RED RIDING HOOD

Anirudh Kaushik
2020111015

Abstract

A 2d platformer game made using algorithms, a comic strip and a concept map

Anirudh Kaushik
2020111015

Introduction

The objective of this project is to demonstrate the use of algorithms in real world applications. To do this I've made a 2d platformer game using algorithms.

The inspiration for this project has been taken from games like Minecraft with a procedurally generated world. A while ago I came across the term "procedural generation" and ever since then I've wanted to create a game using with a new and random world each time.

The game Red riding hood has a new and random world generated each time. We use simplex noise for the generation of this world.

The game has its own sound track, animations for the characters – idle, running, jumping, dying and attacking. The main character has 2 attacks, a melee attack and a ranged attack using a bow. The Minotaurs are the primary antagonist of the game and the objective of the game is to collect as many flowers as possible. Due to time constraints, I have not implemented a starting menu or levels in the game but it is still quite fun to play.

This is a comic strip cum report of the project which in addition includes a small conceptual map of some helpful algorithms covered in class and some additional algorithms I found interesting. I hope you enjoy reading the comic and playing the game.

Except for the main soundtrack of the game, everything in the game has been made by me using free to use sprite animations and various sound effects which have been adjusted to fit the game. This was a very long project and a lot of effort has been put into making it.

Since the game is very large I will be uploading it via a onedrive link.

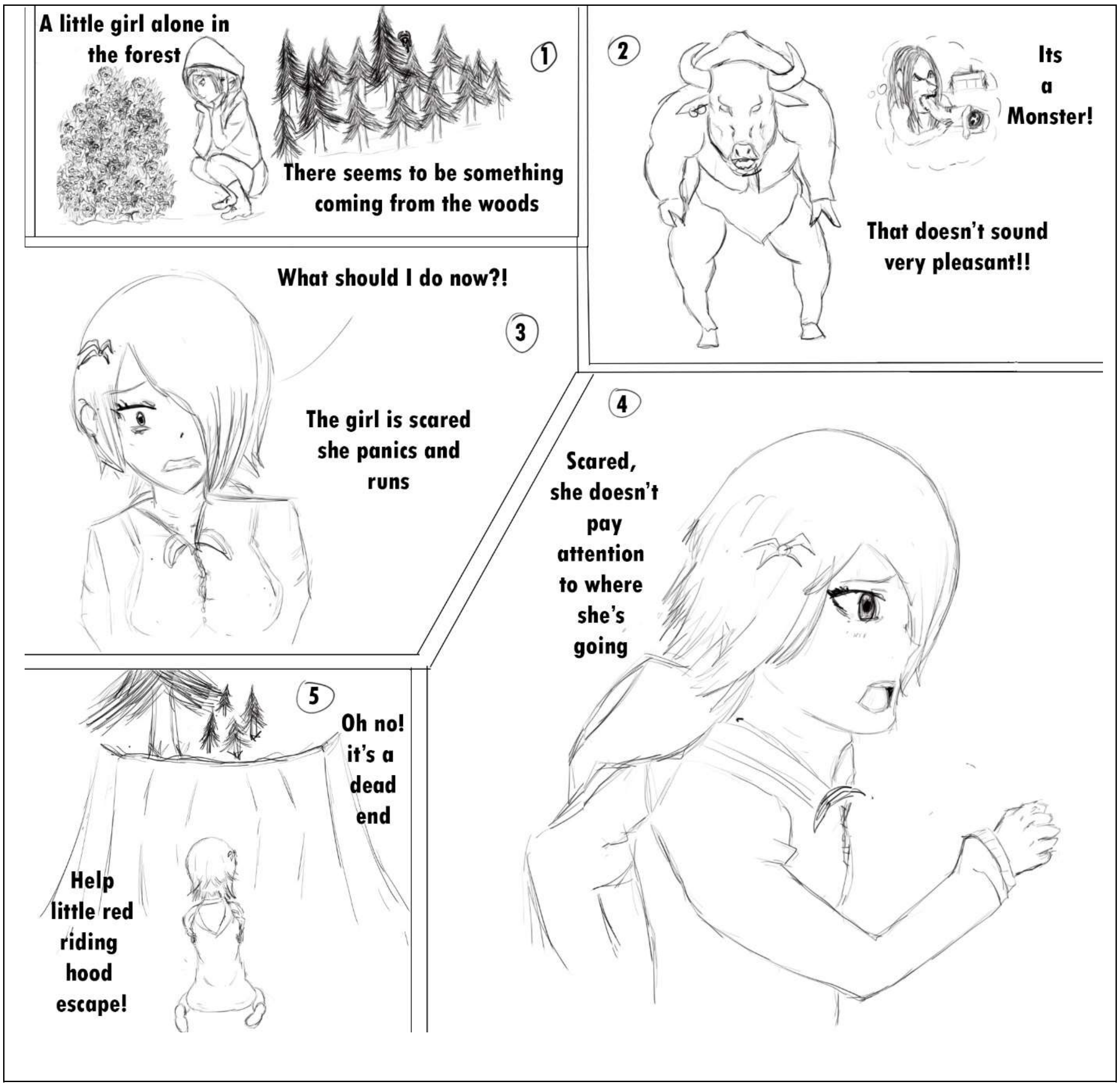
https://1drv.ms/u/s!Aqe0gqqV4W58gfIXhev8Z2fTTmRlhw?e=hEWWeQ

Contents

1. Comic strip discussing problems faced during creation of the game and the corresponding algorithms used and the thought process involved in the construction of the algorithm
2. Here we discuss some useful algorithms and construct a concept map of the topics covered in the course.

Red Riding Hood

Problem discussion illustrated with a comic


A little girl alone in the forest
1
There seems to be something coming from the woods
2
Its a Monster!
That doesn't sound very pleasant!!
What should I do now?!
3
The girl is scared she panics and runs
4
Scared, she doesn't pay attention to where she's going
5
Oh no! it's a dead end
Help little red riding hood escape!

Problem 1

The valid world problem

We are given a 2D matrix of 1s and 0s as an input with around 20 rows and any number of columns. The matrix has a continuous number of exactly one 1 in each column and the rest are all 0s. This 1 can be assumed to be a block and the position of the 1 can be assumed to be the height above the ground. The problem is that the difference between the heights of adjacent 1s may be any number between 1 and 20. However, red riding hood has a jump height of only 1 block and if the height difference between the adjacent columns is greater than 1 than red riding hood will get stuck. Our job is to convert this 2D matrix generated randomly using simplex noise to a valid world by stacking the 1s in every column such that the height difference between any 2 adjacent columns is less than equal to 1. However, there is another constraint. We just spent a lot of time and effort trying to randomly generate terrain of varying heights. We don't want to convert our 2D matrix into a world that is completely flat and boring. Hence, while stacking 1s we must minimize the number of 1s we are adding to the array.

Input: A 2D array of 1s and 0s as described above.

Output: A 2D array satisfying the above constraints such that the number of 1s added is minimized.

Solution Approach

The original problem I faced involved a 2D array with 3 values 0, 60 and 90. 90 was the surface block and its row was the height above the surface. I reduced this to a simpler problem involving 1s and 0s. I decided to convert this 2D array into a 1D array of heights. The ith column in the 1D array was the row index of the ith column in the 2D world matrix. Now, all I had to do was compare the values in the adjacent columns to see if the height difference was greater than 1.

Final solution

The final solution for our problem involved a greedy algorithm. We first convert the 2D array labelled world_data to a 1D array of tuples containing the height and the corresponding index (i,h_i) of every column of world_data. This was stored in heights_list.

We then greedily find the index with the maximum value of h_i. Let's say this is element heights_list[i]. we then compare this with the height stored in heights_list[i-1] and heights_list[i+1] and apply:

heights_list[i-1] = max(heights_list[i-1],heights_list[i]-1);

and,

heights_list[i+1] = max(heights_list[i+1],heights_list[i]-1);

We are able to use a greedy algorithm since this problem possesses both the greedy choice property as well as the optimal substructure property. The greedy choice we make is selecting the element with the greatest value of the height. This insures that the max(heights_list[i-1],heights_list[i]-1) will never increase the height of a column (otherwise we might get an increasing staircase to heaven!).Thus max(heights_list[i-1],heights_list[i]-1) will always give us a value less than or equal to the height of the ith column (if height of i-1 is equal to the height of the ith column).

The optimal substructure property is as follows: after adjusting the height of i-1 and i+1, we remove the index i from the list and again choose the column with the largest height. This means the next height selected will always be less than the previous max height and since the ith column has already been removed, we don't change its height again. In addition max(heights_list[i-1],heights_list[i]-1) always returns 1 less than the max height, so unless the height of the column being changed is equal to the max height we will never stack up extra 1s this preserving the terrain of our world.

Let's say we have an array as follows 1,1,2,3,6,8,3,1. Our algorithm will first select 8. It will check its neighbors indexed 4 and 6.

a[4] = max(a[4],a[5]-1) = max(6,7) = 7

similarly, a[6] = max(a[6],a[5]-1) = max(3,7) = 7

We now remove 8 from the array and our array becomes 1,1,2,3,7,7,1.

We now choose either of the 7s, let is be the one at index 4.

We get a[3] = max(a[3],a[4]-1) = 6

and a[5] = max(a[4],a[4]-1) = 7, thus its value does not change.

Now we remove 7 and our array becomes 1,1,2,6,7,1

In the next iteration the array will become 1,1,2,6,7,6 and we will remove 7 and get 1,1,2,6,6.

In the following iteration we will get 1,1,5,6 followed by 1,1,5 followed by 1,4 followed by 3 and after this our algorithm will simply remove 3 and terminate.

The final list of heights will be 3,4,5,6,7,8,7,6 which satisfies all the given constrains. Hence, we obtain a valid world.

Analysis

For an array of size an we choose the max value n times and hence the algorithm runs in O(N) i.e., a linear pass over the array. Since the algorithm is greedy, we can be sure that it runs in the minimum possible time.

Code

The code is given on the next page. The first few lines are just getting the world data and storing it. Since the top row has index 0 which corresponds to the highest point, we must look for the minimum element each time instead.

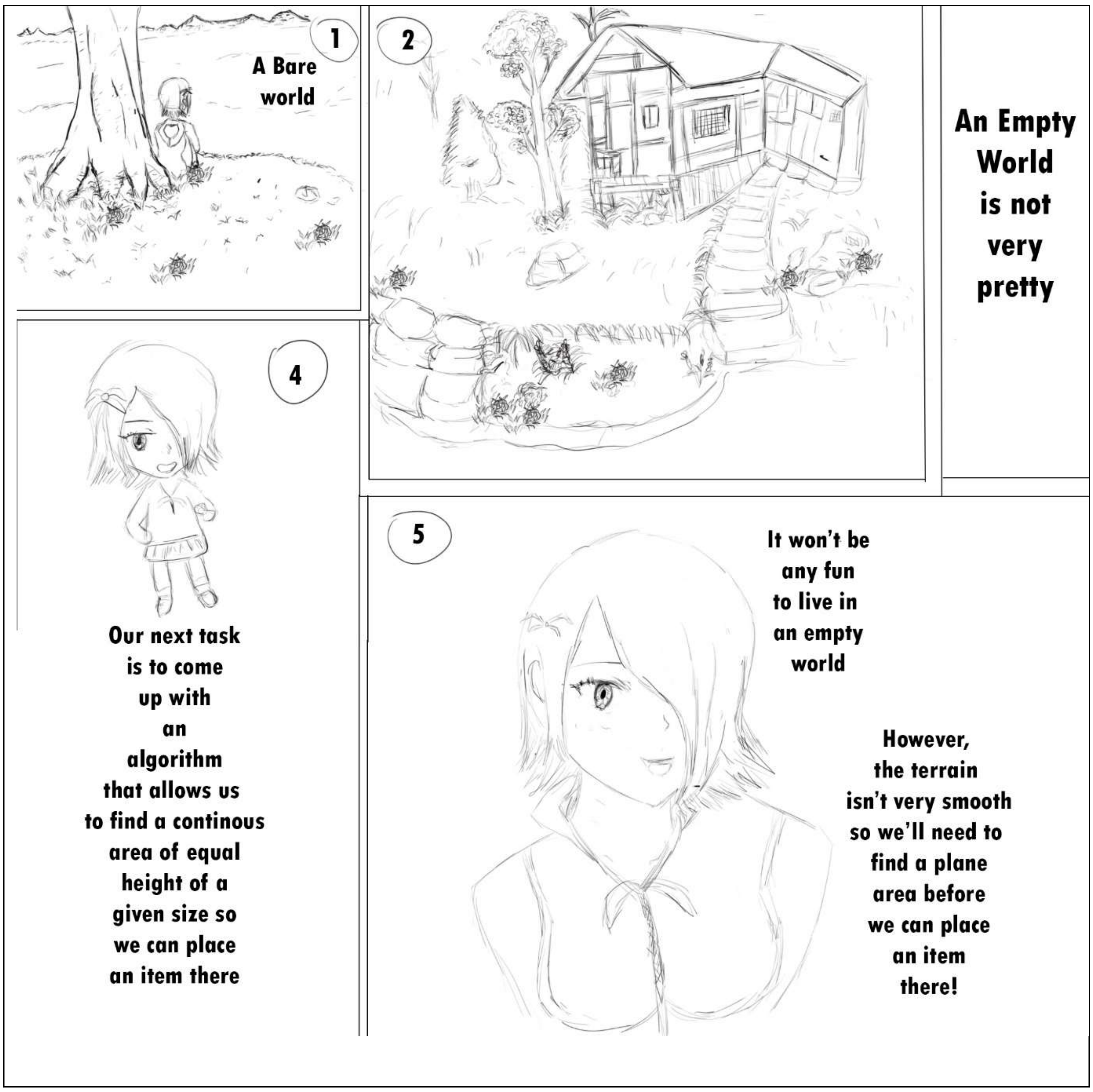
```python
def new_meth():
    world_data = np.array(gen_chal_world())
    p = 0
    height_list = []
    for i in world_data.T:
        height = 0
        for x in i:
            if x == 90:
                break
            if height == 16:
                break
            height += 1

        height_list.append((p, height))
        p += 1

    for i in range(len(height_list)):
        x = min(height_list, key=lambda x: x[1])
        if height_list[height_list.index(x)][1] > 0 and height_list.index(x) >
0:
            a = min(
                height_list[height_list.index(x) - 1][1],
                height_list[height_list.index(x)][1] + 1,
            )
            for k in range(a, height_list[height_list.index(x) - 1][1]):
                world_data[k][height_list[height_list.index(x) - 1][0]] = 90
            height_list[height_list.index(x) - 1] = (
                height_list[height_list.index(x) - 1][0],
                a,
            )

        if (
            height_list[height_list.index(x)][1] < 16
            and height_list.index(x) < len(height_list) - 1
        ):
            a = min(
                height_list[height_list.index(x) + 1][1],
                height_list[height_list.index(x)][1] + 1,
            )
            for k in range(a, height_list[height_list.index(x) + 1][1]):
                world_data[k][height_list[height_list.index(x) + 1][0]] = 90
            height_list[height_list.index(x) + 1] = (
                height_list[height_list.index(x) + 1][0],
                a,
            )

        height_list.remove(x)
    return list(world_data)
```


1
A Bare world
2
An Empty World is not very pretty
4
Our next task is to come up with an algorithm that allows us to find a continous area of equal height of a given size so we can place an item there
5
It won't be any fun to live in an empty world
However, the terrain isn't very smooth so we'll need to find a plane area before we can place an item there!

Problem 2

Populating the world

The next major problem faced during the making of this game was populating our world matrix with items and monsters. My initial approach was to go over the entire matrix and randomly place trees and enemies using a probability function. To avoid slowing down the code I had to use a cool down for placing enemies.
This approach seemed to work well for trees and enemies, however

for items of varying sizes such as carts, wells, boxes, bushes this approach failed since if we try to place an object 3 blocks wide onto a 1 block wide area, two-thirds of the object will be floating in air or will be inserted into a neighboring block.

To solve this issue, we need to place an item only if we have a sufficient amount of flat area available. By flat area we mean a continuous sequence of equal heights in the height array described in problem 1, for example we may choose to place a well with a scarecrow if we find 5 blocks with equal height. Lets say our array has 1777775, we see that the sequence 77777 satisfies our requirements and hence we need a method of storing this index.

This problem can be stated more formally as follows. Given an input n and an array containing values between 1 and 20, find the starting index of all sequences with equal values of size n in the array and output their starting indices.

This problem can be modified to obtain a sequence within a range. We can also provide multiple inputs n and hence we may need to store all the starting indices with different sizes.

Solution approach

I first reduced my problem from the 2D matrix to the simpler form involving a 1D height array. Then I decided to go over the array and store the first value I find, I then check the values at the successive indices and if the value is the same I must increment the variable n denoting the size of the continuous flat land and in case the value I encounter is different from the stored value, I check n and store the pair (n,index) in an array and start from the element that was different and repeat the same procedure.

In this approach we repeat the same procedure of storing the values and incrementing till we encounter a different value and when this happens, we store the start index of the block and the size of the block and continue from the next element which had a different value.

Final solution

I use a python deque to store the values. In the beginning, I enter a for loop starting at i = 0 till i is less than the size of the heights array and then push the ith index value onto the deque and initialize n to 1. I then enter another for loop starting at the index j = i+1 till j < size of heights array. If the element has the same value as the stack top, I push it on to the stack and increment the value of n, if the element has a different value, I store n and i in a list. I then make I equal to the j-1 (one before where we encountered the first different value) and clear the queue and break the inner nested for loop. The outer for loop increments i to j (the index of the first different element we encountered) and we continue this process till we reach the end of input.

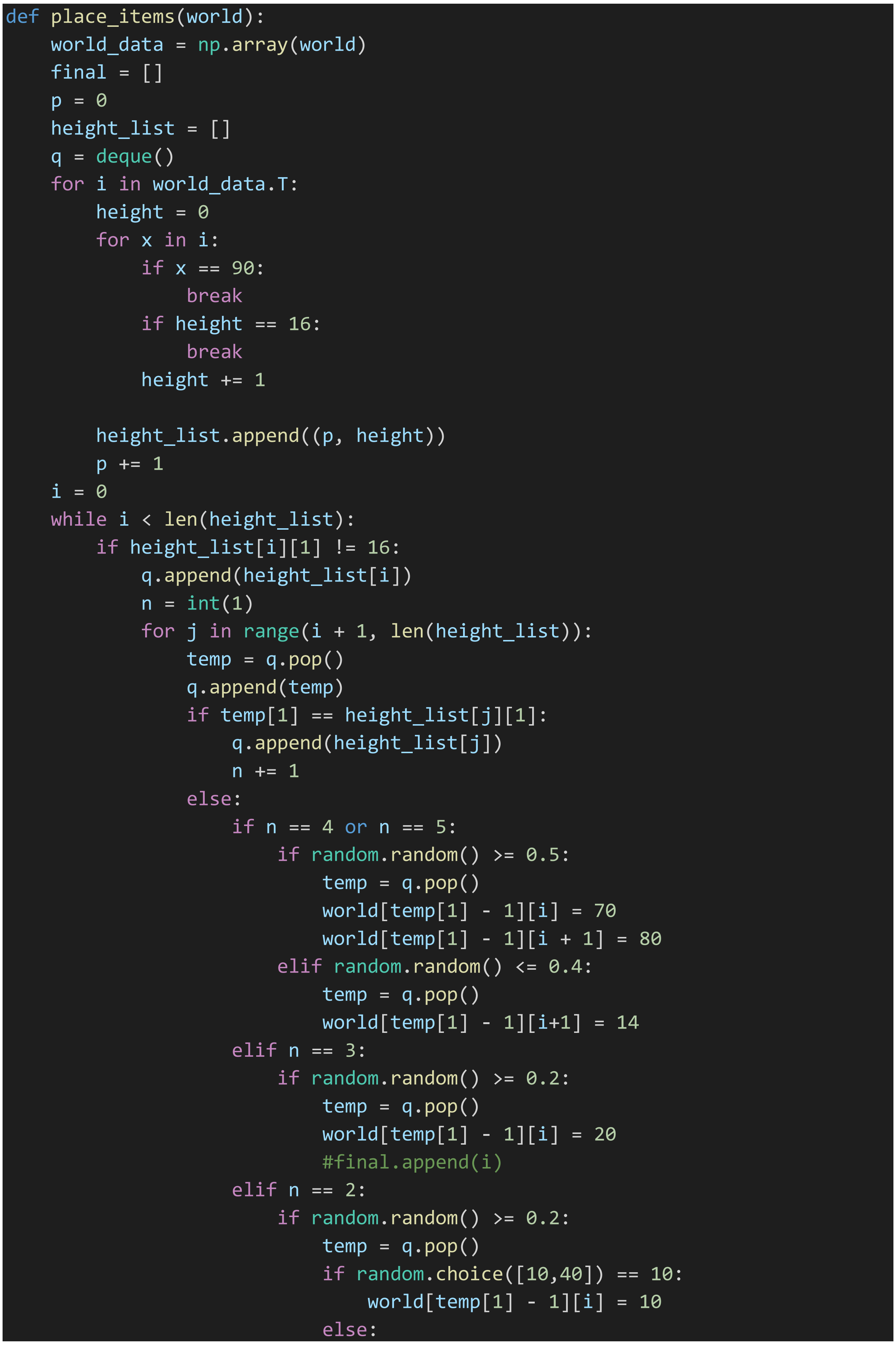
Analysis

There are two nested for loops. However, we make only one linear pass over the array since the inner for loop starts just after the outer for loop and the outer for loop picks up right where the inner for loop ended. We could have done this with a single for loop but for the sake of explanation two for loops were used. Since we make a single pass over an array of size n its complexity is O(N).

Code

The code involves a deque q and the heights array is as described in problem 1. The first part of the code simply initializes the heights array from the 2D matrix.

Since we can make changes to the world matrix along the way, we do not store the indices and values of n and instead we just add the corresponding structure at the appropriate index based on a probabilistic value depending upon the value of n.



```python
def place_items(world):
    world_data = np.array(world)
    final = []
    p = 0
    height_list = []
    q = deque()
    for i in world_data.T:
        height = 0
        for x in i:
            if x == 90:
                break
            if height == 16:
                break
            height += 1

        height_list.append((p, height))
        p += 1
    i = 0
    while i < len(height_list):
        if height_list[i][1] != 16:
            q.append(height_list[i])
            n = int(1)
            for j in range(i + 1, len(height_list)):
                temp = q.pop()
                q.append(temp)
                if temp[1] == height_list[j][1]:
                    q.append(height_list[j])
                    n += 1
                else:
                    if n == 4 or n == 5:
                        if random.random() >= 0.5:
                            temp = q.pop()
                            world[temp[1] - 1][i] = 70
                            world[temp[1] - 1][i + 1] = 80
                        elif random.random() <= 0.4:
                            temp = q.pop()
                            world[temp[1] - 1][i+1] = 14
                    elif n == 3:
                        if random.random() >= 0.2:
                            temp = q.pop()
                            world[temp[1] - 1][i] = 20
                            #final.append(i)
                    elif n == 2:
                        if random.random() >= 0.2:
                            temp = q.pop()
                            if random.choice([10,40]) == 10:
                                world[temp[1] - 1][i] = 10
                            else:
                                world[temp[1] - 1][i+1] = 40
                            #final.append(i)

                    elif n >= 6 and n < 8:
                        if random.random() >= 0.2:
                            temp = q.pop()
                            world[temp[1] - 1][i+1] = random.choice([12,21])
                            world[temp[1] - 1][i+2] = 15
                            world[temp[1] - 1][i+4] = random.choice([12,21])
                    elif n >= 8:
                        if random.random() >= 0.3:
                            temp = q.pop()
                            if temp[1] >= 5:
                                world[temp[1] - 1][i+1] = 
random.choice([60,90])
                                world[temp[1] - 1][i+2] = 
random.choice([60,90])
                                world[temp[1] - 2][i+2] = 
random.choice([60,90])
                                world[temp[1] - 1][i+3] = 
random.choice([60,90])
                                world[temp[1] - 2][i+3] = 
random.choice([60,90])
                                world[temp[1] - 3][i+3] = 
random.choice([60,90])
                                world[temp[1] - 3][i+4] = 
random.choice([60,90])
                                world[temp[1] - 4][i+4] = 50

                                world[temp[1] - 3][i+5] = 
random.choice([60,90])








                    q.clear()
                    i = j - 1

                    break
        i += 1
    l = 0

    return world
```

Interesting Algorithms

A concept Map of some important and fun algorithms
covered in class

In this section we briefly describe several algorithms, their properties and some algorithms and common problems where they are used.

Optimization Algorithms

Dynamic Programming

- Dynamic programming is an optimization over recursive algorithms. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.
- Dynamic programming can be used for solving problems which only have the optimal substructure property
- It provides quite a fast solution.
- It is used when the subproblem form a Directed Acyclic Graph, node a recurses to b, b recurses to c and so on. If this forms a cycle then we'd get stuck in that cycle and hence the program won't terminate.
- If the problem has both optimal substructure property and greedy choice property then we can use a greedy algorithm.
- Dynamic Programming (DP) is a technique that solves some particular type of problems in Polynomial Time. Dynamic Programming solutions are faster than the exponential brute method and can be easily proved for their correctness.

Steps to solve a DP

1) Identify if it is a DP problem
2) Decide a state expression with least parameters
3) Formulate state relationship
4) Do tabulation (or add memoization) to avoid recomputation of overlapping subproblems

In Dynamic programming the main idea is to reduce recomputing shared subproblems by storing their solutions.

Identifying a problem as a DP problem

1) Check if the problem satisfies the optimal substructure property
2) Problems that require maximizing or minimizing certain quantities, counting problems and certain probability problems can be solved this way.

Deciding a state expression with least parameters

A state can be defined as the least set of parameters that can uniquely identify a certain position or standing in the given algorithm. We minimize the parameters to reduce the amount of space being used.

Formulating a state relationship

We need to find a relation that can be used to move from one state to another i.e., traversing the nodes of the directed acyclic graph formed by the subproblems. This should involve having a valid starting condition and computation which is common to every step in the recursive/iterative loop.

Storing the results of overlapping problems

This is the main objective of DP. In order to reduce unnecessary computations, we store the results of overlapping problems.

Some Problems that have use DP to obtain a solution

1. Longest Common Subsequence
2. Shortest Common Super sequence
3. Longest Increasing Subsequence problem
4. The Levenshtein distance (Edit distance) problem
5. Matrix Chain Multiplication
6. 0–1 Knapsack problem
7. Partition problem
8. Rod Cutting
9. Coin change problem
10. Word break problem

Greedy Algorithms

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are best fit for Greedy.

We use greedy algorithms if a problem satisfies the optimal substructure property and the greedy choice property.

Greedy choice Property

- We must know the first step towards solving the problem

Optimal substructure property

- After taking the first step we must be able to model the remaining problem as a smaller version of the original problem so that induction follows
- Following this we'll get the last step the same as the first step by induction

Examples of problems solved using greedy algorithm

1. Activity Selection Problem
2. Egyptian Fraction
3. Job Sequencing Problem
4. Job Sequencing Problem
5. Huffman Coding
6. Efficient Huffman Coding for sorted input
7. Huffman Decoding
8. Water Connection Problem
9. Policemen catch thieves
10. Minimum Swaps for Bracket Balancing
11. Fitting Shelves Problem
12. Assign Mice to Holes

Greedy algorithms can also provide approximate solutions to NP complete problems such as:

1. Set cover problem
2. Travelling Salesman Problem
3. Graph Coloring
4. K-centers problem

Heuristic Algorithms

A heuristic algorithm is one that is designed to solve a problem in a faster and more efficient fashion than traditional methods by sacrificing optimality, accuracy, precision, or completeness for speed. Heuristic algorithms often times used to solve NP-complete problems, a class of decision problems.

Heuristic means “enabling someone to discover something on their own”. These algorithms use clever tricks to simplify computationally difficult problems.

For problems generally solved using this method, a fast and correct solution may not always be available but a solution can be verified when given.

This method is used when approximate solutions are sufficient and an exact solution may be unnecessary or computationally expensive.

Problems solved using this method

1. Travelling salesman problem: given a list of cities and the distances between each city, what is the shortest possible route that visits each city exactly once? A heuristic algorithm used to quickly solve this problem is the nearest neighbor (NN) algorithm (also known as the Greedy Algorithm). Starting from a randomly chosen city, the algorithm finds the closest city. The remaining cities are analyzed again, and the closest city is found
2. Knapsack Problem: a given set of items (each with a mass and a value) are grouped to have a maximum value while being under a certain mass limit. The heuristic algorithm for this problem is called the Greedy Approximation Algorithm which sorts the items based on their value per unit mass and adds the items with the highest v/m as long as there is still space remaining.
3. Searching and sorting: As a search runs, it adjusts its working parameters to optimize speed, an important characteristic in a search function. The algorithm discards current possibilities if they are worse than already found solutions

Number Theoretic Algorithms

GCD Euclid's Algorithm

If x and y are positive integers with x >= y, then gcd(x,y) = gcd(x mod y,y)

Proof

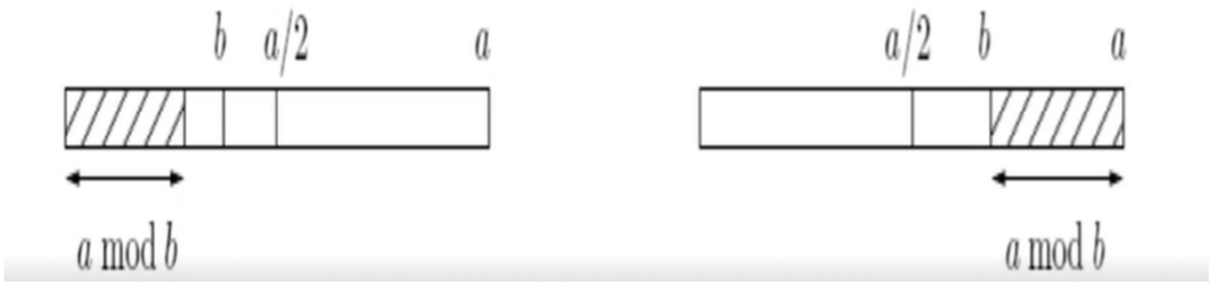
- It is enough to show that gcd(x,y) = gcd(x-y,y) from which the one stated can be derived by repeatedly substracying y from x
- Any integer that divides both x and y must also divide x-y, so gcd(x,y) <= gcd(x-y,y).
- Likewise, any integer that divides both x-y and y must also divide both x and y, so gcd(x,y) >= gcd(x-y,y)

Analysis

- If a >= b, then a mod b < a/2

Proof

- Witness that either b <= a/2 or b > a/2
- if b <=, then we have a mod b < b <= a/2
- if b > a/2 then a mod b = a-b < a/2



- Euclid algorithm is an optimum algorithm
- The overall run time will be bounded by 2loga (base 2) which is O(loga)

Extended Euclid's algorithm

- If d divides both a and b, and d = ax+by for some integers x and y, then necessarily d = gcd(ax+by)

Proof

- By the first two conditions, d is a common divisor of a and b so it cannot exceed the greatest common divisor, that is, d <= gcd(a,b)
- On the other hand, since gcd(a,b) is a common divisor of both a and b, it must also divide ax+by = d , which implies gcd(a,b) <= d.
- Thus from d <= gcd(a,b) and d >= gcd(a,b), we get d = gcd(a,b)

How to find such x and y

- Example: to compute gcd(25,11), Euclid's algorithm would proceed as follows:

$$\underline{25} = 2 \cdot \underline{11} + 3$$

$$\underline{11} = 3 \cdot \underline{3} + 2$$

$$\underline{3} = 1 \cdot \underline{2} + 1$$

$$\underline{2} = 2 \cdot \underline{1} + 0$$

- Lemma: For any positive integers a and b, the extended Euclid algorithm returns integers x.y and d such that gcd(a,b) = d = ax+by
- Inductive Hypothesis: gcd(b,a mod b) = bx' + (a mod b)y'
- Writing (a mod b) as (a - (a/b)b) we find:

$$d = gcd(a,b) = gcd(b, a \ mod \ b) = bx' + (a \ mod \ b)y' = bx' + (a - \frac{a}{b}b)y' = ay' + b(x' - \frac{a}{b}y')$$

Modular Division

- We say x is the multiplicative inverse of a modulo N if ax is congruent to 1 (mod N)

Modular division theorem

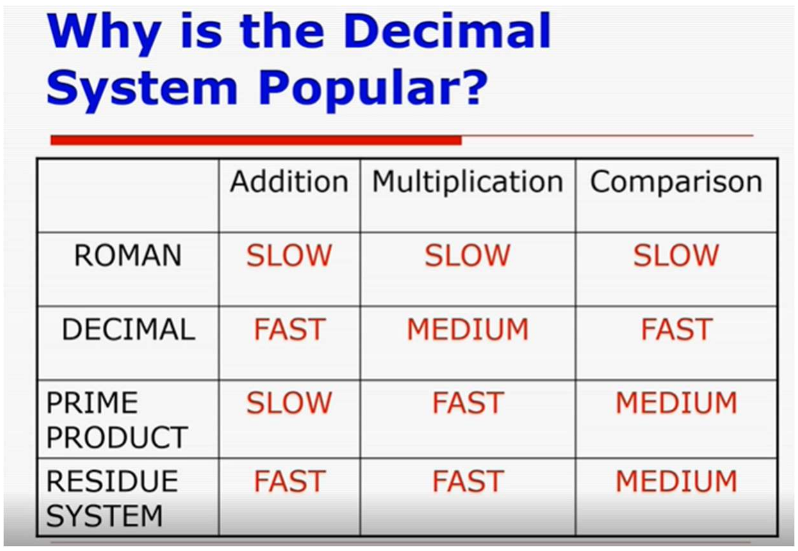
- For any a mod N, a has a multiplicative inverse modulo N if and only if it is relatively prime to N. When this inverse exists, it can be found in time O(n^3) (where as usual n denotes the number of bits of N) by running the extended Euclid algorithm.
- Example. Continuing with our previous example, suppose we wish to compute 11- mod 25.
- Using the extended Euclid algorithm, we find that 15 - 25 – 34 -11 = 1. Reducing both sides modulo 25, we have –34 - 11 = 1 mod 25. So –34 = 16 mod 25 is the inverse of 11 mod 25.

Public key cryptography

How to *Publish* the key and yet be *Secure*?

Ease/Speed of Operation Depends on The Representation

- viii * xvi = cXxviii
- 8 * 16 = 128
- 2^3 * 2^4 = 2^7
- viii + xvi = xxiv
- 8 + 16 = 24
- 2^3 + 2^4 = 2^3.3
- viii < ix is true
- 8 < 9 is true
- 2^3 < 3^2 is true
- Thus if we represent the numbers appropriately, we can perform a particular operation very fast and easily

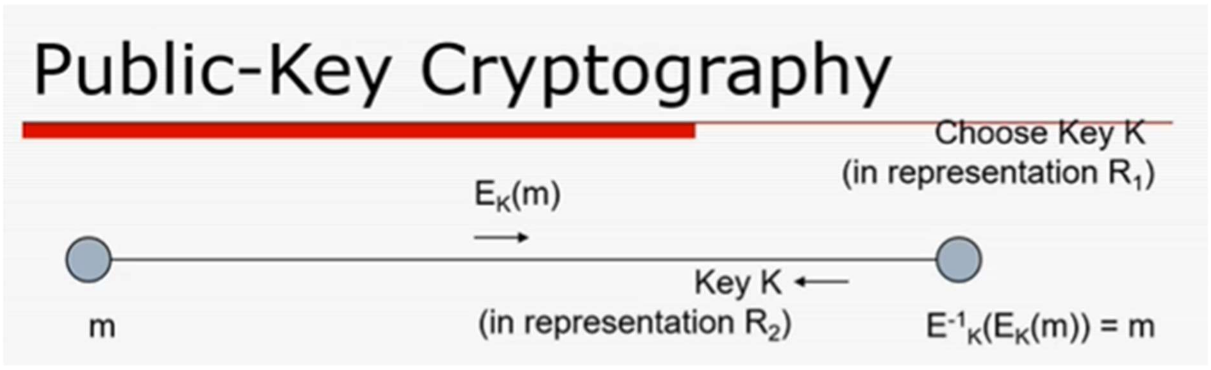


Why is the Decimal System Popular?

	Addition	Multiplication	Comparison
ROMAN	SLOW	SLOW	SLOW
DECIMAL	FAST	MEDIUM	FAST
PRIME PRODUCT	SLOW	FAST	MEDIUM
RESIDUE SYSTEM	FAST	FAST	MEDIUM

Using variable speed of operations in cryptography

- We use this slowness in public key cryptography



- In the above diagram: In representation R2: - Operation Ek will be fast - Operation Ek^-1 will be very slow
- In representation R1 - Operation Ek^-1 is fast

Example RSA Cryptosystem

- R1: Product of primes
- R2: Decimal
- Ek: Modular Exponentiation m^e mod k

RSA

- Pick any two primes p and q and let N = pq.
- For any e relatively prime to (p -1)(q - 1):

- The mapping f(x) = x^e mod N is a bijection on {0, 1, ... N-1}
- Moreover, the inverse mapping is easily realized:
- let d be the inverse of e modulo (p -1)(q - 1).
- Then for all x in {0,...N-1}
- (x^e)^d mod N = x mod N

Correctness

$$x^{ed} - x = x^{1+k(p-1)(q-1)} - x$$

- The above is always 0 modulo N. The second form of the expression is convenient because it can be simplified using Fermat's little theorem. It is divible by p(since x^(p-1) = 1 mod p)and likewise by q. Since pand q are primes, this expression must also be divisible by their product N. Hence x^ed - x = x^(1+k(p-1)(q-1)) - x = 0 (mod N), exactly as we need.

Fermat's little theorem

- If p is prime, then for every 1 <= a < p, a^(p-1) = 1 mod p
- The number a.i mod p are distinct because if a.i = a.j (mod p) then dividing both sides by a gives i = j mod p. They are non-zero because a.i = 0 similarly implies i = 0. We assume a is non zero and relatively prime to p.
- {1,2,….p – 1} = {a · 1 mod p, a - 2 mod p, …., a· (p – 1) mod p}
- (p-1)! = a^(p-1).(p-1)! (mod p)

Randomized Algorithms

A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic or procedure. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random determined by the random bits, thus either the running time, or the output (or both) are random variables.

A very simple example can be randomly shuffling the input before inserting into a binary tree to prevent it from becoming a linked list. Instead of balancing the tree using some parameters, we process the input to prevent this case from arising. This works for the most part and the probability of the input being sorted is significantly reduced.

where the expected running time is finite (Las Vegas algorithms, for example Quicksort), and algorithms which have a chance of producing an incorrect result (Monte Carlo algorithms, for example the Monte Carlo algorithm for the MFAS problem) or fail to produce a result either by signaling a failure or failing to terminate. In some cases, probabilistic algorithms are the only practical means of solving a problem.

Its functioning is simulated using a pseudorandom generator for introducing randomness.

Primality Testing

- Given an input number p, we wish to find out whether p is prime or not

Miller-Rabin Randomized test

- Def: A witness for n is any number 1 <= a < n such that a is co prime to n and either a^(odd(n-1)) = 1 mod n or a^(odd(n-1)2^i) = -1 mod n for some i >= 0
- Consider a hard problem for which there are no efficient algorithm to solve it
- However there exist several efficient algorithm A1, A2, A3...,Ak such that there are atleast 2k/3 values of i for which Ai correctly solves the problem
- A degree of randomness is used as part of the logical procedure for randomized algorithm in the hope of achieving good performance in the average case

- The algorithm: PRIME = On input p:
    - If p is even, accept if p = 2 otherwise reject
    - Select a1,a2,...,ak randomly in Zp+
    - For each i from 1 to k:
        - Compute ai^(p-1) mod p and reject if it is different from 1
        - Let p-1 = st where s is odd and t = 2^h is a power of 2
        - compute the sequence $a_i^{s.2^0}, a_i^{s.2^1}, a_i^{s.2^2}, \ldots, a_i^{s.2^h}$ modulo p.
        - If some element of this sequence is not 1, find the last element that is not 1 and reject if that element is not -1
    - All tests have been passed at this point so accept
- Thus, this method tells us whether a given number is prime or not

Proof

If P is an odd prime number, Probability(PRIME accepts p) = 1

- We first show that if p is prime, no witness exists and so no branch of the algorithm rejects
- If a were a stage 4 witness, (a^(p-1) mod p) != 1 and Fremat's little theorem implies that p is composite.
- If a were a stage 7 witness, some b exists in Zp+, where b != +-1 mod p and b^2 = 1 mod p
- Therefore b^2-1 = 0 mod p. Factoring b^2-1 yields (b-1)(b+1) = 0 mod p
- Which implies that (b-1)(b+1) = cp for some positive integer c.
- Because b != +-1 mod p, both b-1 and b+1 are strictly between 0 and p.
- Therefore p is composite because a multiple of a prime number cannot be expressed as a product of numbers that are smaller than it is

If p is an odd composite number, Probability(PRIME accepts p) <= 2^-k (negligible for large k)

- We show that if p is an odd composite number and a is selected randomly in Zp+ Proba(a is a witness) >= 1/2

- We do this by demonstrating that atleast as many witnesses as nonwitnesses exist in Zp+, we do so by finding a unique witness for each nonwitness
- In every nonwitness, the sequence computed in stage 6 is either all 1s or contains -1 at some position followed by 1s
- For example, 1 itself is a non witness of the first kind, and -1 is a non witness of the second kind because s is odd and (-1)^(s.2^0) = -1
- Similarly -1^(s.2^1) = 1.
- Among all nonwitnesses of the second kind, find a nonwitness for which the -1 appears in the largest position in the sequence
- Let h be that nonwitness and let j be the position of -1 in its sequence, where the sequence positions are numbered starting at 0.
- Hence h^(s.2^j) = -1 mod p
- Because p is a composite number, either p is a power of a prime or we can write p as the product of q and r, two numbers that are relatively prime.
- We consider the latter case first.
- The Chinese remainder theorem implies that some number t exists in Zp whereby t = h mod q and t = 1 mod r
- Therefore:
  $t^{s.2^j} \equiv -1\ mod\ q$
  $t^{s.2^j} \equiv 1\ mod\ r$
- Hence t is a witness because t^(s.2^j) != +- mod p but t^(s.2^(j+1)) = 1 mod p
- Now that we have one witness, we can get many more
- We prove that dt mod p is a unique witness for each non witness d by making two observations.
- First d^(s.2^j) = +- 1 mod p and d^(s.2^(j+1)) = 1 mod p owing to the way j was chosen
- Therefore dt mod p is a witness because (dt)s.2^j != +- 1 and (dt)^(s.2^(j+1)) = 1 mod p
- Second, if d1 and d2 are distinct nonwitnesses, d1t mod p != d2t mod p.

- The reason is that t^(s.2^(j+1)) mod p = 1.
- Hence t.t^(s.2^(j+1)-1) mod p = 1. Therefore if td1 mod p = td2 mod p then
  $d_1 = t.t^{s.2^{j+1}-1} d_1 \ mod\, p = t.t^{s.2^{j+1}-1} d_2 \ mod\, p = d_2$
- Thus the number of witnesses must be as large as the number of nonwitnesses, and we have completed the analysis for the case where p is not a prime power.
- For the prime power case, we have p = q^e where q is prime and e > 1. Let t = 1+q^(e-1)
- Expanding t^p using the binomial theorem we obtain:
- t^p = (1+q^(e-1))^p = 1 + p.q^(e-1) + multiples of higher powers of q^(e-1), which is equivalent to 1 mod p.
- Hence t is a stage 4 witness because if t^(p-1) = 1 mod p then t^p != 1 mod p
- As in the previous case, we use this one witness to get many others
- if d is a non witness, we have d^(p-1) = 1 mod p, but then dt mod p is a witness
- moreover if d1 and d2 are distinct nonwitnesses, then d1t mod p != d2t mod p.
- Otherwise d1 = d1.t.t^(p-1) mod p = d2.t.t^p-1 mod p = d2
- Thus the number of witnesses must be as large as the number of nonwitnesses, and the proof is complete