit SAT to 3SAT in polynomial time

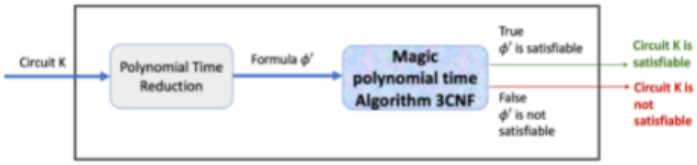
Let us see how to convert each circuit element into 3CNF form.



Problem: 3CNF or 3SAT

Reduce Circuit SAT to 3SAT in polynomial time

Using the equations shown previously, we can covert a circuit K that has n symbols into a 3CNF form ϕ' in O(n). The circuit is satisfiable if and only if the 3CNF form ϕ' is satisfiable.
Now suppose there exists a magic algorithm that can solve ϕ' in polynomial time. Then we can say that Circuit SAT can be solved in polynomial time.



The magic algorithm concludes that Circuit SAT can be solved in polynomial time which is a contradiction since the circuit SAT is a known NP-Hard problem. Hence, we can conclude that **3SAT** is **NP-Hard.**

Problem: 3CNF or 3SAT

Show that 3SAT NP-Complete.

To show 3SAT is NP-Complete, we need to show:

Step 1: 3SAT is NP-Hard. We have already shown this.

Step 2: 3SAT is verifiable in polynomial time. Given a 3SAT formula and the input variable values we can verify if the formula is satisfiable in polynomial time.

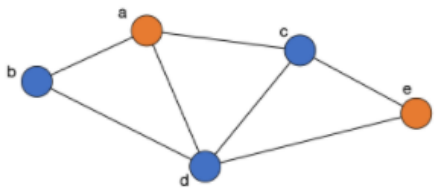
Hence, we can conclude that 3SAT is in **NP-Complete.**

INPENDENT SET PROBLEM

The below deck explains what is a Independent Set problem and how to show it to be NP-Complete

Basics: Independent Set

Given an unweighted and undirected graph G, the set of vertices S form a independent set if there is no edge between the vertices of S.



Graph G

S = {a, e} is an independent set in graph G

Problem: Independent Set

Definition: Given a graph G and number n. Does G contains an independent set of size at least n?

Show that Independent Set Problem is NP-Complete

To show that the problem is in NP:

If we have a Graph G, a set of vertices S and a number n. We can check for each edge in G does not have their end points S and verify that that S is independent set. This verification can be done in polynomial time.

Independent set is in NP-Class.

Problem: Independent Set

To show that the problem is NP-Hard:

- Pick a known NP-Hard problem. Let us pick 3SAT. You will see the connection between these problems in a while.

- Reduce 3SAT to Independent Set:

  As you have seen reduction has two parts:

  1. To show that 3SAT can be converted to Independent Set problem in polynomial time
  2. From Independent Set problem solution we can get the solution for the 3SAT problem (the transformation that is done after the magic box).

Problem: Independent Set

Reduction - General Idea to convert 3SAT to a graph: Let us take a 3SAT formula.

$(a \vee b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (\bar{a} \vee b \vee \bar{c}) \ldots$.

If this formula is satisfiable then there is at least one literal (or it's complement) that has a value TRUE which is making the clause TRUE.

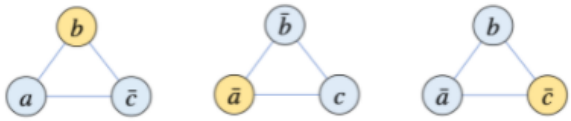
Let us pick one literal from each clause that is contributing to the satisfiability of this formula.

$(a \vee b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (\bar{a} \vee b \vee \bar{c})$

Problem: Independent Set

Reduction - General Idea to convert 3SAT to a graph:
We can convert each clause into a triangle (a mini graph).

$(a \vee b \vee \bar{c}) \wedge (\bar{a} \vee \bar{b} \vee c) \wedge (\bar{a} \vee b \vee \bar{c})$



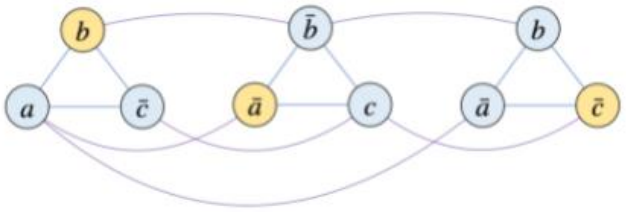
If we chose a literal from a clause, we cannot chose it's negation from other clause. Why? Because we are choosing only those literals which are TRUE, both the literal and it's negation cannot be TRUE at the same time.

Problem: Independent Set

Reduction - General Idea to convert 3SAT to a graph:

To force a condition to chose only either a literal or it's negation we can draw a connected edge between the literals and their negations between different triangles.

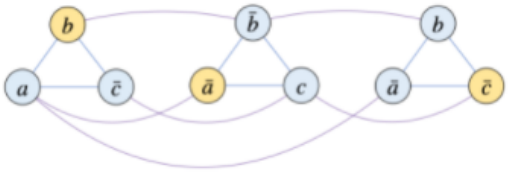
(a ∨ b ∨ $\bar{c}$) ∧ ($\bar{a}$ ∨ $\bar{b}$ ∨ c) ∧ ($\bar{a}$ ∨ b ∨ $\bar{c}$)



Problem: Independent Set

Reduction - General Idea to convert 3SAT to a graph:

(a ∨ b ∨ $\bar{c}$) ∧ ($\bar{a}$ ∨ $\bar{b}$ ∨ c) ∧ ($\bar{a}$ ∨ b ∨ $\bar{c}$)



Now, our graph is built in such a way that if we pick a TRUE literal that satisfies the 3SAT formula, we would have also picked a set of vertices that form independent set as they are not connected.

Here we need to construct n clauses to find a n sized independent set of a graph.

We have shown that we can transform a 3SAT problem into a independent set problem.

Problem: Independent Set

Let us recap our graph construction steps:

To find if a graph G has independent set of size k

1. We take a k clause 3CNF formula.
2. We convert the 3CNF formula into a graph such that each clause forms a triangle and we add edges connecting x and $\bar{x}$

REDUCTION STEPS:

Step 1: To show that 3SAT can be converted to Independent Set problem in polynomial time

> Converting n clauses into n triangles can be done in linear time. This would cover 3n edges of the graph. Additionally we need to add edges between literals and their complements. This can be done in polynomial time.

Problem: Independent Set

REDUCTION STEPS:

Step 1: To show that 3SAT can be converted to Independent Set problem in polynomial time

Step 2: From Independent Set problem solution we can get the solution for the 3SAT problem

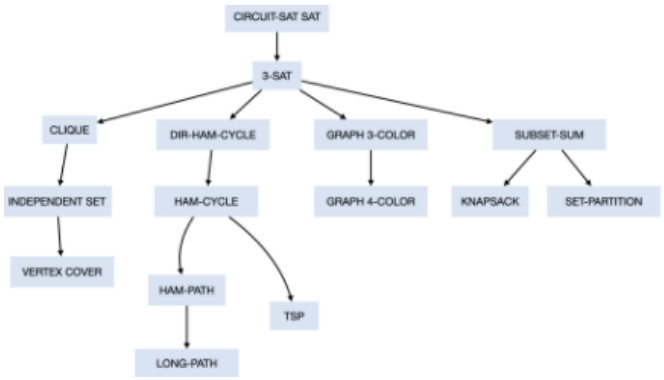
> If a graph G has k sized independent set S, then each vertex in set S must lie in a different clause triangle. As the contradictory literals are connected by edges, this is in consistent with the independent set solution and both will not be selected. Since S contains only one vertex from each triangle which can be represented as TRUE, this means that our 3SAT problem with k clauses is satisfiable.

We can conclude that, the Independent Set problem **is NP-Hard**

Previously we have shown that Independent Set problem is in NP-class.

Hence, it is a NP-Complete problem.

'NP-Complete proof template' provided in the optional reading material could be handy resource when you want to prove for NP completeness. The below graph shows a list of NP-Complete that are provided with the help of other known NP-Hard problems. You have seen a few of the problems like the Knapsack problem.



Exploration: Approximation Algorithms to Solve NP-Hard Problems

Introduction

We have seen the hard problems out in the world that cannot be solved in polynomial time. But what do we do about them? There are two options: 1. Solve small input sizes of these problems by following exhaustive search, but it might not always work for us as the exhaustive search time complexity grows very fast. 2. Solve these problems and obtain an approximate solution.

In this section, we will look into the second option. We will explore some strategies that are followed to obtain approximate solutions for NP-Hard problems.

Approximation Algorithms

While we know that algorithms are step by step procedures to solve a problem, a **heuristic** is a simple strategy that we build mentally to form an approximate solution to a problem, it is not mathematical approach. We have already used one heuristic technique before. Can you think which one? Yes, the Greedy strategy. Using a greedy strategy we mentally run through possible ways of solving a problem and then pick one that appears to give us the near-optimal solution. Those algorithms that produce near-optimal solutions are called approximation algorithms.

Approximation Ratio

C : The solution produced by the algorithm

C* : The optimal solution

ρ(n) is the approximation ratio for an approximation algorithm of problem size n.

$$\rho(n) = \max\left(\frac{C}{C^*}, \frac{C^*}{C}\right)$$

If an algorithm has ρ(n) approximation ration then it is called ρ(n)-approximation algorithm.

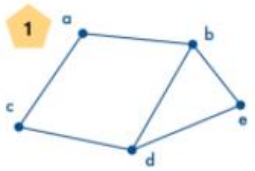
To maintain a uniform scale we either perform C/C* or C*/C , to make the ratio greater than 1. If the scale is equal to 1 we can say that the algorithm produces the optimal solution.

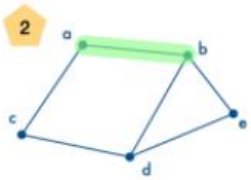
For example, if we are computing a minimum optimal solution and if ρ(n) = 1.5, then we can say that the approximation algorithm has computed a solution that is at most 50% larger than the optimal solution.

VERTEX COVER PROBLEM

Problem: Given a graph, find the minimum set of vertices that cover all the edges of the graph.

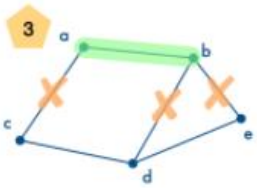
One possible approach to solve this problem will be to pick a random edge E that has vertices (u, v). Add these vertices to the solution set. Remove all edges that are connected to either u, v.



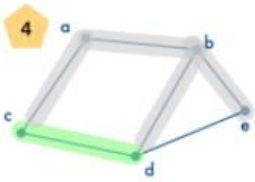


Solution: {a, b}

Pick a random edge E that has vertices (a,b). Add these vertices to the solution set.

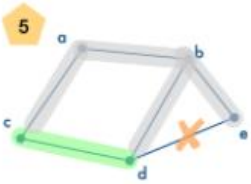


Remove all edges that are connected to either a, b

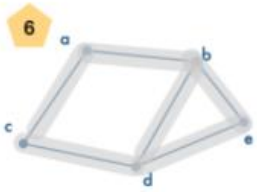


Solution: {a, b, c, d}

Pick a random edge E from remaining edges and add these vertices to the solution set.



Remove all edges that are connected to either c or d.



Solution: {a, b, c, d}

We have covered all edges.

```
approx-vertex-cover(G):
    solution = {}
    Edges = All edges in G
    while(E is not empty):
        pick arbitrary edge E whose vertices are (u,v)
        solution = union(solution, {u,v})
        remove all edges that are connected to either u or v

    return solution
```

Time complexity: The while loop will cover all edges E in O(E) time and we add each vertex in the worst case when we perform a union, this will make it the additional time of O(V). The total time complexity will be O(V+E). Which is a polynomial time.

We will see that this solution has an approximation ratio of 2, which means that it guarantees a solution that is no more than twice the size of the optimal solution.

In the algorithm, we are selecting a pair of vertices (u,v) that are not connected to any of the selected vertices. Let k denote the set of edges arbitrarily picked in the while loop.

What is the relation between k and the optimal solution for this problem C* ? Any vertex cover (C* in particular) must include at least one endpoint of each edge in k. This is because no two edges in k share a vertex since we are removing all edges that are connected to picked edge (u,v). Thus, no two edges in k are covered by the same vertex from C*; and hence k give us a lower bound on C*.

$|C*| \geq k$

Each execution of picking an arbitrary edge (u,v) for which neither of the end points are already in C, gives us an upper bound on the size of vertex cover returned. We can say that the solution constructed by the approximation algorithm will have 2k vertices. This is because these k edges have no adjacent edges as we are removing the edges that are connected to these edges.

```
remove all edges that are connected to either u or v
```

$|C| = 2k$

$|C| \leq 2|C*|$ (from the equation $|C*| \geq k$)

$C/C* \leq 2$

This gives us an approximation ratio of 2.

Traveling Salesman Problem (TSP)

The Traveling Salesman Problem is one of the most well-known NP-Complete problems. It has been intriguing scientists for more than a century. In simple language, the problem is: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?".

The below deck covers related terminology that we will use further and the TSP problem explanation pictorially.

Greedy Approach

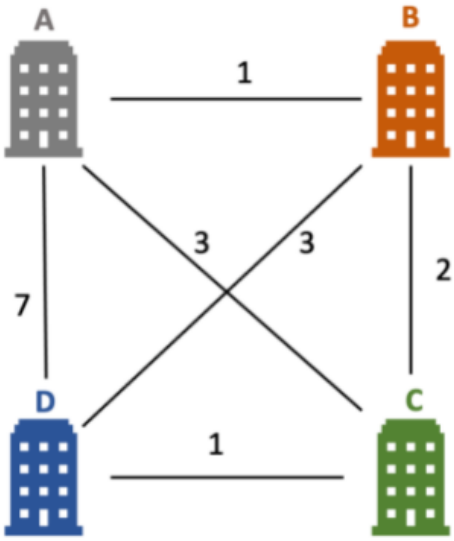
We use a greedy approach to find the near-optimal solution for the TSP problem.

The simplest approach to solve the TSP problem is by employing the greedy technique. We start at an arbitrary vertex and we ask what is the closest vertex that we have not covered so far and travel towards it. This heuristic is called 'closest point heuristic'.

The steps we will cover:

1. Pick an arbitrary starting point
2. Pick next closest un-visited vertex
3. Repeat until all the vertices are visited and return to the starting point.

Let us apply these steps on the below map.



We get the path: A->B-> C->D->A. This give us the cost of (1 + 2+ 1+ 7) = 11.

The optimal path is A->B->D->C->A. This path has a cost of (1 + 3+ 1+ 3) = 8.

The approximation ratio for these values:

$$\rho(n) = \frac{C}{C^*} = \frac{11}{8} = 1.4\,(approx)$$

This is a sample approximate value and this value can be large depending upon the value cost of (D, A) in this example.

Minimum Spanning Tree-based approach

Do you notice any similarity between the Hamiltonian circuit and the TSP problem? Basically, TSP is just asking for the minimum cost Hamiltonian circuit for a graph. We have seen the Minimum Spanning Tree (MST) in the previous modules. MST gives us the lower bound for the TSP problem. There is a close connection between MST and Hamiltonian Circuit problem. Removing an edge from the minimum-cost Hamiltonian circuit gives us an MST. There is an approach to solve TSP problem using these known techniques and using the triangle inequality.

1. We will find the MST for the graph, we can use any known MST algorithm for this. Let us use Prim's Algorithm.

2. Then we will use MST to create a walk using the vertices given by MST
3. Next we shall create a Hamiltonian cycle based on the path

This deck demonstrates the steps.

Note that this algorithm works for a fully connected graph.