CS213 2024 Tutorial Solutions

CS213/293 UG TAs

2024

Contents

Tutorial 1

1. *The following is the code for insertion sort. Compute the exact worst-case running time of the code in terms of n and the cost of doing various machine operations.*

Algorithm 1: Insertion Sort Algorithm

```
Data: Array $A$ of length $n$
for $j \leftarrow 1$ to $n-1$ do
    $key \leftarrow A[j]$;
    $i \leftarrow j-1$;
    while $i \geq 0$ do
        if $A[i] > key$ then
            $A[i+1] \leftarrow A[i]$;
        end
        else
            break;
        end
        $i \leftarrow i-1$;
    end
    $A[i+1] \leftarrow key$;
end
```

Solution: To compute the worst-case running time, we must decide the code flow leading to the worst case. When the if-condition evaluates to false, the control breaks out of the inner loop. The worst case would have the if-condition never evaluates false. This will happen when the input is in a strictly decreasing order.

Counting the operations for the outer iterator j (which goes from 1 to $n-1$). Consider for each outer loop iteration:

1. Consider the outer loop without the inner loop. Then it has 1 Comparison (Condition in for loop), 1 Increment (1 Assignment + 1 Arithmetic) (Updation in for loop), 3 Assignments (Line 2, 3, 11) and 2 Memory Accesses (A[j] and A[i+1]) and 1 Jump. Hence $(C + 2Ar + 4As + 2M + J)$
2. while loop runs for j times (as i goes from $j-1$ to 0). Without the if-else, the while loop still has to do 1 Comparison (while condition), 1 Decrement (1 Assignment + 1 Arithmetic) ($i \leftarrow i-1$), 1 Jump, every iteration. Hence $(C + As + Ar + J) * j$
3. The if condition is checked in all iterations of the while loop. It involves 1 Comparison (if) and 1 Memory Access (A[i] in the if condition). if block is executed for all iterations of while in the worst case. if block has 2 Memory Accesses and an Assignment and then a Jump happens to outside the if-else statement. Hence $(C + 3M + As + J) * j$

The total cost would then be (terms are for outer loop, inner loop, if and else):

$$\sum_{j=1}^{n-1} ((C + 2Ar + 4As + 2M + J) + (C + As + Ar + J) * j + (C + 3M + As + J) * j)$$

$$= (C + 2Ar + 4As + 2M + J) * (n-1) + (C + As + Ar + J) * (n * (n-1)/2)$$
$$+ (C + 3M + As + J) * (n * (n-1)/2)$$

This means Insertion Sort is $\mathcal{O}(n^2)$.

(C is Comparison, As is Assignment, Ar is Arithmetic, M is Memory Access, J is Jump)

Note: Some intermediate operations might have been omitted but they do not affect the overall asymptotic performance of the algorithm. The time taken for comparisons, jumps, arithmetic operations, and memory accesses are assumed to be constants for a given machine and architecture.

2. *What is the time complexity of binary addition and multiplication? How much time does it take to do unary addition?*

Solution: Note that time complexity is measured as a function of the input size since we aim to determine how the time the algorithm takes scales with the input size.

Binary Addition
Assume two numbers A and B. In binary notation, their lengths (number of bits) are m and n. Then the time complexity of binary addition would be $\mathcal{O}(m+n)$. This is because we can start from the right end and add (keeping carry in mind) from right to left. Each bit requires an O(1) computation since there are only 8 combinations (2 each for bit 1, bit 2, and carry). Since the length of a number N in bits is $\log N$, the time complexity is $\mathcal{O}(\log A + \log B) = \mathcal{O}(\log(AB)) = \mathcal{O}(m+n)$.

Binary Multiplication
Similar to above, but here the difference would be that each bit of the larger number (assumed to be A without loss of generality) would need to be multiplied by the smaller number, and the result would need to be added. Each bit of the larger number would take $\mathcal{O}(n)$ computations, and then m such numbers would be added. So the time complexity would be $\mathcal{O}(m \times \mathcal{O}(n)) = \mathcal{O}(mn)$[1]. Following the definition above, the time complexity is $\mathcal{O}(\log A \times \log B) = \mathcal{O}(mn)$.

Side note: How do we know if this is the most efficient algorithm? Turns out, this is NOT the most efficient algorithm. The most efficient algorithm (in terms of asymptotic time complexity) has a time complexity of $\mathcal{O}(n \log n)$ where n is the maximum number of bits in the 2 numbers.

Unary Addition
The unary addition of A and B is just their concatenation. This means the result would have $A + B$ number of 1's. Iterating over the numbers linearly would give a time complexity of $\mathcal{O}(A+B)$ which is linear in input size.

3. *Given $f(n) = a_0 n^0 + \ldots + a_d n^d$ and $g(n) = b_0 n^0 + \ldots + b_e n^e$ with $d > e$, then show that $f(n) \notin \mathcal{O}(g(n))$*

Solution: Let us begin by assuming the proposition is False, ergo, $f(n) \in \mathcal{O}(g(n))$. By definition, then, there exists a constant c such that there exists another constant n_0 such that

$$\forall n \geq n_0, f(n) \leq cg(n)$$

. Hence, we have

$$\forall n \geq n_0, a_0 n^0 + \ldots + a_d n^d \leq cb_0 n^0 + \ldots + b_e n^e$$

$$\forall n \geq n_0, \sum_{i=0}^{e} (a_i - cb_i) n^i + a_{i+1} n^{i+1} + \ldots + a_d n^d \leq 0$$

By definition of limit

$$\lim_{n\to\infty} \sum_{i=0}^{e} (a_i - cb_i)n^i + a_{i+1}n^{i+1} + \ldots + a_d n^d \leq 0$$

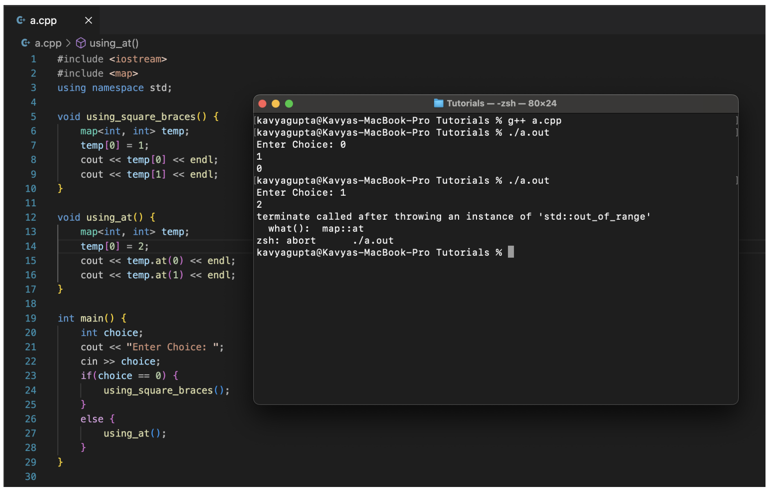
$$\implies a_d \leq 0$$

Assuming $a_d > 0$ (since we are dealing with functions mapping from $\mathbb{N}$ to $\mathbb{N}$), this results in a contradiction; thus, our original proposition is proved.

4. *What is the difference between "at" and "..[..]" accesses in C++ maps?*

Solution: Both accesses will first search for the given key in the map but will behave differently when the key is not present. The at method will throw an exception if the key is not found in the map, but the [ ] operator will insert a new element with the key and a default-initialized value for the mapped type and will return that default value.

Look at the following illustration:-



```cpp
#include <iostream>
#include <map>
using namespace std;

void using_square_braces() {
    map<int, int> temp;
    temp[0] = 1;
    cout << temp[0] << endl;
    cout << temp[1] << endl;
}

void using_at() {
    map<int, int> temp;
    temp[0] = 2;
    cout << temp.at(0) << endl;
    cout << temp.at(1) << endl;
}

int main() {
    int choice;
    cout << "Enter Choice: ";
    cin >> choice;
    if(choice == 0) {
        using_square_braces();
    }
    else {
        using_at();
    }
}
```

```
kavyagupta@Kavyas-MacBook-Pro Tutorials % g++ a.cpp
kavyagupta@Kavyas-MacBook-Pro Tutorials % ./a.out
Enter Choice: 0
1
0
kavyagupta@Kavyas-MacBook-Pro Tutorials % ./a.out
Enter Choice: 1
2
terminate called after throwing an instance of 'std::out_of_range'
  what():  map::at
zsh: abort      ./a.out
kavyagupta@Kavyas-MacBook-Pro Tutorials %
```

5. *C++ does not provide active memory management. However, smart pointers in C++ allow us the capability of a garbage collector. The smart pointer classes in C++ are*

   - auto_ptr
   - unique_ptr
   - shared_ptr
   - weak_ptr

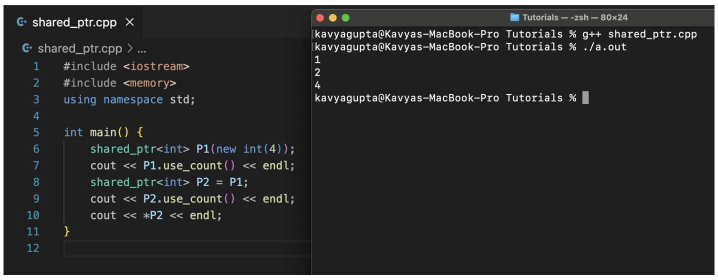
   Write programs that illustrate the differences among the above smart pointers.

Solution: Memory allocated in heap (using new or malloc) if not de-allocated can lead to memory leaks. Smart pointers in C++ deal with this issue. Broadly, they are classes that store reference counts to the memory they point. When a smart pointer is created, reference count to that memory is increased by one. Whenever a smart pointer (pointing to the same memory) goes out of scope, its destructor is automatically executed, which reduces the reference count of that memory by one and when it hits zero, that memory is deallocated.

- shared_ptr allows multiple references to a memory location (or an object)
- weak_ptr allows one to refer to an object without having the reference counted
- unique_ptr is like shared_ptr but it does not allow a programmer to have two references to a memory location
- auto_ptr is just the deprecated version of unique_ptr

The use for weak_ptr is to avoid the circular dependency created when two or more object pointing to each other using shared_ptr.

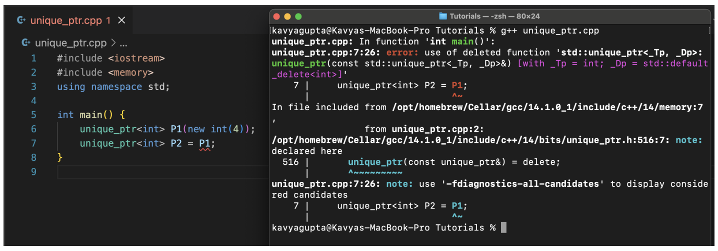
You can see the reference count of a shared_ptr using its use_count() function.



```cpp
#include <iostream>
#include <memory>
using namespace std;

int main() {
    shared_ptr<int> P1(new int(4));
    cout << P1.use_count() << endl;
    shared_ptr<int> P2 = P1;
    cout << P2.use_count() << endl;
    cout << *P2 << endl;
}
```

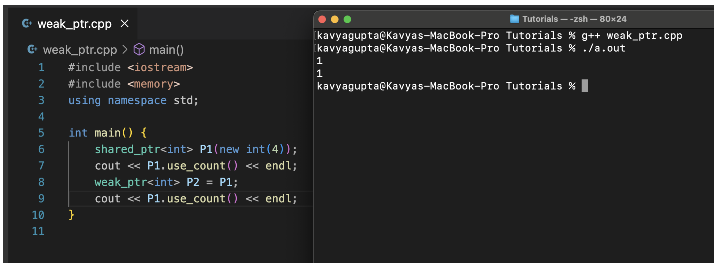
```
kavyagupta@Kavyas-MacBook-Pro Tutorials % g++ shared_ptr.cpp
kavyagupta@Kavyas-MacBook-Pro Tutorials % ./a.out
1
2
4
kavyagupta@Kavyas-MacBook-Pro Tutorials %
```



```cpp
#include <iostream>
#include <memory>
using namespace std;

int main() {
    unique_ptr<int> P1(new int(4));
    unique_ptr<int> P2 = P1;
}
```

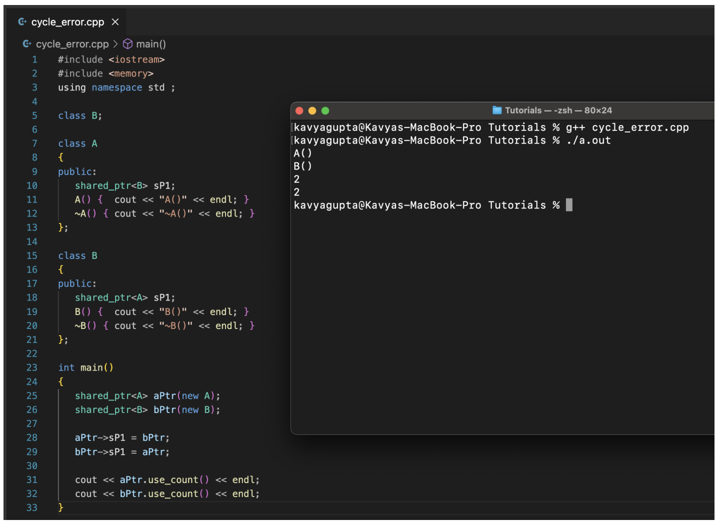
```
kavyagupta@Kavyas-MacBook-Pro Tutorials % g++ unique_ptr.cpp
unique_ptr.cpp: In function 'int main()':
unique_ptr.cpp:7:26: error: use of deleted function 'std::unique_ptr<_Tp, _Dp>::
unique_ptr(const std::unique_ptr<_Tp, _Dp>&) [with _Tp = int; _Dp = std::default
_delete<int>]'
    7 |     unique_ptr<int> P2 = P1;
      |                          ^~
In file included from /opt/homebrew/Cellar/gcc/14.1.0_1/include/c++/14/memory:7
,
                 from unique_ptr.cpp:2:
/opt/homebrew/Cellar/gcc/14.1.0_1/include/c++/14/bits/unique_ptr.h:516:7: note:
declared here
  516 |       unique_ptr(const unique_ptr&) = delete;
      |       ^~~~~~~~~~
unique_ptr.cpp:7:26: note: use '-fdiagnostics-all-candidates' to display conside
red candidates
    7 |     unique_ptr<int> P2 = P1;
      |                          ^~
kavyagupta@Kavyas-MacBook-Pro Tutorials %
```



```cpp
#include <iostream>
#include <memory>
using namespace std;

int main() {
    shared_ptr<int> P1(new int(4));
    cout << P1.use_count() << endl;
    weak_ptr<int> P2 = P1;
    cout << P1.use_count() << endl;
}
```

```
kavyagupta@Kavyas-MacBook-Pro Tutorials % g++ weak_ptr.cpp
kavyagupta@Kavyas-MacBook-Pro Tutorials % ./a.out
1
1
kavyagupta@Kavyas-MacBook-Pro Tutorials %
```



```cpp
#include <iostream>
#include <memory>
using namespace std ;

class B;

class A
{
public:
    shared_ptr<B> sP1;
    A() {  cout << "A()" << endl; }
    ~A() { cout << "~A()" << endl; }
};

class B
{
public:
    shared_ptr<A> sP1;
    B() {  cout << "B()" << endl; }
    ~B() { cout << "~B()" << endl; }
};

int main()
{
    shared_ptr<A> aPtr(new A);
    shared_ptr<B> bPtr(new B);

    aPtr->sP1 = bPtr;
    bPtr->sP1 = aPtr;

    cout << aPtr.use_count() << endl;
    cout << bPtr.use_count() << endl;
}
```

```
kavyagupta@Kavyas-MacBook-Pro Tutorials % g++ cycle_error.cpp
kavyagupta@Kavyas-MacBook-Pro Tutorials % ./a.out
A()
B()
2
2
kavyagupta@Kavyas-MacBook-Pro Tutorials %
```

6. *Why do the following three writes cause compilation errors in the C++20 compiler?*

```
class Node {
    public :
    Node(): value(0) {}
    const Node& foo(const Node* x) const {
        value = 3; // Not allowed because of ______________
        x[0].value = 4; // Not allowed because of ______________
        return x[0];
    }
    int value;
};

int main () {
    Node x[3], y ;
    auto &z = y.foo(x);
    z.value = 5; // Not allowed because of ______________
}
```

Solution: The line 'value = 3;' is not allowed since the method 'foo' is marked **const** at the end. Using **const** after the function signature and its parameters for a class method implies that the class members cannot be changed and the object itself is constant within the bounds of the function. As a result, an error is raised.

The line 'x [0]. value = 4;' raises an error since the parameter 'x' is of type '**const** Node*', meaning that different values can be assigned to the pointer itself, but the subscripts of the pointer cannot be assigned, since it points to constant members of class 'Node'. Likewise, '*x.value = 3;' is equally illegal. Note that 'Node **const** *' also does the same thing, whereas 'Node * **const**' means that the subscripts of the pointer can be assigned since it does not point to constant members of class 'Node' but the pointer itself cannot be assigned to point to a different Node or array of Nodes. '**const** Node* **const**' or 'Node **const** * **const**' imply that the pointer cannot be assigned AND subscripts or dereferences cannot be assigned.

The last line 'z.value = 5;' is illegal since the datatype of 'z', as determined from the method 'foo' is '**const** Node &' NOT 'Node &'. The implication is that 'z' refers to a Node, which is constant and hence cannot be assigned. Note that assignments to class members are as bad as assignments to the class object itself regarding constant objects in C++.

Tutorial 2

1. *The span of a stock's price on i^{th} day is the maximum number of consecutive days (up to i^{th} day and including the i^{th} day) the price of the stock has been less than or equal to its price on day i.*

   Example: for the price sequence 2 4 6 5 9 10 of a stock, the span of prices is 1 2 3 1 5 6.

   Give a linear−time algorithm that computes s_i for a given price series.

Solution: The idea here is to find out the latest (most recent) day till the i^{th} day where the price was greater than the price on day i. To do this, we will maintain a stack of indices, and for each index i, we will keep the stack in a state such that the following invariant holds for each day i:

If the stack is not empty, then the price on the day whose index is at the top of the stack is the most recent price that was strictly greater than the current price.

With this invariant, the updates to the stack and the stock span follow as:

- we remove the top index in the stack till the price corresponding to the top index becomes strictly greater, or the stack becomes empty
- if the stack is empty, the current price is the largest, so the span is the number of days so far
- otherwise the span is the number of days since the index at the top of the stack

(Such a stack is known as a "Monotonic Stack").

The base case is that at day 1 the price is the only encountered price, and hence is the largest, so the span is 1.

The time complexity is indeed $\mathcal{O}(n)$ (n being the number of days/size of array s) because each element is pushed **exactly once** in the stack and popped **atmost once**.

Look at Algorithm 2 for the pseudo-code.

Algorithm 2: Stock Span Algorithm

```
Data: Array $price$ of length $n$ and an empty stack $St$
for $i \leftarrow 0$ to $n-1$ do
    $s[i] \leftarrow i+1$ //If price[i] is $\geq$ all the previous prices, then $s[i]$ will be $i+1$
    while not $St.isEmpty()$ do
        if $price[St.top()] > price[i]$ then
            $s[i] \leftarrow i - St.top()$
            break
        end
        $St.pop()$
    end
    $St.push(i)$
end
```

2. *There is a stack of dosas on a tava, of distinct radii. We want to serve the dosas of increasing radii. Only two operations are allowed:*

   (i) serve the top dosa
   (ii) insert a spatula (flat spoon) in the middle, say after the first k, hold up this partial stack and flip it upside-down and put it back

Design a data structure to represent the tava, input a given tava, and to produce an output in sorted order. What is the time complexity of your algorithm?

This is also related to the train-shunting problem.

Solution: Read the first line of the question with the emphasis being on **stack**. We will use the array representation of a stack for the tava.

Our stack-tava has the following abstraction:

- Initialization and input: Initialize an array S of size n, to represent the stack. Using the given input, fill the array elements with the radii of dosas on the tava in the initial stack order, with the bottom dosa at index 0 and the top at $n-1$. Maintain a variable sz which contains the current size of the stack, and initialize it to n.
- Serving the top dosa: This method is essentially carrying out the "pop" operation on our stack S. Return the top dosa in the stack, $S[sz-1]$, and decrement sz by 1. Clearly, this method takes a constant $\mathcal{O}(1)$ time for every call.
- Flipping the top k dosas: This method is equivalent to reversing the slice of the array S from index $sz-k$ to $sz-1$. This is simple enough: initialize an index i to $sz-k$ and another index j to $sz-1$, swap elements at indices i and j, increment i and decrement j by 1, and repeat the swap and increment/decrement while $j > i$. This method takes $\mathcal{O}(k)$ time for every call.

Note that in this implementation, the serve operation is the only way by which the problem can be reduced in size (number of dosas reduced by 1). The flip operation, if used wisely, can give us access to dosas that can help us reduce the problem size.

With this implementation, the algorithm to serve the dosas in increasing order of the radii can be designed as follows. While the stack is not empty, repeat the following:

- Iterate over the array S and find the index of the minimum element in the array. Let this index be m. This is the position of the dosa having the smallest radius
- Flip over the top $sz-m$ dosas in the stack, using the flipping operation. This brings the smallest dosa from index m to index $sz-1$, id est, the top of the stack
- Serve the top dosa, using the serving operation. This pops the smallest dosa off the stack

It is easy to argue the correctness. The invariant is that we keep serving the minimum radii dosa from the remaining stack. The resulting order has to be sorted.

For computing the time complexity: in a stack of size z, finding the index of the minimum element takes $\mathcal{O}(z)$ time, flipping over the top $k \leq z$ elements takes $\mathcal{O}(k)$ time (and thus $\mathcal{O}(z)$ time) and serving off the top dosa takes $\mathcal{O}(1)$ time. Thus serving the smallest dosa takes $\mathcal{O}(z)$ time overall. This has to be repeated for all dosas, so the stack size z goes from n to 1, decrementing by 1 every time. Therefore the algorithm has a time complexity of $\mathcal{O}(n^2)$.

3. *(a) Do the analysis of performance of exponential growth if the growth factor is three instead of two? Does it give us better or worse performance than doubling policy?*

   (b) Can we do the similar analysis for growth factor 1.5?

Solution: Let us do the analysis for a general growth factor α. Suppose initially N = 1 and there are n = α^i consecutive pushes. So, total cost of expansion is (refer to slides for detailed explanation):

$$(\alpha + 1) \cdot (\alpha^0 + \alpha^1 + \alpha^2 ... + \alpha^{i-1})$$
$$\frac{\alpha + 1}{\alpha - 1} \cdot (\alpha^i - 1)$$
$$\frac{\alpha + 1}{\alpha - 1} \cdot (n - 1)$$

For α equals to 3 cost of expansion is $2 \cdot (n - 1)$, it's better than doubling policy. For α equals 1.5 cost of expansion is $5 \cdot (n - 1)$, which is worse than doubling policy. Trade-off involved is extra memory allocation, maximum extra memory allocated is $\alpha - 1$ times the requirement, so with increasing alpha extra memory allocation is increasing.

4. *Give an algorithm to reverse a linked list. You must use only three extra pointers.*

Solution: Let us initialise three pointers - *prev, curr*, and *next*, initialised to null, head and $head -> next$ respectively. If head is null, then it is an empty linked list and we return. Otherwise, we will be iterating over each element in the linked list. In each iteration, perform the following updates.

- $curr -> next$ = prev
- prev = curr
- curr = next
- next = $next -> next$

The loop terminates when *next* is null, in which case, set head to curr.

5. *Give an algorithm to find the middle element of a singly linked list.*

Solution: The idea is as follows:

- Make 2 pointers middle and end initialized to head of linked list
- At each step, increment middle by one and end by 2
- Continue this process until end reaches the end of linked list
- return the middle element
- return null if the head itself was null (empty linked list)

Note: When there is an odd number of elements, the above algorithm returns the element at the median location. What happens when there is an even number of elements?

6. *Given two stacks S1 and S2 (working in the LIFO method) as black boxes, with the regular methods: "Push", "Pop", and "isEmpty", you need to implement a Queue (specifically: Enqueue and Dequeue working in the FIFO method). Assume there are n Enqueue/Dequeue operations on your queue. The*

time complexity of a single method Enqueue or Dequeue may be linear in n, however the total time complexity of the n operations should also be $\Theta(n)$

Solution:

Let us try to formulate a simple implementation of a queue using 2 stacks.

Label the stacks as A and B. We will use stack A to store elements of the queue sequentially, and stack B to implement the dequeue operation.

Enqueue: To insert a new element to the queue, we can just *push* the element to the top of stack A. Here, each *enqueue* operation requires $\mathcal{O}(1)$ time.

Dequeue: To remove an element from the queue in a FIFO manner, we must somehow delete the first element pushed to stack A. Since we cannot access any element of the stack other than the top one (last inserted), we will have to use stack B to implement this.

To get access to the first inserted element, we need to *pop* all the remaining elements from stack A and store them into stack B. As you can see in the figure 1, the order of elements is reversed while moving them to stack B. When we reach the first element of A, we can just *pop* it without then adding it to B.

We must now decide what to do with the elements of stack B. One idea is to simply use stack B for storing the queue in the place of stack A, but this doesn't work since the order of elements would change.

Another possible approach would be to again transfer all the elements saved in stack B to stack A and restore the queue. This would preserve the order of elements and allow new elements to be pushed to A while maintaining correctness. However, think of the worst case time complexity of this approach. For every *dequeue* operation, all the elements have to be shifted twice - once from A to B and then from B to A. This causes each *dequeue* operation to be of $\mathcal{O}(n)$ time complexity and the total time complexity of n operations to be $\mathcal{O}(n^2)$. (think when?)

The blowup in time complexity is caused due to elements being shifted back to A from B on every *dequeue*. Do we really need that? What happens if we just let the elements stay in B?

What happens now if we have 2 successive dequeue operations? Suppose we transferred the elements of queue from stack A to B in the first operation. For the 2nd dequeue operation, we can simply *pop* from stack B. Hence we don't need to re-transfer the elements. Now, if we have n operations, our complexity reduces to $\mathcal{O}(n)$.

To sum it up: Look at Algorithm 3 for the pseudo-code

Algorithm 3: Queue using Two Stacks

```
1  Function Enqueue(element):
2      S1.Push(element)
3  end
4  Function Dequeue():
5      if S1.isEmpty() and S2.isEmpty() then
6          throw("Queue is empty.")
7      end
8      if S2.isEmpty() then
9          while not S1.isEmpty() do
10             S2.Push(S1.Pop())
11         end
12     end
13     return S2.Pop()
14 end
```

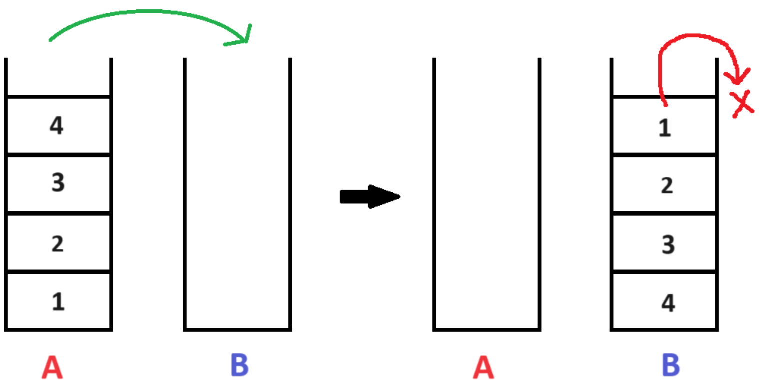


Figure 1: 2 Stacks making a Queue

Tutorial 3

1. *What is the probability for the 3rd insertion to have exactly two collisions while using linear probing in the hash table?*

Solution: Let the size of the hash table be $n \geq 3$. We assume that insertions take place via linear probing. Further, we assume that no deletions or insertions occur in the same sequence. Lastly, we assume that the hash function is uniform, i.e., for a random key, the likelihood of any hash value occurring is exactly $1/n$. After collisions, let the first two insertions occur at indices i and j.

If $((i+1) \ mod \ n) \neq j$ and $((j+1) \ mod \ n) \neq i$, then the third insertion cannot have two collisions. The reason is that if the hash value of the third key collides with the first key, then index $(i+1) \ mod \ n$ is vacant, and there is only one collision. If the hash value is at index j, then index $(j+1) \ mod \ n$ is vacant and hence, only one collision. Any other hash value implies no collision. Let us neglect this case.

We focus only on cases where i and j are adjacent to each other on the table. Let us re-label the indices as a and b such that $((a+1) \ mod \ n) = b$. In this case, for any constant $0 \leq c < n$, the absolute value of $P(a = c)$ (not conditional) can be thought of as the probability that (($i = c$ and $j = (i+1) \ mod \ n$) or ($j = c$ and $i = (j+1) \ mod \ n$)). The first event occurs when the second hash value is either i or $(i+1) \ mod \ n$, with a chance of $(1/n) \cdot (2/n)$. The second event occurs when the second hash value is $(n+i-1) \ mod \ n$, with a chance of $(1/n) \cdot (1/n)$.

From the above, we can say that $P(a = c) = (3/n^2)$ for all constants c, we safely say that two collisions occur if and only if the third hash-value is a, which occurs with a probability of $\frac{1}{n}$. However, c and, therefore, a can take any value. Thus, the product $(3/n^2) \cdot (1/n)$ has to be summed up across all values (n times), giving us the required answer: $(3/n^2)$.

2. *Given that k elements have to be stored using a hash function with target space n. What is the probability of the hash function having an inherent collision? What is an estimate of the probability of a collision in the insertion of n elements?*

Solution: To compute the probability, we first count the negative case: no collisions at all. This means that the k keys get mapped to distinct k keys out of n. We assume that for a random key and for any constant c, $P(hash(k) = c) = 1/n$.

The first key has n choices of hash values. Given each choice of the first key, the second key has $n-1$ possible hash values such that there is no collision. Given the first two keys and hash values, the third has $n-2$ choices, and so on, till the k^{th} key has $n-k+1$ choices. Overall, the ordered k-tuple has $\prod_{i=0}^{k-1}(n-i)$ choices for no collision. All those are mutually exclusive events with likelihoods $(1/n)^k$ each, being the product of k probabilities of $1/n$ for k independent events and their respective outcomes. Thus,

$$P(No\ collisions) = \frac{n!}{(n-k)! \cdot n^k}$$

One minus this expression gives the likelihood of a collision.

If $k = n$, the expression boils down to $(n!/n^n)$. We apply Stirling's approximation here:

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n+1}} \ < \ n! \ < \ \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}$$

$$\sqrt{2\pi n}\left(\frac{1}{e}\right)^n e^{\frac{1}{12n+1}} \;<\; \frac{n!}{n^n} \;<\; \sqrt{2\pi n}\left(\frac{1}{e}\right)^n e^{\frac{1}{12n}}$$

For large n, we can neglect the terms $e^{(1/(12n+1))}$ and $e^{(1/12n)}$ as both are very close to 1. However, the term $\sqrt{2\pi n}\cdot e^{-n}$ tends to zero as n grows very large. This means that the required probability is sandwiched between terms that exponentially decay toward zero.

The implication of the above is the likelihood that all n keys get uniquely mapped to the n possible hash values decay exponentially with n, and thus, collisions are almost certain for large n. This can be linked with the birthday paradox in that in a class of 365; it is almost impossible for all students to have different birthdays; even a single tutorial batch has a high likelihood of at least two birthdays being the same. In the context of hashing, although the expected number of terms sharing a hash value is one per hash value, we expect a good number of collisions to occur during hashing.

3. *Let $C(i)$ be the chain of array indices that are queried to look for a key k in linear probing where $h(k) = i$.*

   (a) *How does this chain extend by an insertion, and how does it change by a deletion?*

   (b) *A search for a key k ends when an empty cell is encountered. What if we mark the end of C (i) with an end marker? We stop the search when this marker is encountered. Would this work? Would this be efficient?*

   (c) *Is there a way of not using tombstones?*

Solution: To recall, linear probing covers indices i, $i+1$, ... until an empty index is found (then k not found in the table) or the cell stores k. Note that tombstones do not count as empty cells in the hash table; counting them so after deletion would break existing chains and result in false negatives when searching. Including them means no change in these chains, answering part of the first (a) subproblem.

If the chain terminates at k (key found), then inserting a new element does not change anything. If the added element is k itself, then it occupies the position at the end of the chain, a vacant cell. So, no change. In the other case, the chain extends by one if the inserted element is at the end of the chain and is not k. The chain could extend a lot more if the inserted element joins the existing chain with another series of filled cells in the table. Note that the chain, except for the end, has only filled or tombstone cells; thus, the insertion cannot occur in the middle of the chain. If the insertion of the element that is not k does not occur at the end of the previous chain for searching k, then there is no change.

The answer to the second (b) part is that it would work but not be any more efficient than the standard algorithm. For this to work, an empty cell once encountered during the search should be marked an 'end.' Otherwise, the first search would go on an infinite loop, looking for an end marker. If this is the case, the search must look for both end marks and empty cells in each iteration, making it slightly less efficient but at the same time complexity.

For the third (c) part, a slight modification should be made to the algorithm discussed above: all cells should be marked 'end' at the start. Otherwise, a deletion without a tombstone would cause false negatives in future searches, just like the discussions without tombstones. If this modification is made, then we basically replace "empty" with an "end" marker and "tombstone" marker with "empty". The algorithm is, otherwise, identical to the normal algorithm with tombstones, and hence, we really do not avoid tombstones, although we do not use them. Note that marking the freed cells as 'end' on deletions would be as bad as not using tombstones, which results in false negatives.

4. *Let $m = 11$, $h_1(k) = (k \text{ mod } 11)$, $h_2(k) = 6 - (k \text{ mod } 6)$.*
Let us use the following hash function for an open addressing scheme.

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \text{ mod } m$$

1. *What will be the state of the table after insertions of 41, 22, 44, 59, 32, 31, and 74?*
2. *Let $h_2(k) = p - (k \text{ mod } p)$. What should be the relationship between p and m such that h is a valid function for linear probing?*
3. *What is the average number of probes for an unsuccessful search if the table has α load factor?*
4. *What is the average time for a successful search?*

Solution: To recall, the standard linear probing uses the function $h(k,i) = (h(k) + i) \text{ mod } m$ where m is the size of the hash table [Here 11]. The way it works is that we check the cells indexed at $h(k,0), h(k,1), h(k,2), \ldots$ till we find an empty cell $h(k,a)$ for some a. The same logic should be extended here, except $h(k,i)$, which is defined in the problem statement.

Let the table before adding 41 be

Now, $h_1(41) = 8$ and $h_2(41) = 1$. This means that $h(41,0) = 8$ and cell 8 is empty. So, we add 41 there:

								41		

$h_1(22) = 0$ and cell 0 is empty so we can add 22 there. $[h(k,0) = h_1(k)]$

22								41		

$h_1(44) = 0$ but cell 0 is filled. Now, $h_2(44) = 4$ and $h(44,1) = 4$ which is an empty tile.

22				44				41		

$h_1(59) = 4$ which is a filled cell. $h_2(59) = 1$ and $h(59,1) = 5$ is an empty tile.

22				44	59			41		

$h_1(32) = 10$ and $h_1(31) = 9$ are both empty and different cells, so we can add them.

22				44	59			41	31	32

Lastly, $h_1(74) = 8$, which is occupied. $h_2(74) = 4$. Now, $h(74,1) = 1$ which is free.

22	74			44	59			41	31	32

This is how the table looks after inserting the elements in the given order.

Moving on to the next part, a valid function for linear probing should eventually cover all cells in the grid as i increments. This is not possible if $gcd(h_2(k), m) > 1$, for one or more values of k. The reason is that $h(k,i)\ mod\ gcd(h_2(k), m)$ will be fixed, and hence, $h(k,i)\ mod\ m$ will only take some values. But if that gcd is 1, then it means that for all $0 \le i < m$, $gcd \cdot i\ mod\ m$ will have a unique value, and the number of these unique values will be m, so all cells of the table will be covered.

Note that $gcd(h_2(k), m) = 1$ should hold for all values of k, not just for any one k. Keeping $h_2(k) = p - (k\ mod\ p)$ can take any value in $1, 2, \ldots p$ depending on the value of k. Thus, none of the first p natural numbers should share a common factor with m. In other words, the smallest prime factor (not 1) of m, the size of the hash table, should be larger than p. It is not necessary that m is prime – for $p = 6$, $m = 7 \cdot 13 = 91$ also works.

Exact expression of expected number of probes in a search (unsuccessful/successful) is mathematically complex hence we'll prove an appropriate upper bound for them. Claim: The mentioned expression for an unsuccessful one is atmost $1/(1-\alpha)$. This is assuming $\alpha < 1$.

Proof: Let X be the number of probes made in an unsuccessful search, and let A_i be the event that an i^{th} probe occurs and it is to an occupied slot. Then $X \ge i$ when the event $A_1 \cap A_2 \cap \ldots \cap A_{i-1}$ occurs. Hence,

$$\begin{aligned} P(X \ge i) &= P(A_1 \cap A_2 \cap \ldots \cap A_{i-1}) \\ &= P(A_1) \times P(A_2|A_1) \times P(A_3|A_2 \cap A_1) \times \ldots \times P(A_{i-1}|A_1 \cap \ldots \cap A_{i-2}) \end{aligned}$$

$P(A_1) = n/m$ because there are n elements and m slots. For $j > 1$, the probability that there is a j^{th} probe and it is to an occupied slot, given that the first $j-1$ probes were to occupied slots, is $(n-j+1)/(m-j+1)$ because we are finding one of the remaining $(n-(j-1))$ elements in one of the $(m-(j-1))$ unexamined slots, and we have uniform hashing so each of the slots is equally likely to be chosen next.

Because we have $n < m$, we have $(n-j)/(m-j) \le n/m \ \forall\ 0 \le j < m$. Thus,

$$P(X \ge i) = \frac{n}{m} \times \frac{n-1}{m-1} \times \ldots \times \frac{n-i+2}{m-i+2} \le (\frac{n}{m})^{i-1} = \alpha^{i-1}$$

So $E[X] = \sum_{i=1}^{m-1} P(i^{th}$ probe occurs$) = \sum_{i=1}^{m-1} P(X \ge i) \le \sum_{i=1}^{m-1} \alpha^{i-1} \le \sum_{i=1}^{\infty} \alpha^{i-1} = 1/(1-\alpha)$. Hence proved.

Claim: Expected number of probes for a successful search is atmost $\frac{1}{\alpha} ln(\frac{1}{1-\alpha})$.

Proof: A search for a key k reproduces the same probe sequence as when the element with key k was inserted. If k was the $(i+1)^{th}$ key inserted into the hash table, there were only i keys in the table at the time, so the expected number of probes made in a search for k is at most $1/(1-i/m) = m/(m-i)$ (as an insertion is equivalent to an unsuccessful search, here $\alpha = i/m$). Averaging over all n keys in the hash table, we get,

$$\begin{aligned} \frac{1}{n}\sum_{i=0}^{n-1} \frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1} \frac{1}{m-i} \\ &= \frac{1}{\alpha}\sum_{k=m-n+1}^{m} \frac{1}{k} \\ &\le \frac{1}{\alpha}\int_{m-n}^{n} \frac{1}{x}\,dx \\ &= \frac{1}{\alpha} ln(\frac{m}{m-n}) \\ &= \frac{1}{\alpha} ln(\frac{1}{1-\alpha}) \end{aligned}$$

Hence Proved.

5. *Suppose you want to store a large set of key-value pairs, for example, $(name, address)$. You have operations in this set: addition, deletion, and search of elements. You also have queries about whether a particular name or address is in the set, and if so, then count them and delete all such entries. How would you design your hash tables?*

Solution: Let us recall that in the classic hashing, the hash function operates on the key, and the value is simply stored alongside the key. An intuitive idea is to keep the key (from the point of view of hashing) as **both** the name and the address, with no special value otherwise. The hash function operates on this and gives us the index. Unfortunately, this gives rise to a problem: there are multiple different hash values for a given name, and hence, searching for the existence of an entry given a name (but not address – we do not know which addresses are there given this name) is as bad as linearly iterating through the table.

One way to get around this is to use a two-level hash table: the outer table N stores the keys as the names alone, and each value corresponding to a name points to a hash table itself. This inner hash table $N[name]$ stores only addresses and only addresses where the name is the corresponding key of the outer table. This makes finding elements based on names easy but does not solve the problem of searching for given addresses.

To solve that, we can use a double two-level hash table: one two-level table N as described above and one two-level table A done the other way round, i.e., A stores the keys as addresses and values as pointers to inner hash tables $A[address]$ that store names corresponding to addresses. Note that this requires additional storage space (double) and more maintenance.

Specifically, it is important to ensure integrity: for every $(name, address)$ pair in the set, $N[name]$ should exist and contain *address*, and $A[address]$ should exist and contain *name*. Insertion should be done in both tables at once. On the other hand, if any $(name, address)$ pair does not exist, then $A[address]$, if it exists, should not contain *name*, and $N[name]$, if it exists, should not contain *address*. This should be reflected in the deletion of pairs.

Tutorial 4

1. *Given a tree with a maximum number of children as k. We give a label between 0 and $k-1$ to each node with the following simple rules.*

   (i) The root is labelled 0.
   (ii) For any vertex v, suppose that it has r children, then arbitrarily label the children as $0, \ldots, r-1$.

   This completes the labelling. For such a labelled tree T, and a vertex v, let $Seq(v)$ be the labels of the vertices of the path from the root to v. Let $Seq(T) = \{Seq(v) \mid v \in T\}$ be the set of label sequences.

   What properties does $Seq(T)$ have? If a word w appears, what words are guaranteed to appear in $Seq(T)$? How many times does a word w appear as a prefix of some words in $Seq(T)$?

Solution: Some properties are $Seq(T)$ are:-

1. Size of this set will be n (n is the number of nodes in the tree) (See that for different v_1, v_2 vertices, $Seq(v_1) \neq Seq(v_2)$).
2. If there are h_j nodes of j length in set $Seq(T)$, then the tree has h_j nodes at j^{th} level.
3. If we consider ordering of set $Seq(T)$ in lexicographical order as the metric, then it will just represent a DFS on the tree.
4. If we consider ordering of set $Seq(T)$ by length followed by lexicographical order as the metric, then it will just represent a BFS on the tree.

$p_0 = 0$ (root of tree)

$$w = p_0p_1p_2..p_n$$
$$S_w = \{x \mid x = p_0p_1p_2..p_iq \text{ where } 0 \leq i \leq n-1 \ \& \ 0 \leq q \leq p_{i+1}\} \bigcup \{0\}$$

We can prove S_w is the set which is guaranteed to be present if w is present by following argument:- Each element of set S_w is guaranteed to be present if w is present. We can prove this by considering two claims,

1. Any prefix of w will be present since this just represents the path from root to ancestor of w.
2. If a word $l_0l_1l_2..l_n$ is present then all the words of form $l_0l_1l_2..l_{n-1}q$ where $0 \leq q \leq l_n$ will also be present since these are just the child of the node represented by sequence $l_0l_1l_2..l_{n-1}$

If $w = Seq(v)$ for some $v \in T$ and the number of nodes in the subtree of v be x, then w appears as a prefix for x number of words in $Seq(T)$.

2. *Write a function that returns $lca(v, w, T)$. What is the time complexity of the program?*

Solution: This is for a general tree and not just a binary tree.

See the Pseudo-code in Algorithm 4.

At any node (say x), if you find the LCA in x's subtree, return it. Otherwise, see if v, w are present in x's subtree (subtree includes x). If yes, then x is the LCA, otherwise return $NULL$.

Time complexity is $\mathcal{O}(n)$, n being the number of nodes of T, as we go through each node atmost once.

If there existed a "parent" pointer in each node, then can you form a new LCA algorithm ?

Algorithm 4: LCA

```
Function LCA(curr, v, w):
    if curr = NULL then
        return False, False, NULL //This return type is an ordered tuple
    end
    v_present ← (curr = v) // True or False
    w_present ← (curr = w)
    for child ∈ children[curr] do
        v_in_child, w_in_child, LCA_child ← LCA(child, v, w)
        if LCA_child ≠ NULL then
            return True, True, LCA_child
        end
        v_present ← v_present || v_in_child
        w_present ← w_present || w_in_child
    end
    if v_present = True and w_present = True then
        return True, True, curr
    end
    return v_present, w_present, NULL
end
// Call LCA(root, v, w) in the main function
```

3. *Given $n \in T$, Let $f(n)$ be a vector, where $f(n)[i]$ is the number of nodes at depth i from n.*
   - *Give a recursive equation for $f(n)$.*
   - *Give a pseudo-code to compute the vector $f(root(T))$. What is the time complexity of the program?*

Solution:

Observations:

- For any node n, the only node at depth 0 from n is the node n itself.
- For a leaf node l, there is no node n such that *depth(n)* from l is greater than or equal to 1.
- For a node n, if node m is at depth k, then there exists a *unique path* from n to m of length k, and this path must go through one of the children of n (let's say p) i.e. node m is at a depth $k-1$ from node p.
  Conversely, if a node m is at a depth $k-1$ from node p, then it must be at a depth of k from node n where n is parent of p.

With these observations in mind, we can say that $f(n)[i] = \sum_{m \in children(n)} f(m)[i-1]$. Recursive equation of $f(n)$ then can be written as:

$$f(n) = [1] + (\sum_{m \in children(n)} f(m))$$

which is nothing but concatenating [1] in front of the sum obtained from the f vectors of the children.
Here base case will be when n is a leaf, for that we have

$$f(n) = [1, 0, 0, \cdots, 0]$$

Pseudocode: (Look at Algorithm 5)
As you all know, we will be doing some kind of traversal to get $f(root(T))$!!!
Can you think of the traversal method we gonna use by looking at the recursion obtained in **part 1** ???
I hope you got it correct, we are going to use post-order traversal (why so?)
We will use array data structure for storing $f(i)$ of size n, where n is the number of nodes in the tree
The algorithm given above, does a post order traversal on tree (note that this algorithm holds good for any tree and not just a binary tree), on getting to a leaf, it returns an array $[1, 0, \cdots, 0]$, and on a internal node, it returns f according to the recursion obtained above.
Time complexity
Assume that vectors at all the nodes are of size n. There will be $\mathcal{O}(n)$ edges and for each edge we will do addition of two such vectors. Hence this algorithm has $\mathcal{O}(n^2)$ time complexity.

Algorithm 5: Computing $f(n)$

```
Data: Tree $T$ having $n$ nodes
Function ModifiedPostOrderWalk($p$):
    $f \leftarrow [1, 0, \ldots, 0]$
    if $p.isLeaf()$ then
        return $f$
    end
    for $n' \in children[p]$ do
        $f_c \leftarrow$ ModifiedPostOrderWalk($n'$)
        for $j \leftarrow 1$ to $n-1$ do
            $f[j] \leftarrow f[j] + f_c[j-1]$
        end
    end
    return $f$
end
```

4. *Give an algorithm for reconstructing a binary tree if we have the preorder and inorder walks.*

Solution: The key idea here is to note that we will recursively use both traversals in parallel. We will use the inorder traversal for finding the node index and iterate over the preorder to build the tree.

We assume that we have a function that can search for a node in the inorder traversal given the start and end indices of the search range.

To utilise both the traversals, we maintain a start and end index for the inorder traversal of the tree and make recursive calls while moving these indices and the iterator of the preorder traversal.

The tree build function can be written recursively as follows:

- if the start index is greater than the end index the tree is $NULL$
- otherwise we first create the root as the first element of the preorder
- if the start index and the end index are equal then the root is the answer
- otherwise we search for the index of the root in the inorder traversal
- the left sub-tree can be built recursively by setting the end index as the index previous to the root while the right sub-tree can be built after this by setting the start index as the index next to the root (note that the preorder traversal will be linearly iterated over by both these recursive calls but the iterator will be common across the calls)
- we return the root after setting the results of the sub-tree computations

Explanation: Note that if we know the position of the root in the inorder traversal, the left side of the inorder traversal is the inorder traversal of the left sub-tree and similarly for the right sub-tree. We also note that the first element in the preorder traversal is always the root, and the following elements will be the preorder traversal of the left sub-tree followed by the right sub-tree. The time complexity of this will be $\mathcal{O}(n^2)$ (think what is the worst case).

Correctness: Arguing for correctness is straightforward. Within each recursive call, the root is the element pointed to by the preorder iterator, which increments by one for each recursive call. After that, we find the root in the inorder traversal and then build the tree using the left side of the inorder traversal and the next set of the elements in the preorder traversal (both of these correspond to the left sub-tree). Then we build the tree using the right side of the inorder traversal and the remaining set of the elements in the preorder traversal (both correspond to the right sub-tree). Since the invariants (left-root-right for inorder and root-left-right for preorder) are maintained, the recursion terminates correctly.

Additional: What is the **minimal** set of changes required for building a tree using the postorder and inorder traversals?

5. *Let us suppose all internal nodes of a binary tree have two children. Give an algorithm for reconstructing the binary tree if we have the preorder and postorder walks.*

Solution:

Let preorder walk be $f = (f_1, f_2, \cdots, f_n)$ and postorder walk be $l = (l_1, l_2, \cdots, l_n)$
Base case
f and l have just 1 element let's say e, then the tree is nothing but a single-node tree with root as e.
Recursion step
Observations for tree with each internal node having two children and number of nodes greater than 1

1. In preorder walk, *first* element is *root* and *second* element is left child of root.

2. In postorder walk, *last* element is *root* and *second last* element is right child of root.

So, now with these observations in mind let's try to build the tree from f and l.
We have the root as $r = f_1 = l_n$
Left child of root as f_2 and right child of root as l_{n-1}
Now we will break f and l so as to get preorder and postorder walks for left and right subtree
Given f, which element in right subtree of r is first to be in the f?
By Observation 1, preorder on right subtree (which is a tree) of r should have it's root i.e. right child of r which is l_{n-1} first in the walk and so we have the break of f!!!
Find l_{n-1} in f, let us say f_k, then we have $f' = (f_2, f_3, \cdots, f_{k-1})$ as preorder walk of left subtree of r and $f'' = (f_k, f_{k+1}, \cdots, f_n)$ as preorder walk of right subtree of r
Similarly, given l, which element in left subtree of r is the last to be in the l?
By Observation 2, postorder on left subtree (which is a tree) of r should have it's root i.e. left child of r which is f_1 last in the walk and so we have the break of l!!!
Find f_2 in l, let us say l_k, then we have $l' = (l_1, l_2, \cdots, l_k)$ as postorder walk of left subtree of r and $l'' = (l_{k+1}, l_{k+2}, \cdots, l_{n-1})$ as postorder walk of right subtree of r

With f' and l', we **recursively** construct the left subtree and with f'' and l'', the right subtree. The time complexity will again be $\mathcal{O}(n^2)$.

6. (a) *Show that in order printing of BST nodes produces a sorted sequence of keys.*
   (b) *Give a sorting procedure using BST.*
   (c) *Give the complexity of the procedure.*

Solution:

(a) At any non-leaf BST node, the key at any node in the left subtree is not greater than the key at the node. Similarly, the key at any node in the right subtree is not less than the key at the node. An inorder traversal of a BST visits the left subtree first, then the node, and then the right subtree. Thus, the inorder traversal of the BST will produce a sorted sequence of keys.

(b)
- Create a BST with the first element of the unsorted array as the root and no children.
- Insert each of the remaining elements into the BST.
- Perform an inorder traversal of the BST to get the sorted sequence of keys.

(c) The complexity of the procedure is $\mathcal{O}(n \log n)$. This is because average insertion of n elements in a BST takes $\mathcal{O}(n \log n)$ time. And the inorder traversal takes $\mathcal{O}(n)$ time.

7. *Given a BST tree T and a value v, write a program to locate the leftmost and rightmost occurrence of the value v.*

Solution: Leftmost occurrence in a BST means leftmost occurrence in the inorder traversal of that BST.

Look at Algorithm 6 for the Pseudo-code of the leftmost occurrence in a BST.

The code itself is self-explanatory. Similarly you can right this for the rightmost occurrence.

Time complexity is $\mathcal{O}(h)$, h being the height of T, as we go through each level atmost once.

Algorithm 6: Finding the leftmost occurrence

```
Data: Tree $T$ having $n$ nodes
Function Leftmost($curr, v$):
    if $curr = NULL$ then
        return $NULL$
    end
    if $curr \to val \geq v$ then
        $left\_check \leftarrow Leftmost(curr \to left, v)$
        if $left\_check \neq NULL$ then
            return $left\_check$
        end
    end
    if $curr \to val = v$ then
        return $curr$
    end
    if $curr \to val \leq v$ then
        $right\_check \leftarrow Leftmost(curr \to right, v)$
        if $right\_check \neq NULL$ then
            return $right\_check$
        end
    end
    return $NULL$
end
// Call $Leftmost(root, v)$ in the main function
```

Tutorial 5

1. *Given a BST T and a key k, the task is to delete all keys $< k$ from T.*
   (a) *Write pseudocode to do this.*
   (b) *How much time does your algorithm take?*
   (c) *What is the structure of the tree left behind?*
   (d) *What is its root?*

Solution:

(a) The code (See Algorithm 7) provided deletes all keys $< k$ from T and returns the root of the modified BST.

(b) The complexity of the procedure is $\mathcal{O}(h)$, h being the height of T, as we are visiting each level atmost once (worst case being reaching a leaf node).

(c) The structure of the tree left behind is still a BST.

(d) If the previous root was greater than or equal to k, then the new root is the same as the previous root. Otherwise, the new root is different.

Algorithm 7: Deleting all keys $< k$

```
Function DeleteKeys($curr, k$):
    if $curr = NULL$ then
        return $NULL$
    end
    if $curr \to val \geq k$ then
        $(curr \to left) \leftarrow DeleteKeys((curr \to left), k)$
        return $curr$
    end
    else
        return $DeleteKeys((curr \to right), k)$
    end
end
// Call $DeleteKeys(root, k)$ in the main function
```

2. *Let $H(n)$ be the expected height of the tree obtained by inserting a random permutation of $[n]$. Write the recurrence relation for $H(n)$.*

Solution: The height of a binary tree equals one plus maximum of that of left and right subtrees. In a random permutation of $[n]$, any element (say k) can be the root of the subtree with probability $1/n$. The left subtree will contain $k-1$ nodes and the right one $n-k$.

Let X_n be the random variable representing the height of a n-node BST. Also, $H(n) = E[X_n]$. Then the recurrence relation is given as :-

$$\forall n \in \mathbb{N},\ H(n) = E[X_n] = 1 + \frac{1}{n}(\sum_{i=1}^{n} E[\max(X_{i-1}, X_{n-i})])$$
$$\leq 1 + \frac{1}{n}(\sum_{i=1}^{n} E[X_{i-1} + X_{n-i}])$$
$$= 1 + \frac{1}{n}(\sum_{i=1}^{n} E[X_{i-1}] + \sum_{i=1}^{n} E[X_{n-i}])$$
$$= 1 + \frac{2}{n}(\sum_{i=1}^{n-1} E[X_i]) = 1 + \frac{2}{n}(\sum_{i=1}^{n-1} H_i)$$

Base Case: $H(0) = 0$

Solving the recurrence gives $H(n)$ as $\mathcal{O}(\log n)$. You can refer this for a rough idea of its solution.

3. *Prove that after rotation the resulting tree is a binary search tree.*

Solution: See Figure 2. Suppose we go from left to right.

A becomes the left child of B and T_2 becomes the right child of A. $A \to val \leq B \to val$ as B was initially the right child of A (definition of BST) and $T_2 \to val \geq A \to val$ as T_2 was originally in the right subtree of A. T_1 is a BST, so is T_2, hence A's subtree is a BST too. Hence the new structure follows the rules of a BST.

Similarly we can prove for right to left in the figure.

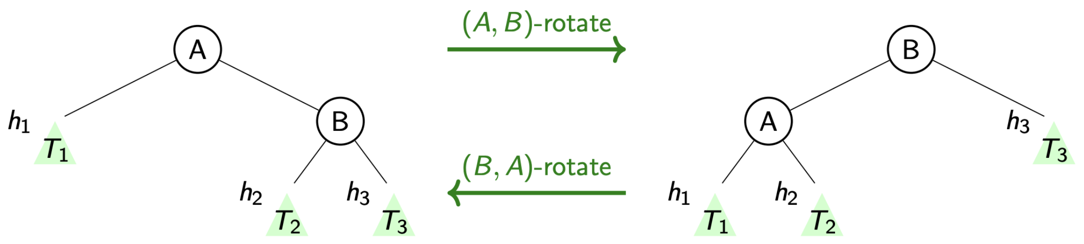


Figure 2: Rotation

4. *Insert sorted numbers 1,2,3,..., 10 to an empty red-black tree. Show all intermediate red-black trees.*

Solution: See Figure 3 below.

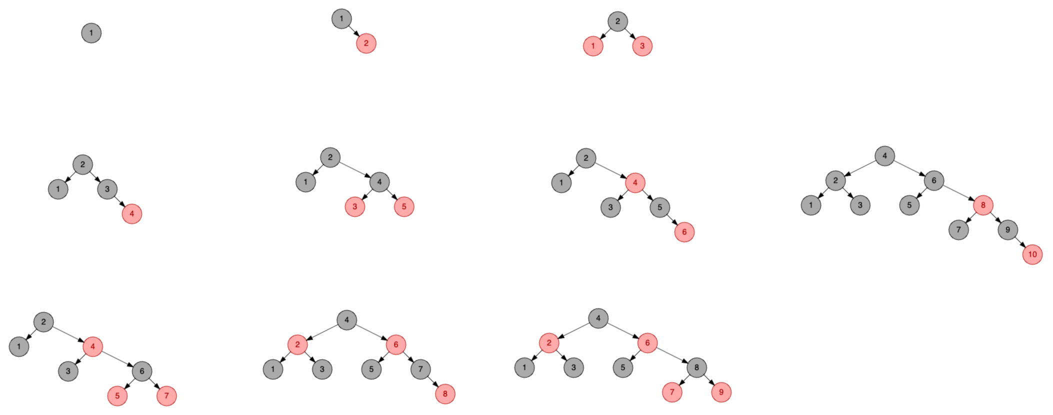
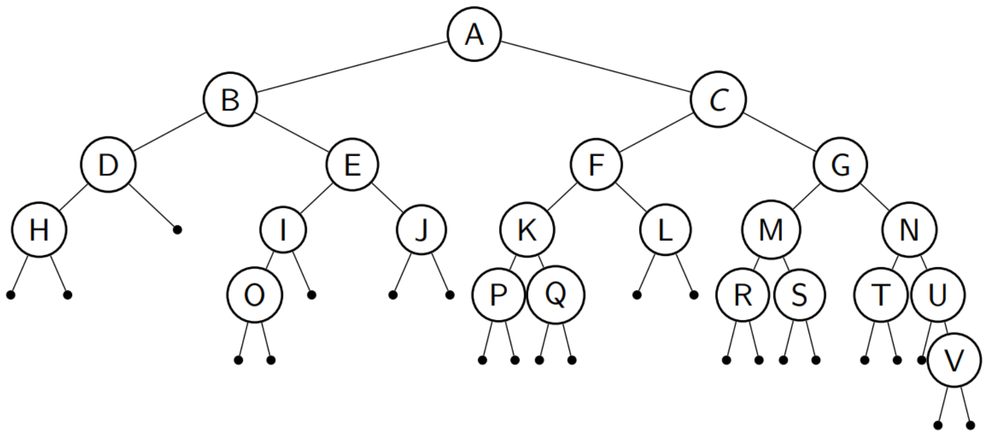


Figure 3: RBT Tree Insertions

5. *Consider the tree below. Can it be coloured and turned into a red-black tree? If we wish to store the set* $1, \ldots, 22$*, label each node with the correct number. Now add* 23 *to the set and then delete* 1*. Also, do the same in the reverse order. Are the answers the same? When will the answers be the same?*



Solution: If we color this RB tree, Black height of this tree must be 3. Why?? (Hint: Argue in terms of property of RB tree) Colour the nodes H, E, O, C, N, V, K and M red; the rest should be black. This is a red-black tree with a black height of 3. Using the inorder traversal of the tree,
$A = 8, B = 3, C = 14, D = 2, E = 6, F = 12, G = 18, H = 1, I = 5, J = 7, K = 10$
$L = 13, M = 16, N = 20, O = 4, P = 9, Q = 11, R = 15, S = 17, T = 19, U = 21, V = 22$

Insertion Followed by Deletion
The insertion is very straightforward as 23 gets added as the right child of V = 22, let's call it W. Red black tree condition gets violated as V and W are of same color now. So, according to case 3, apply a left rotation such that after rotation N.right=V, V.left=U and V.right = W. H=1 is a red color leaf, so we can simply delete it without any trouble of black height violation.

Deletion Followed by Insertion
In this case, our answer will be the same as the one above. Since, deletion of H doesn't change anything significant for the right tree and we can simply insert 23 as described above.
Difference can be in cases where there is upward propagation of violation while dealing

with violation after insertion and deletion. Since, upward propagation checks and affect different parts and colors of the tree

6. *(a) Give running time complexities of delete, insert, and search in red-black tree.*

   (b) What are the advantages of red-black tree as compare to Hash table, where every thing is constant time?

Solution: (a) $\mathcal{O}(\log n)$, n being the number of nodes of RBT. (Refer slides to know why).

(b) These are the following advantages:-

- Red-black trees maintain elements in a sorted order, which is useful for operations that require order (e.g., finding the minimum/maximum, range queries). Hash tables, on the other hand, do not maintain any order among elements.
- The performance of a red-black tree is predictable and does not depend some factor like hash table does on the quality of the hash function or the distribution of data.
- RBT has no collisions like a Hash Table.

7. *Suppose we use a hash function h to hash n distinct keys into an array T of length m. Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{k, l\} : k \neq l$ and $h(k) = h(l)\}$?*

Solution: Let's define a binary random variable X_{ij}:-

$$X_{ij} = \begin{cases} 1 & \text{if there is a collision between the } i^{th} \text{ and } j^{th} \text{ keys during insertion,} \\ 0 & \text{otherwise.} \end{cases}$$

$E[X_{ij}] = 1/m$ as the probability that the key that is inserted later gets the same slot as the one inserted earlier is $1/m$. Our hashing function is **uniform**, i.e. probability of getting any slot is equal and hence is 1/(number of slots).

Let X be the random variable returning the number of collisions during insertion of the n keys, then,

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

This is because the RHS represents the sum over all possible distinct unordered pairs $\{\{k, l\}, k \neq l\}$ seeing if there is a collision within each pair or not. Hence,

$$\begin{aligned} E[X] &= E[\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}] \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[X_{ij}] \text{ (Due to linearity of the Expectation operator)} \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{m} \text{ (Explained above)} \\ &= \frac{\binom{n}{2}}{m} = \frac{n(n-1)}{2m} \text{ (There are } \binom{n}{2} \text{ total unordered pairs)} \end{aligned}$$

Hence the expected number of collisions is $\frac{n(n-1)}{2m}$.

Tutorial 6

1. *Give an implementation for the scheduling problem using Heap.*

Solution: Refer the slides to know about the scheduling problem.

See Algorithm 8 for the Pseudo-code.

Use of the Heap (Min Heap here) is to select the minimum remaining processing time among the processes present at any time. Before the timestamp of any new process is added, we will keep performing the jobs (available at that moment) with the minimum remaining times. As soon as a new process is added, the current process (if any) is preempted (i.e. interrupted) and the algorithm now looks for the process with the minimum remaining time. In the end, the for loop exits only when all the jobs are completed.

Given solution will have a time complexity of $\mathcal{O}(n \log n)$. Why? Think. (Hint: Number of operations (push or pop) won't exceed $4n$.)

Algorithm 8: Shortest Remaining Time Scheduling

```
Data: Array (sorted) $time$ containing the timestamp the $i^{th}$ job is called and a
      corresponding array $duration$ containing the duration of the $i^{th}$ job. Both are of
      size $n$.
Function Schedule($n$, $time$, $duration$):
    MinHeap $RemTime$ //A Min Heap for the remaining processing times
    for $i \leftarrow 0$ to $n-1$ do
        $RemTime.push((duration[i], i))$ //Sorted by the first of the pair
        $CurrTime \leftarrow time[i]$
        if $i = n-1$ then
            $EndTime \leftarrow \infty$ //Some large value
        end
        else
            $EndTime \leftarrow time[i+1]$
        end
        while not $RemTime.empty()$ do
            $MinRemTime, CurrIndex \leftarrow RemTime.top()$
            $RemTime.pop()$
            $NextCurrTime \leftarrow CurrTime + MinRemTime$
            if $NextCurrTime > EndTime$ then
                $print$("Job", $CurrIndex$, "starts at", $CurrTime$, "and ends at", $EndTime$)
                $duration[CurrIndex] \leftarrow duration[CurrIndex] - (EndTime - CurrTime)$
                $RemTime.push((duration[CurrIndex], CurrIndex))$
                break
            end
            $print$("Job", $CurrIndex$, "starts at", $CurrTime$, "and ends at", $NextCurrTime$)
            $print$("Job", $CurrIndex$, "will be completed.")
            $CurrTime \leftarrow NextCurrTime$
        end
    end
end
```

2. *Can a Priority Queue be implemented as a red-black tree? What advantages does a heap implementation have over a red-black tree implementation?*

Solution: Yes, the priority queue can be implemented using red-black trees. Priority queue has 3 operations which RB trees can also do:

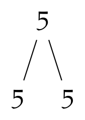
- insert: Insert can be done in an RB tree using the priority as the property of the tree
- top: Return the pointer to the rightmost element of the RB tree
- deleteMax: delete the rightmost element of the RB tree

Time complexity can be made equal for all of the above operations. (top can work in $\mathcal{O}(1)$ time. Think.)

The advantage of heap implementation is you can store the priority queue in an array, in which you aren't storing anything extra like child pointer, parent pointer etc, you are just storing the data of the node. Also, array implementation increases the cache performance.

3. *Give a tree, if exists, that is a binary search tree, is a heap, and has more than two nodes. If such a tree does not exist, give a reason.*

Solution: Consider the given binary tree:-



Since all keys are equal (and last level is complete hence no worry about being left-filled), this tree satisfies the definitions of both BST and Heap (Refer the slides for the definitions).

4. *Suppose we have a* 2*D array where we maintain the following conditions: for every* (i, j), *we have* $A(i, j) \leq A(i+1, j)$ *and* $A(i, j) \leq A(i, j+1)$. *Can this be used to implement a priority queue?*

Solution: Yes, it is very similar to that of a heap. There are some modifications to the heap implementation:

- Store the negative of the priority in this data structure
- $A[i, j]$ has two children $A[i, j+1]$ and $A[i+1, j]$
- $A[i, j]$ has two parent $A[i, j-1]$ and $A[i-1, j]$
- Level of $A[i, j]$ is denoted by $i + j$
- You can again define notation of level-wise left filled.
- Deletion is done similar to heap
- Insertion is done similar to heap with a modification that the max of both parent is considered for swap.
- $A[0, 0]$ is the max-priority element.

5. *Given an unsorted array, find the kth smallest element using a priority queue.*

Solution: Here are the steps:-

1. Push all the elements of the array in the priority queue (Min Heap one).
2. Pop k times. Answer is the k^{th} pop.

Follow up: Can you do this in $\mathcal{O}(n)$ time (Not necessarily using Heaps)?

6. *Given two heaps give an efficient algorithm to merge the heaps.*

Solution: Here are the steps:-

1. Concatenate the array of the heaps (or copy all the elements from both arrays to a new one).
2. Apply $\mathrm{BuildHeap}$ function on this new array.

Time Complexity of this algorithm is $\mathcal{O}(m+n)$, m and n being the size of the heaps. This is because both the steps above are linear in time.

Tutorial 7

1. *Compute KMP-table array h for pattern "babbaabba".*

Solution: The algorithm at the end is given. In the beginning, index i is 1, and index j is 0. Note that the length m is 9 and the array h looks like this:

-1									

Now, $pat[1]$ is 'a' while $pat[0]$ is 'b'. Since the two are not equal, we assign $h[1]$ to 0 (j).

-1	0								

The while loop sets $j = h[j] = -1$. Then, both get incremented so that $i = 2$ and $j = 0$. Now, $pat[2] = pat[0]$ so we assign $h[2] = h[0] = -1$.

-1	0	-1							

Now, after incrementing and in the next iteration, $pat[3] \neq pat[1]$ so $h[3]$ is assigned to 1.

-1	0	-1	1						

One iteration of the (inner) while loop assigns j to $h[j]$ which is 0. Then, $pat[3] = pat[0]$ so it stops there. After incrementing both variables, we see that $pat[4] = pat[1]$ so we assign $h[4]$ to $h[1]$ which is 0.

-1	0	-1	1	0					

On incrementing both variables, $pat[5] \neq pat[2]$ so we assign $h[5]$ to 2.

-1	0	-1	1	0	2				

One iteration of the while loop assigns j to $h[j]$ which is -1. After incrementing, we get $i = 6$ and $j = 0$. Now, $pat[6] = pat[0]$ so $h[6]$ is assigned to $h[0]$ which is -1.

-1	0	-1	1	0	2	-1			

Next iteration, $pat[7] \neq pat[1]$ so $h[7]$ is assigned to 1.

-1	0	-1	1	0	2	-1	1		

The while loop resets j to -1, following which j is incremented to 0 and i to 8. Now, $pat[8] \neq pat[0]$ so $h[8]$ gets assigned to zero.

-1	0	-1	1	0	2	-1	1	0	

At last, $j = 0$ and $i = 9$. Now, since $i = m$ we break the standard loop and assign $h[i]$ to j, which is zero. At the end, the KMP table looks like this:

-1	0	-1	1	0	2	-1	1	0	0

Algorithm 9: KMPtable

```
Function KMPtable(String pat):
    $i \leftarrow 1$;
    $j \leftarrow 0$;
    $m \leftarrow pat.length()$;
    $h \leftarrow new\ int\ [n+1]$;
    $h[0] \leftarrow -1$;
    while $i < m$ do
        if $pat[i] \neq pat[j]$ then
            $h[i] \leftarrow j$;
            while $j \geq 0$ and $pat[i] \neq pat[j]$ do
                $j \leftarrow h[j]$;
            end
        end
        else
            $h[i] \leftarrow h[j]$;
        end
        $i \leftarrow i + 1$;
        $j \leftarrow j + 1$;
    end
    $h[m] \leftarrow j$;
    return $h$;
end
```

2. *Is this version of the KMPtable algorithm correct?*

Algorithm 10: KMPtableDiff

```
Function KMPtableDiff(String pat):
    $i \leftarrow 1$;
    $j \leftarrow 0$;
    $m \leftarrow pat.length()$;
    $h \leftarrow new\ int\ [n+1]$;
    $h[0] \leftarrow -1$;
    while $i < m$ do
        $h[i] \leftarrow j$;
        while $j \geq 0$ and $pat[i] \neq pat[j]$ do
            $j \leftarrow h[j]$;
        end
        $i \leftarrow i + 1$;
        $j \leftarrow j + 1$;
    end
    $h[m] \leftarrow j$;
    return $h$;
end
```

Solution: The answer is **no**. This algorithm is incorrect. Whenever $pat[i] = pat[j]$, the wrong value is inserted into the KMP table (meaning the table is incorrect). This was discussed in slides that if $h[i]$ is assigned to j, it does not meet the requirement that when the shifted string *pat* is aligned with *pat*, the compared indices i and j differ. If they do not, then the main search algorithm gets more inefficient.

3. *Suppose that there is a letter z in pat of length n such that it occurs in only one place, say k, which is given in advance. Can we optimize the computation of h?*

Solution: Let the character z appear in position $m < n$ and nowhere else. Recall the function of the table is to find the *largest index j given i such that* $pat[0:j] = pat[i-j:i]$ *but* $pat[j] \neq pat[i]$. By definition, that j has to be less than i, else it implies a shift of the search string by 0 in the main search (since in case of inequality between $txt[i]$ and $pat[j]$, there, we move j to $h[j]$ and large $h[j]$ means small shift forward; $j = h[j]$ means no shift).

If $j > m$, it means that $pat[0:j]$ contains z. For the equality to hold, $pat[i-j:i]$ should contain z at index m, meaning that $pat[j+m] = z$, which goes against the premise of our question: there is only one index where z is located. So, we say that j in the computation can never exceed m or, more specifically, no entry in the KMP table can exceed m. Further, $pat[i-j:i]$ cannot contain z or index m in its range.

The computation of the KMP table till index m is the same as the earlier algorithm, and $h[m]$ will get assigned to the correct value anyway. For higher values, we split *pat* into $subpat = pat[0:m]$ and $src = pat[m+1:n]$. In order to compute the value of $h[i]$ for $i > m$, we try to follow the KMP search algorithm with *src* as the text and *subpat* as its pattern. Except, we do not look for exact matches. Rather, when $src[i] \neq subpat[j]$, we assign $h[i+m+1]$ to j, as expected.

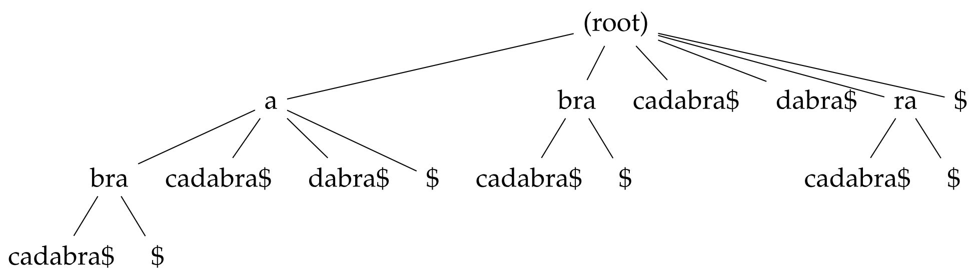
The case when $src[i] = subpat[j]$ seems a bit complicated, but just like the KMP table computation, doing $h[i+m+1] \leftarrow h[j]$ will work, since $src[i] = subpat[h[j]] \neq subpat[j]$ and all previous indices are equal. When $src[i] \neq subpat[j]$, after assigning $h[i+m+1] \leftarrow j$, we keep moving j much like KMPsearch and KMPtable algorithms.

This begs the question: is it really more efficient? To begin with, we cannot achieve less than $O(n)$ since we need to parse the entire string at least once (the output depends on all characters) and generate an $O(n)$ output. But can we make it faster *O(n)*? Such as reducing the constant c rather than focusing on asymptotic complexity?

One idea is to multi-thread the two computations, the KMP-table of *subpat* and the modified KMP-search of *subpat* in *src* that just updates the same table (algorithm not shown, since it gets cluttered and tedious with managing threads, and guaranteeing that i in the *subpat* thread is not less than j in the KMPsearch – this requires locks and semaphores knowledge, and can still be as 'inefficient' which is $O(n)$). Any single-threaded solution (such as doing the 2 procedures in sequence or in one master function with somewhat on-demand computation of $h[j]$ for j in *subpat*) necessarily takes as much time as the normal KMPtable-filling algorithm.

4. *Compute the suffix tree for 'abracadabra\$'. Compress degree 1 nodes. Use substrings as edge labels. Put a square around nodes where a word ends. Use it to locate the occurrences of 'abr'*

Solution: This is the required suffix tree (dollar (\$) – where a word ends):



To locate ‘abr’, we traverse the suffix tree to node for ‘a’. Then we look at the child corresponding to ‘br’. It is the left child. We see that there are two suffixes: ‘abra’ (just that, ending in $ symbol node) and ‘abracadabra’, which is the whole word itself.

5. *Review the argument that for a given text T, consisting of k words, the ordinary trie occupies space, which is a constant multiple of $\|T\|$. How is it that the suffix tree for a text T is of size $O(\|T\|^2)$? Give a worst-case example.*

Solution: Yes, for a given text, the trie occupies space a constant multiple of $n = \|T\|$, since each character has almost one node associated with it, and the space taken by each node in the trie is constant. Even if the alphabet is infinite, a different representation of trie (linked list for children) implies that each trie-node has at max one parent, and therefore, the space taken is directly proportional to the number of nodes.

On the other hand, the suffix tree stores **all** suffixes. Consider the text T having only one word: each suffix (corresponding to a character in the word, of the n) takes up $O(size\ of\ suffix)$ space. The net space evaluates to $O(n^2)$ since suffixes are substrings of length $1, 2, \ldots n$ where n is the length of one word.

Sure, there may be patterns that are repeated (and therefore, the number of nodes is fewer), and the best case is $O(n)$ for strings like $aaaaaa \ldots aa$. However, the worst case is $O(n^2)$ when all characters are distinct (particularly when there is no restriction on character space), and the text comprises one word.

Tutorial 8

1. *In an Huffman code instance, show that if there is a character with frequency greater than* $2/5$*, then there is a codeword of length 1. Further, show that if all frequencies are less than* $1/3$ *then there is no codeword of length 1.*

Solution: According to Huffman compression, the two trees that have the least net frequency become the left and right child of a new node, forming a new tree. This process is repeated till there is only one tree left. Recall that in Huffman compression, a tree can have one node (root and leaf) that represents one character, or it can be any binary tree, such that each leaf represents a character and the net frequency of the tree is the sum of frequencies of occurences of all the leaves (characters) in the text.

1. There exists a frequency more than $2/5$, then there exists a codeword of length 1.

If that character c_0 is not the most frequent character, but it has $f_{c_0} > 2/5$, then there exist two or more characters with frequencies $2/5$ or higher. Let the second be c_1 with $f_{c_1} \geq f_{c_0} > 2/5$. If this is the case, either there are just c_0 and c_1 (they are encoded 0 or 1), or there are more than these two characters, but all the rest have net frequency $1 - f_{c_0} - f_{c_1} < 1/5$. All of those get clubbed together to form one tree, since, until this is the case, the two trees with the least frequency cannot include either of c_0 and c_1.

After this happens, the 'less frequent' tree T gets clubbed with c_0 first, then the resultant gets clubbed with c_1 [Clubbed = they become children of a new node, forming one tree]. At the end, the c_1 is either the left or right child of the root of the final Huffman tree, and has a codeword of length 1 (codewords '0' or '1')

If c_0 is the most frequent character and we expect it to not have a codeword of length 1, it should be clubbed with another tree sometime before the last step, when there are atleast two trees, both of which do not contain c_0. Further, all trees except for $\{c_0\}$ and one other tree (that gets clubbed with c_0 whenever it gets clubbed), should have a net frequency greater than or equal to $f_{c_0} > 2/5$. This is possible only when there is one such tree, and there are three trees in all.

Let us examine that case: let T_0 be the tree with highest frequency and T_1 the low frequency tree, that gets clubbed with c_0. Clearly, $f_{T_0} > 2/5$ from this and the frequency of c_0, we say that $f_{T_1} < 1/5$. Since T_0 is not a singleton tree, we look at the previous step wherever T_0 was formed from two children, T_2 and T_3. Without loss of generality, let $f_{T_2} \geq f_{T_3}$.

Then, $2 \cdot f_{T_2} \geq f_{T_2} + f_{T_3} = f_{T_0} > 2/5$ so, before its combination, $f_{T_2} > 1/5 > f_{T_1}$; there existed atleast two trees T_3 and T_1 (more if T_1 was not formed yet), both with net frequency less that that of T_2. This means that if T_2 was clubbed (it had to be for our case to hold), its combination with T_2 did not follow the Huffman algorithm. This arises a contradiction.

Therefore, if there exists a character of frequency more than $2/5$, then there has to be a character with codeword of length 1.

2. If all frequencies are less than $1/3$, there exists no codeword of length 1.

Let there be a character c_0 that has the luxury of a codeword of length 1. If this were so, at the last iteration of Huffman, we have a singleton tree c_0 (else, it would not have a codeword of length 1), and a tree T_0. Since $f_{c_0} < 1/3$, $f_{T_0} > 2/3$.

Since this tree T_0 has to contain two or more characters, it must have two subtrees T_2 and T_3, and those must have been clubbed together in the last step (the only possibility). Without loss of generality, let $f_{T_2} \geq f_{T_3}$. This implies that $2 \cdot f_{T_2} geq f_{T_2} + f_{T_3} = f_{T_0} > 2/3$

or $f_{T_2} > 1/3 > f_{c_0}$. Then, the two trees with least net frequencies are T_3 and $\{c_0\}$. Those would have been clubbed together, arising in a contradiction.

Therefore, we conclude that there cannot exist a codeword of length 1 if all characters have frequency less than 1/3.

2. *Suppose that there is a source that has three characters 'a', 'b', 'c'. The output of the source cycles in the order of 'a', 'b', 'c' followed by 'a' again, and so on. In other words, if the last output was a 'b', then the next output will either be a 'b' or a 'c'. Each letter is equally probable. Is the Huffman code the best possible encoding? Are there any other possibilities? What would be the pros and cons of this?*

Solution: Yes, we can use a location-aware encoding i.e. one wherein the encoding of a character depends on its surroundings, specifically, the character before it. This is a better encoding scheme than Huffman encoding.

For instance, we choose to encode a character with a '1', assuming that it is different from the previous character (there can only be one possible value corresponding to this 1), and a '0' assuming it is same. We can encode the first character using any Huffman tree, say, 1 for 'a', 00 for 'b' and 01 for 'c'. Then, abccaaabbca will be encoded as 11101001011. Also, 010011010110 would correspond to cccabbccabb.

Roughly, this corresponds to one bit per character. Compare that with using on an average 5/3 bits if we encoded each character individually (and they had roughly same frequency each; true since each letter is equally probable), which would be the case with Huffman encoding. This clearly means that our encoding is better, the big pro. Also, ours does not take a very long time to evaluate (definitely not more than Huffman). Another pro (plus point) is that this is very much scalable to m character systems, say the English alphabet, except, when we know that an 'a' is followed by 'a' or 'b'; 'b' by 'b' or 'c'; ... and 'z' by 'z' or 'a'.

Speaking of cons (downsides) of our encoding, this is not generalizable to arbitrary strings. If 'ac', 'cb' or 'ba' occured as a substring anywhere (very likely), then our encoding fails (there does not exist an encoding by our scheme). Granted, we know that this is not the case for our source; but we first need to verify that the source is of this form (this can easily be done) so this is not really a con.

Another improvement would be to use something like run-length-encoding algorithm, particularly efficient for strings with long runs like aaaaaaaaabbbbbbbbbbbbcccccccc; but those would be inefficient if there are lot of short runs, particularly, if lot of runs have length 1.

3. *Given the following frequencies, compute the Huffman code tree.*

Character	a	d	g	j	b	e	h	k	c	f	i	l
Frequency	20	7	8	4	6	25	8	2	6	1	12	1

Solution:

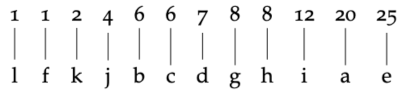


Figure 4: The initial forest for Huffman algorithm

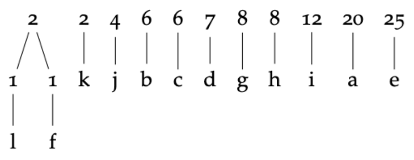


Figure 5: Combining least frequency trees (just 'l' and 'f')

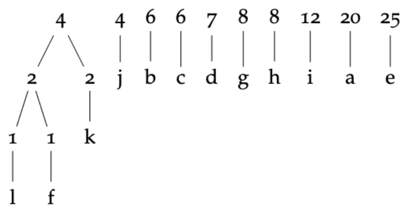


Figure 6: Next iteration, combining trees with least frequency (2 and 2)

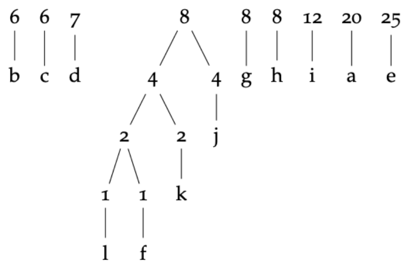
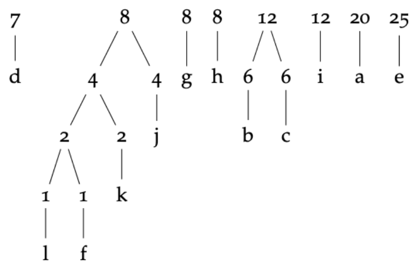
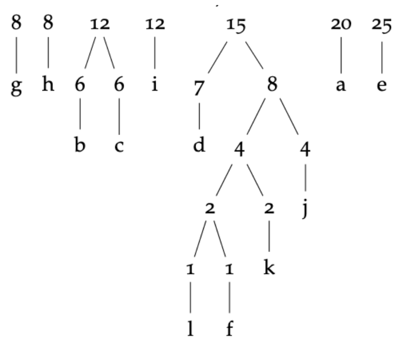
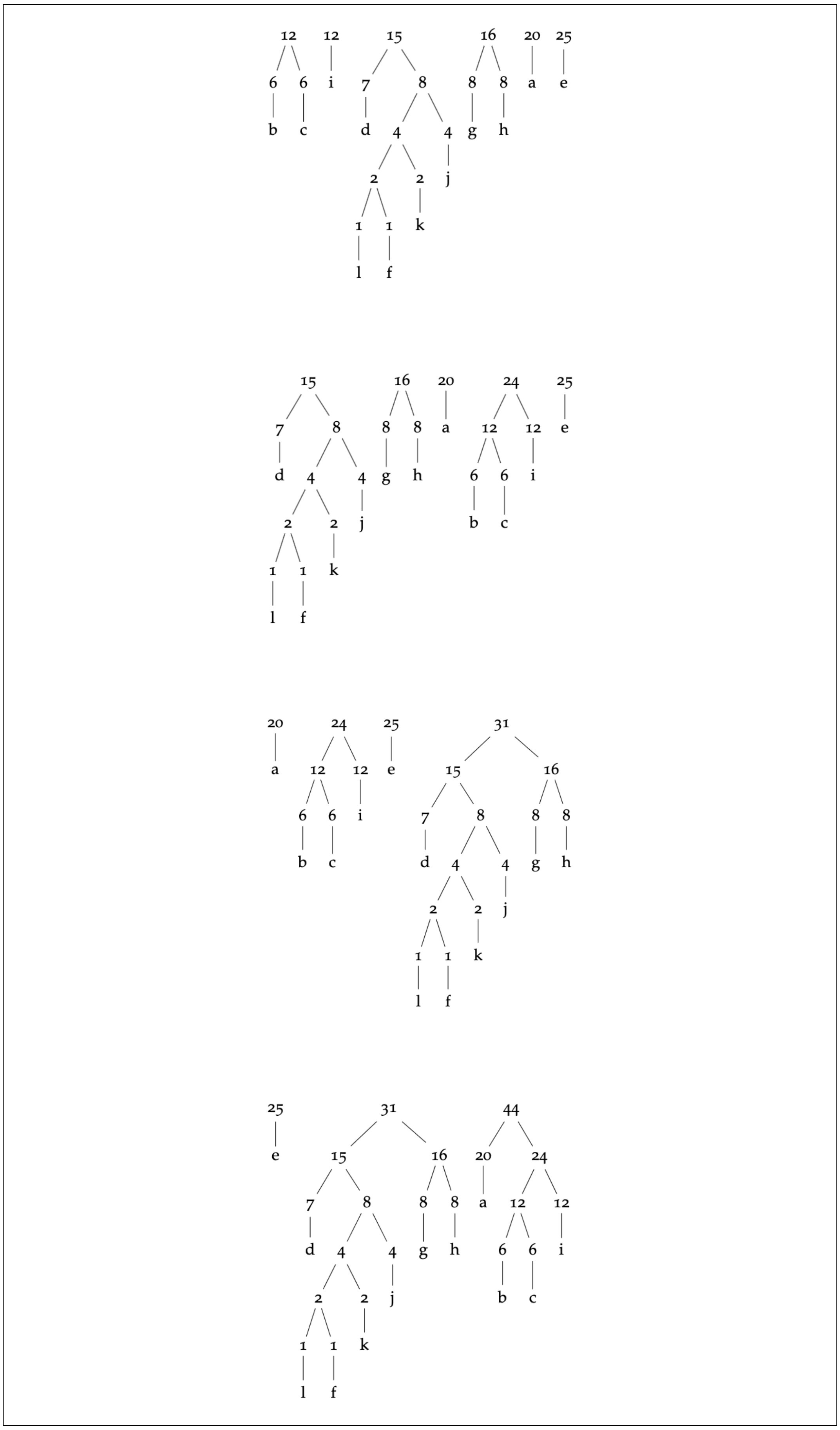


Figure 7: We keep iterating the process as shown below.







12 12 15 16 20 25
6 6 i 7 8 8 8 a e
b c d 4 4 g h
2 2 j
1 1 k
l f
15 16 20 24 25
7 8 8 8 a 12 12 e
d 4 4 g h 6 6 i
2 2 j b c
1 1 k
l f
20 24 25 31
a 12 12 e 15 16
6 6 i 7 8 8 8
b c d 4 4 g h
2 2 j
1 1 k
l f
25 31 44
e 15 16 20 24
7 8 8 8 a 12 12
d 4 4 g h 6 6 i
2 2 j b c
1 1 k
l f

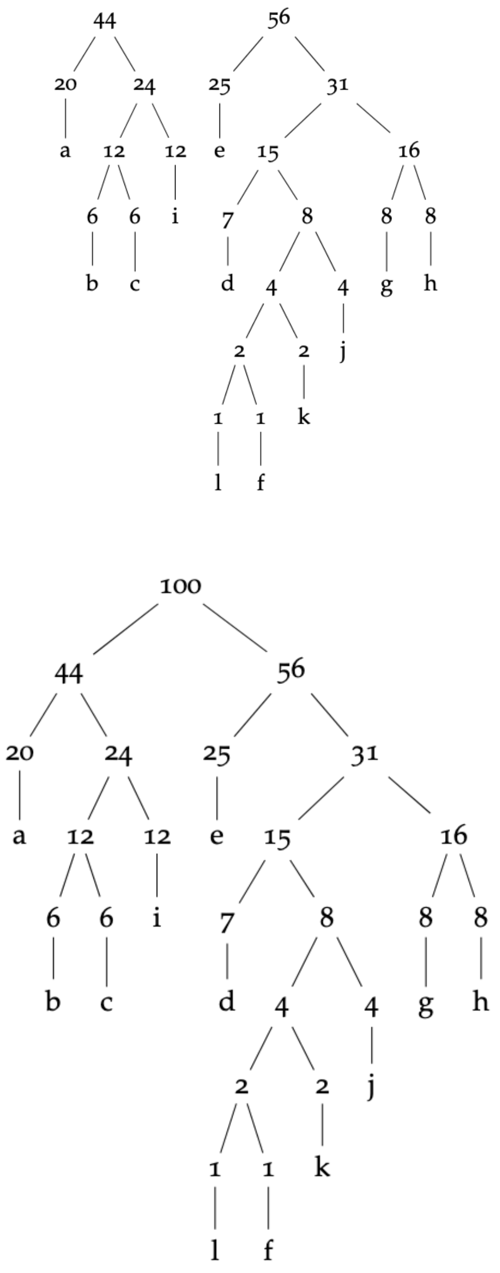


Figure 8: This is the complete Huffman tree.

4. *We have seen a "proof of optimality" of the Huffman Coding. Is Huffman Coding really the best lossless text compression method out there? If no, give an example where Huffman Coding costs more than some other coding. Then is Huffman "optimal"? What "optimality" did we talk back there? Can you think why is Huffman not the best then?*

Solution: No, Huffman Coding is NOT the best text compression method we have.

There will be 2 cases:- 1) We build a different Huffman tree for every string, 2) We use some universal Huffman tree for all strings.

In case 1, we need to use some bits to store the Huffman tree made every time for an input string. Take "Today_is_Monday" for example. Original string takes $15 \times 8 = 120$

bits (Assuming 8 bits for each ASCII), but Huffman takes *atleast* $49 + 10 \times 8 = 129$ bits (49 for the Huffman string created (You can work this out on your own) and the table (which maps each character of the word to a Huffman code) will contain 10 entries as 10 distinct characters (Also assumed that each entry takes minimum 8 bits)).

In case 2, if there is some pre-processed Huffman table, then take "xxxx". Since 'x' is the least used English alphabet, it's Huffman code will likely take more than 8 bits.

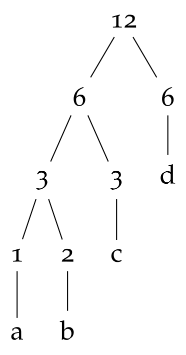
Huffman is "optimal" only when we consider bits used to represent the Huffman string. The proof done in class doesn't cover the bits that maybe required for making the Huffman table.

Another reason why Huffman isn't the best is that it assigns *integer*-number of bits to any symbol. Huffman codes are constrained to bit boundaries. In a Shannon information theoretic sense, if we just look at context-free probabilities of symbols, the ideal bit-length of a symbol is $-\log_2(P(x))$ where $P(x)$ is the probability of occurrence of the symbol x.

That number is unlikely to be an integer! The Huffman code's inefficiency lies in the distance between the information theoretic optimal bit length and the actual integer bit length the Huffman code assigned.

5. *What is the largest Huffman code for any character that can be created in a word? Give an example.*

Solution: Largest Huffman code for any character can be one less than number of distinct characters in the word. Consider "abbcccdddddd":-



See that code length for 'a' or 'b' is 3.

6. *You are given a stick of length l. Your job is to break the stick to create pieces of lengths $l_1, l_2, l_3, \ldots, l_m$ (These sum up to l). But the cost of breaking any stick (at any point) into two parts is equal to the length of that stick. What will be the minimum cost to do your job?*

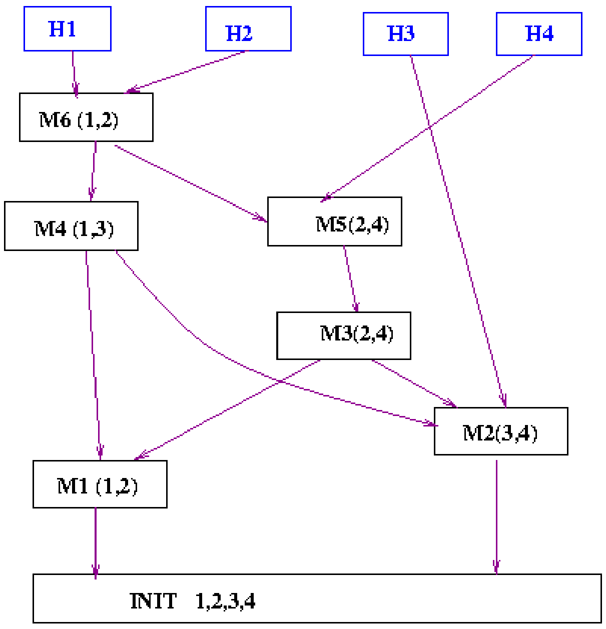
Solution: The solution is pretty similar to that of Huffman Coding. We build a tree just like the Huffman tree, just here the weights of the nodes is the length of the sticks instead of frequency of character. Proof of optimality follows from that of Huffman's.

Tutorial 9

1. *The graph is an extremely useful modelling tool. Here is how a Covid tracing tool might work. Let V be the set of all persons. We say (p,q) is an edge (i) in $E1$ if their names appear on the same webpage and (ii) in $E2$ if they have been together in a common location for more than 20 minutes. What significance do the connected components in these graphs, and what does the BFS do? Does the second graph have epidemiological significance? If so, what? If not, how would you improve the graph structure to get a sharper epidemiological meaning?*

Solution: The first graph, with edges E1, gives us a measure of the closeness of two people with each other. This may be interpreted as a measure of how frequently they come in contact with each other, and clearly, the more often they come in contact, the more likely the virus is to spread from one of them to the other. Thus, we can use this as a kind of contact tracing tool; BFS gives us the shortest path between any two people, and the shorter this path is, the more likely it is that the virus has spread from one end to the other.

For every infected person, we can do a BFS over the graph, and all nodes (people) having small shortest distances from that infected person can be classified as having high risk. Any connected component thus consists of a number of people who have high chances of spreading the virus among themselves if any one gets infected. This can be used to determine the risk faced by every individual in the system, due to a particular person being infected.



The second graph, with edges $E2$, can similarly be used to identify high-risk individuals since any infected person is likely to have spread the infection to their neighbour(s) in the $E2$ graph, in the 20-minute interval where they were close together. However, this information is not sufficient for points separated by more than one edge because the graph does not contain any information about the chronology of the 20-minute interactions. For example, for two individuals separated by three edges, the virus would only spread from one end to the other if the 20-minute interactions occurred in the right order. So, we can improve this data structure by additionally maintaining a timestamp on each edge, indicating the time at which the 20-minute interaction took place.The vertices are $V = \{[M_i, S_i] | \text{i-th meeting label}, S_i = \text{people who participated}\}$. There is an edge from $[M_i, S_i]$ to $[M_j, S_j]$ if and only if

- there is a person p common to S_i and S_j
- $i < j$
- p did not participate in any meeting between i and j.

A natural question follows. How is one to generate such a graph?

Let header(p) point to the last meeting where p participated. Suppose a new meeting M_k, S_k happens with $S_k = \{p, q, r\}$ then the code is

Algorithm 11: Insert Function

```
Function Insert($k, S_k$):
    New meeting $m$;
    $m.S \leftarrow S_k$;
    $m.id \leftarrow k$;
    $m$.next $\leftarrow$ [] ;                                  // Adjacency list
    foreach $p$ in $S_k$ do
        $m$.next $\leftarrow$ append($m$.next, header($p$));
        header($p$) $\leftarrow m$;
    end
end
```

2. *Show that a bipartite graph does not contain cycles of odd length.*

Solution: We will prove this by contradiction, let us assume there is a odd length cycle. If a graph is bipartite, we can divide the set of vertices into two sets V_1 and V_2 such that there is no edge among nodes in V_1 or among nodes in V_2. Now, let's say we color all V_1 vertices by red color and all V_2 vertices by blue color. Now, there can't be any edge between two vertices of same color otherwise the graph wasn't bipartite. This implies that nodes in the cycles should be alternatively colored red and blue. Now, relate every blue node in the cycle to a red node connected to the blue node by a edge in clockwise direction. This proves that their are equal no of blue and red nodes in the cycle. This contradicts our assumption, that the cycle is of odd length. Hence, a bipartite graph does not contain cycles of odd length.

Follow up: A graph with no odd cycles is bipartite. Is this correct? If yes, prove. If not, give a counterexample.

3. *Let us take a plane paper and draw circles and infinite lines to divide the plane into various pieces. There is an edge (p, q) between two pieces if they share a common boundary of intersection (which is more than a point). Is this graph bipartite? Under what conditions is it bipartite?*

Solution: Yes, the graph is bipartite. We'll introduce a colouring scheme for the regions created by the shapes. Follow these steps:-

1. Colour the entire plane red (or blue as you like) and create a list of **unique** line(s)/circle(s) and insert them one-by-one.

2. If there's a circle/line to be inserted, then insert it in the plane, else exit this procedure.
3. Insertion will divide some of the previously existing regions into more regions. Now flip ALL the colours present inside the circle/to the left of the line (red $\rightarrow$ blue, blue $\rightarrow$ red).
4. Goto Step 2.

Notice that this scheme ensures that no two regions that share a boundary will have the same colour. Why so? Let's assume the contrary. Everything was well till I inserted $i-1$ line(s)/circles(s) but as soon as I insert the i^{th} line/circle, I find two red (WLOG) regions that share a boundary (line/arc between two vertices (vertices include infinity too) (say B). There will be two cases:-

- B is a part of the newly inserted shape. But this case shouldn't happen as, before the flipping, both sides of B should've the same colour and hence after flipping B would've different colours across it. Now you may think why should both sides of B be of same colour before flipping, because for it to be different, B should overlap on an already existing boundary that had different colours on its sides but boundaries overlapping would only mean that the newly inserted line/circle is exactly same as one of the previously inserted ones, but I've removed that possibility (see step 1).
- B is not a part of the newly inserted shape and hence of some already existing boundary. But this shouldn't happen as otherwise something should've been noticed in the previous $i-1$ steps.

Hence such a colouring scheme that prevents two same colours from having an edge (boundary) will make the graph bipartite.

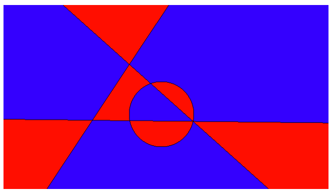


Figure 9: Initial

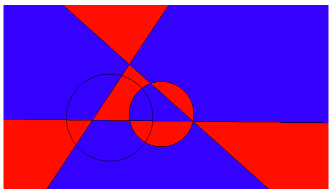


Figure 10: Insertion

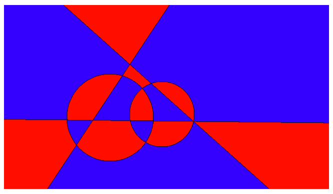
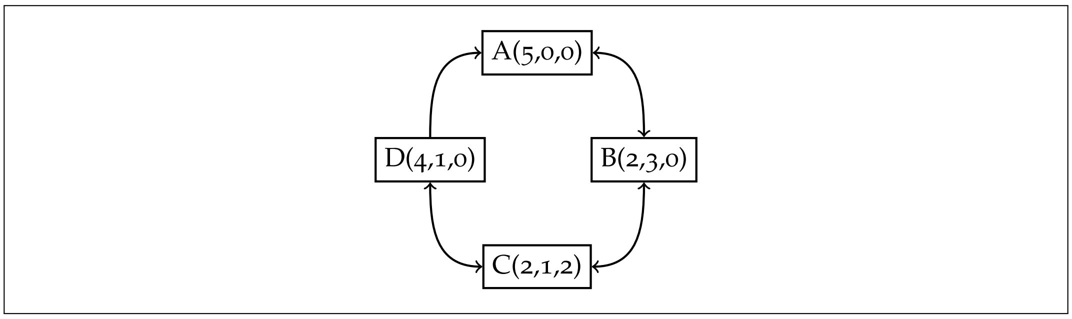


Figure 11: Flipping

4. *There are three containers, A, B, and C, with capacities of* 5, 3, *and* 2 *liters respectively. We begin with A has* 5 *liters of milk, and B and C are empty. There are no other measuring instruments. A buyer wants* 4 *liters of milk. Can you dispense this? Model this as a graph problem with the vertex set V as the set of configurations* $c = (c1, c2, c3)$ *and an edge from c to d if d is reachable from c. Begin with* $(5, 0, 0)$. *Is this graph directed or undirected? Is it adequate to model the question: How to dispense* 4 *liters?*

Solution: At node D we can measure 4 liters. This is a directed graph because we can go directly from D to A but not reverse along that edge.

Graph below:-



5. There are many variations of BFS to solve various needs. For example, suppose that every edge $e = (u, v)$ also has a weight $w(e)$ (say the width of the road from u to v). Assume that the set of values that $w(e)$ can take is small. For a path $p = (v_1, v_2, \dots, v_k)$, let the weight $w(p)$ be the minimum of the weights of the edges in the path. We would like to find a shortest path from a vertex s to all vertices v. If there are multiple such paths, we would like to find a path whose weight is maximum. Can we adapt BFS to detect this path?

Solution: The key changes are:

- Maintain a new variable called width[u] which records the highest width path so far.
- Whenever a new node is first encountered, both the width and distance is set.
- If an alternate path is discovered, then the width is checked and the path is updated.

See Pseudo Code in Algorithm 12. Using *parent* variable, you can trace out the path.

Algorithm 12: Shortest Path with Highest Width Path BFS

```
Data: All nodes initially have variables $visited$ set to $False$, $parent$ set to $void$, $width\_path$
      set to $-\infty$ and $level$ set to 0
Function ModifiedBFS(Node $s$):
    Queue $Q$
    $s.visited \leftarrow True$
    $Q.enqueue(s)$
    while not $Q.empty()$ do
        $curr \leftarrow Q.dequeue()$
        foreach $(neighbour, edge\_weight)$ in $curr.adjacent$ do
            if not $neighbour.visited$ then
                $neighbour.visited \leftarrow True$
                $neighbour.level \leftarrow curr.level + 1$
                $Q.enqueue(neighbour)$
            end
            $new\_width\_path \leftarrow \min(curr.width\_path, edge\_weight)$
            if $curr.level = neighbour.level - 1$ and
             $new\_width\_path > neighbour.width\_path$ then
                $neighbour.width\_path \leftarrow new\_width\_path$
                $neighbour.parent \leftarrow curr$
            end
        end
    end
end
```

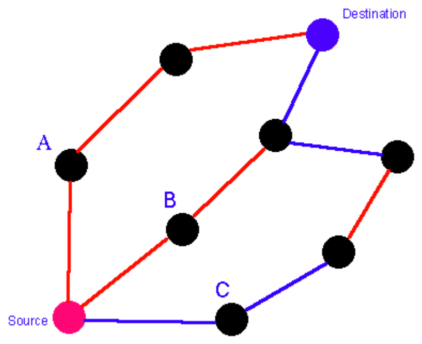
6. *Write an induction proof to show that if vertex r and v are connected in a graph G, then v will be visited in call BFSConnected(Graph G = (V, E), Vertex r, int id).*

Solution: Let's use induction on shortest distance d from r.

- Base Case: d = 1, While r is popped, all nodes from adjacency list of r is added to queue and hence get visited.
- Now, let us assume this holds for a distance k, i.e, all the nodes at a shortest distance k from a node r are visited in call BFSConnected(Graph G = (V, E), Vertex r, int id), we have to prove that all nodes at a shortest distance of k+1 will also be visited. Consider the path from r to the node, let's denote it by $r.p_0....p_k.p_{k+1}$. p_k is visited by induction hypothesis as it's at a distance of k from r and is added to BFS queue. When p_k was popped out of queue if p_{k+1} was not present in the queue then it will be added to the queue by the algorithm implying p_{k+1} will eventually get visited.

7. *Suppose that there is an undirected graph G(V,E) where the edges are colored either red or blue. Given two vertices u and v. It is desired to (i) find the shortest path irrespective of colour, (ii) find the shortest path, and of these paths, the one with the fewest red edges, (iii) a path with the fewest red edges. Draw an example where the above three paths are distinct. Clearly, to solve (i), BFS is the answer. How will you design algorithms for (ii) and (iii)?*

Solution: Example:-

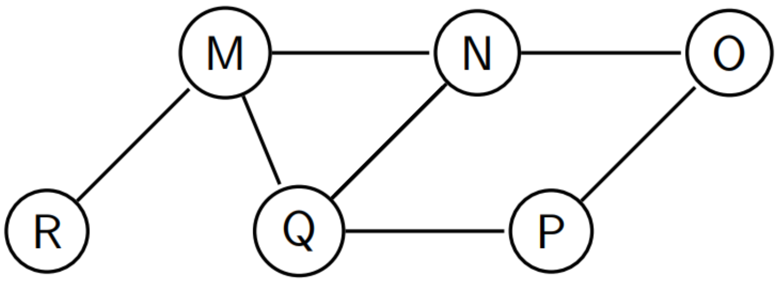


Here (i) takes A path, (ii) takes B path and (iii) takes C path.

- (i) Normal BFS will work.
- (ii) (Similar to Q5 in this Tutorial). We can modify BFS to obtain this. We want that between adjacent levels of BFS, if we have a choice between red and blue, then we choose the blue edge. Modification: if in bfs while checking in adjacency list if some node is already visited and the node is linked to it's parent with a red edge and that node's parent and current node are at same level then switch the parent of that node to the current node.
- (iii) We will do 0-1 bfs to do it. Visit all the nodes that can be visited from blue edges from source till blue edge got all exhausted. Now, visit all the nodes that can be visited from one red edge from the visited nodes. Then again visit the nodes which can be visited only with blue edge. Continue this process, till all nodes are visited. (Can you generalize it to problem with k color edges and you want to reach all the nodes with minimum no of color changes of edges?)

All the parts can be easily solved from Dijkstra's algorithm by suitable choices of weight.

8. *Is MNRQPO a possible BFS traversal for the following graph?*



Solution: No, because N was visited before Q, so O should be visited before P. This is because O will be added to queue when N was visited while P will be added when Q was visited.

Tutorial 10

1. *Give an algorithm that checks if a graph is 2-vertex connected.*

Solution: We will modify DFS such that it can answer the question of whether the graph is 2VC or not. The key here is to identify a condition that can be verified at a particular vertex with respect to the rest of the graph. First, some relevant definitions.

Two-Vertex Connected (Definition): A connected graph is said to be two-vertex connected (2-connected or 2VC) if it has more than two vertices and remains connected on the removal of any one vertex.

Articulation Point (Definition): A vertex in a connected graph is said to be an articulation point (or a cut-vertex) if the removal of that vertex results in a disconnected graph.

Back Edges: An edge from u to v such that v is an ancestor of u but not part of DFS of tree (treat the directed graph as a tree). Presence of a back edge indicates a cycle in a directed graph.

So essentially, if a graph is 2VC, then it has no articulation points (and the converse is also true). A vertex u in a DFS tree is an articulation point if and only if there is a child v of the vertex which has no back edge from the sub-tree rooted at the child to some ancestor in DFS tree of u (exception: root with at least two children is an articulation point).

Hence, the necessary and sufficient condition for 2VC is: For every vertex u, there is at least one back edge from the sub-tree rooted at u (excluding u) to some ancestor of u.

For the root of the tree if it has more than one child, it is an articulation point. Now consider an edge from u to v. We will modify DFS such that DFS(v) returns the smallest arrival time to which there is a back edge from the sub-tree rooted at v (including v) to some ancestor of u. If there is a back edge out of the sub-tree rooted at v, that vertex is something visited before v and thus will have a smaller arrival time (arrival[a] > arrival[b] for a back edge a to b). We also keep track of the earliest visited vertex that is reachable from the sub-tree of u. Now for the vertex u, if the value returned by DFS(v) is more than arrival[u] (for any such v), then u is an articulation point. The time complexity will be $\mathcal{O}$(V + E).

Algorithm 13: Two-Vertex Connected

```
Function twoVC(V):
    parent ← null
    foreach u in V do
        if not visited[u] then
            DFS(u, parent)
        end
    end
    foreach u in V do
        if AP[u] then
            return False
        end
    end
    return True
end
```

Algorithm 14: Modified DFS

```
time ← 0;
Function DFS(u, parent):
    visited[u] ← true
    arrival[u] ← evv[u] ← time++
    foreach v in adj[u] do
        if not visited[v] then
            children[u]++
            DFS(v, u)
            evv[u] ← min(evv[u], evv[v]) // update earliest visitable vertex time
            if parent is not NULL and evv[v] ≥ arrival[u] then
                AP[u] ← True // no back edge from v to ancestors of u
            end
        end
        else
            if v ≠ parent then
                evv[u] ← min(evv[u], arrival[v])
            end
        end
    end
    if parent is NULL and children[u] > 1 then
        AP[u] ← True // root with more than one child
    end
end
```

2. *Suppose that we have a graph and we have run DFS starting at a vertex s and obtained a DFS tree. Next, suppose that we add an edge (u,v). Under what conditions will the DFS tree change? Give examples.*

Solution: Assuming the graph was directed, the following cases are:-

1. If (u, v) forms a back edge (an edge from a descendant to an ancestor in the DFS tree), the DFS tree structure does not change, but the graph's overall structure now includes a cycle.

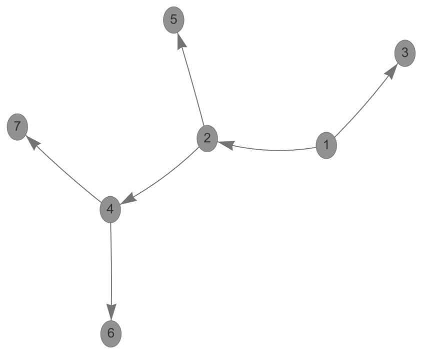


Figure 12: Original

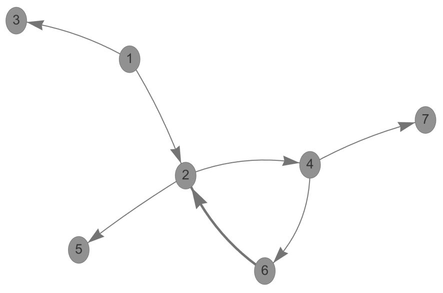


Figure 13: Added Edge

As you can see the edge (6, 2) was added but it doesn't affect the original tree as 2 was already visited first.

2. If (u, v) forms a forward edge (an edge from an ancestor to a descendant in the DFS tree), the DFS tree structure may change. Also this is assuming that an edge (u, v) didn't already exist before.

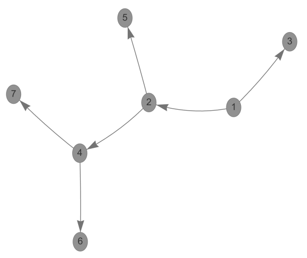


Figure 14: Original

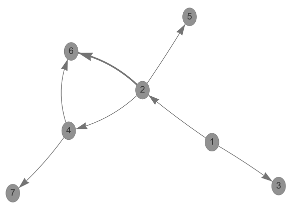


Figure 15: Added Edge

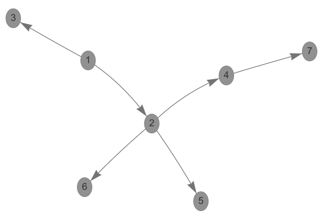


Figure 16: Changed Tree

As you can see adding the edge (2, 6) may affect the tree if 2 visits 6 first instead of 4.

3. If neither of the above, then the DFS tree structure changes if u is visited before v in the original tree.

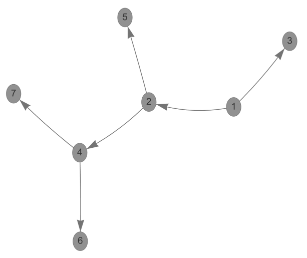


Figure 17: Original

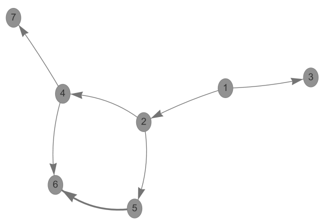


Figure 18: Added Edge

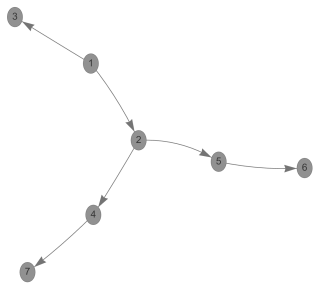


Figure 19: Changed Tree

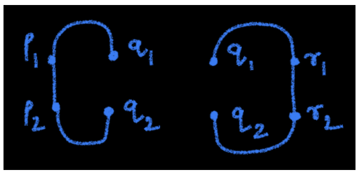
As you can see adding the edge (5, 6) may affect the tree if 5 is visited before 6.

3. *Let G(V,E) be a graph. We define a relation on edges as follows: two edges e and f are related (denoted by e f iff there is a cycle containing both.) Show that this is an equivalence relation. The equivalence class [e] of an edge e is called its connected component. What is the property of the equivalence relation when we say the graph is 2-edge connected?*

Solution:

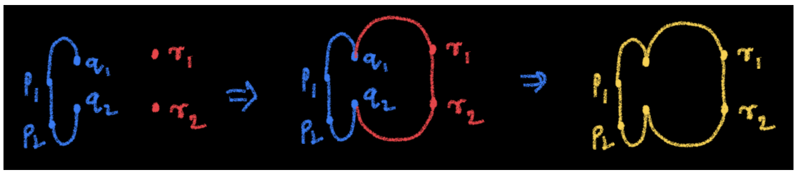
- Let's call this relation R. R is symmetric trivially.
- For transitivity, let's consider 3 edges p,q,r such that pRq and qRr, we need to show pRr. Let e_1 and e_2 denotes endpoints of any edge e. Now, let's suppose we delete r from the graph and showed that there is a path between r_1 and r_2 with p being one of the edge, then we are done, we can construct the cycle by using the path and r. Now, pRq implies that after deleting q you have a path between end points of q which contain p as an edge, let's call this P_1. Also, qRr implies that after deleting r you have a path between end points of r which contain q as an edge, let's call this P_2. Claim is augmenting P1 into P2 you can get the desired path which contain p as an edge. Augmentation had to be carefully handled!

Cycles of p, q and q, r looks like this:-

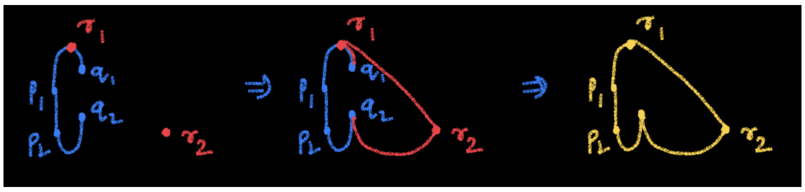


Cases are following (the figure on left shows placement of r_1 r_2, middle shows a possible cycle for q and r and the right one shows a possible cycle for p and r):-

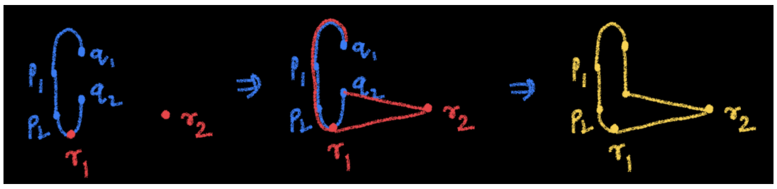
- **None of r_1 and r_2 lie on the cycle of p and q:**



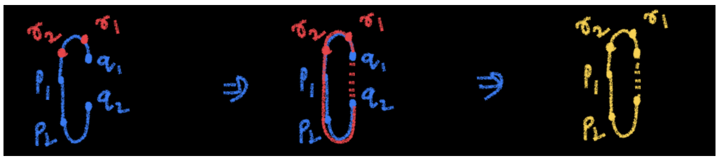
- **Only r_1 lies on the cycle of p and q:**
  1. r_1 on path p_1 and q_1:



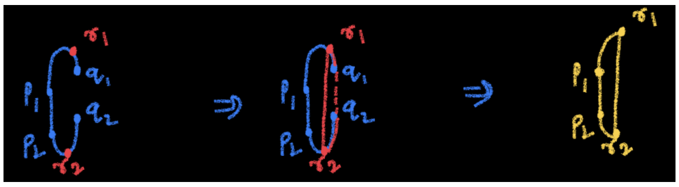
  2. r_1 on path p_2 and q_2:



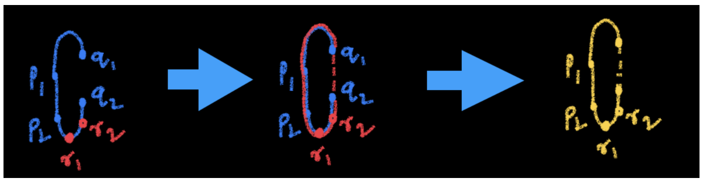
- **Only r_1 lies on the cycle of p and q:** Same as the previous case.
- **Both r_1 and r_2 lie on the cycle of p and q:**
  1. Both on path p_1 and q_1:



  2. r_1 on path p_1 and q_1 and r_2 on path p_2 and q_2:



  3. Both on path p_2 and q_2:



- For 2-edge connected component, the given equivalence relation turns into edges which are not separated by a Articulation Point

4. *Modify Kosaraju's algorithm to identify all SCCs of a graph.*

Solution: Modification: Run DFS on G and partion them into different sets depending upon whether they are visited in the same call of $DFS(v)$ or not. Let's call this partion as P_1. And then run DFS on G^R and partion them into different sets depending upon whether they are visited in the same call or not. Let's call this partion P_2. Then element wise intersection of P_1 and P_2 form the SCC's of given graph. A better (linear time) way to do this is mentioned in the link click here.

5. *Give similar modification for the algorithm SC.*

Solution: In a strongly connected component, there must be a node n that is visited the "earliest". The array "earliest" for any node x contains the smallest (hence earliest) arrival time of vertex reachable from x. All the nodes in the SCC same as n will have "earliest" value equal to arrival time of n (that can be seen from the code below). Any node not in that SCC (obviously arrival time > that of n) will have a different "earliest" value (otherwise it means that it is connected to n).

Algorithm 15: SCC Detection using Tarjan's Algorithm

```
Function SCC(u):
    visited[u] ← true
    arrival[u] ← earliest[u] ← time++
    stack.push(u)
    inStack[u] ← true
    foreach v in adj[u] do
        if not visited[v] then
            SCC(v)
            earliest[u] ← min(earliest[u], earliest[v]) // Update earliest value based on
                subtree
        end
        else if inStack[v] then
            earliest[u] ← min(earliest[u], arrival[v]) // Update earliest value based on
                back edge
        end
    end
    if earliest[u] = arrival[u] then
        // Root of an SCC detected
        while stack.top() ≠ u do
            w ← stack.pop()
            inStack[w] ← false
            w.scc_id ← SCC_ID
        end
        w ← stack.pop()
        inStack[w] ← false
        w.scc_id ← SCC_ID++ // Increment the SCC_ID
    end
end
```

Algorithm 16: Main Function

```
Data: $V$ is the set of nodes.
time ← 0
SCC_ID ← 0
Function Main($V$):
    foreach $v$ in $V$ do
        if not visited[$v$] then
            SCC($v$)
        end
    end
end
```

6. *If we run BFS on a directed graph, can we define the same classes of edges, i.e., cross, tree, back, and forward edges? Give conditions for each class.*

Solution: For breadth-first search, there are no forward edges. This is easy to prove: If there is an edge from A to B and from B to C then the edge from A to C is called a forward edge. But if such an edge exists, then at vertex A, both B and C will be part of the next visited level of BFS. Hence A to B and A to C would be part of the BFS tree, making both of them tree edges.

The classification for the other edges would be as follows (assume edge from u to v):

Tree: All the edges that are part of the BFS tree. Here v.level = u.level + 1.

Back: Not a tree edge, and vertex with smaller level is ancestor of vertex with higher level.

Cross: Neither a tree edge nor a back edge.

7. *Let us modify DFSRec to detect cycles during the run. Give the expression for the condition to detect the cycles.*

Algorithm 17.12: DFSREC(Graph G, vertex v)

```
$v.visited$ := $True$;
$v.arrival$ := $time++$;
for $w \in G.adjacent(v)$ do
    if $w.visited == False$ then
        DFSREC($G, w$)
    else
        if condition then
            throw "Found Cycle"
$v.departure$ := $time++$;
```

Solution: $w.arrival < v.arrival$ **and** $w.departure = DEFAULT$ **and** $w \neq v.parent$

All variables are initialized to a default value called $DEFAULT$.

Tutorial 11

1. *Sorting a sorted array*
    1. *Modify the algorithm* Partition *such that it detects that the array is already sorted.*
    2. *Can we use this modification to improve quicksort?*

Algorithm 17: ModifiedPartition(int* A, int l, int u)

```
pivot ← A[l];
is_sorted ← true;
i ← l-1;
r ← u+1;
while true do
    if $i \geq l$ && $A[i] > A[i+1]$ then
        is_sorted ← false;
    end
    i ← i+1;
    while $pivot < A[i]$ do
        if $A[i] > A[i+1]$ then
            is_sorted ← false;
        end
        i ← i+1;
    end
    if $j \leq r$ && $A[j] < A[j-1]$ then
        is_sorted ← false;
    end
    j ← j-1;
    while $pivot > A[j]$ do
        if $A[j] < A[j-1]$ then
            is_sorted ← false;
        end
        j ← j-1;
    end
    if $i \geq j$ then
        return (is_sorted ? -1 : j);
    end
    swap(A[i], A[j]);
end
```

Algorithm 18: ModifiedQuicksort(int* A, int l, int u)

```
if $l \geq u$ then
    return;
end
p ← ModifiedPartition(A, l, u);
if $p == -1$ then
    return;
end
ModifiedQuicksort(A, l, p);
ModifiedQuicksort(A, p+1, u);
```

Solution: The idea is to add a boolean flag is_sorted, which is true initially and becomes false whenever there is $A[i] > A[i+1]$ or $A[j] < A[j-1]$. Since the standard partition algorithm traverses the entire array, any index i (or j) where the original order is against the sorted order is detected.

Further, once detected, we can output -1 (or an invalid index) as the answer, and the quicksort algorithm can detect -1 as the partitioned output and exit quickly. This way, sorting an already sorted array takes only $O(n)$ time, versus $O(nlog(n))$ or worse, $O(n^2)$. The modified algorithms are written above.

2. *Modify* RadixSort *such that it considers bits from right to left.*

Solution: We cannot just recursively partition and sort from the right to left bit; the array will not remain sorted. One idea is to partition the array by the rightmost (least significant) bit first, then by the second bit, all the way till the leftmost bit, but partition the entire array each iteration.

Note that the partitioning should be **stable** i.e. when we partition by the second bit, we do not want the previous order to be upset. To illustrate, consider the array (of binary numbers) $[00, 11, 10, 01]$. After the first partition by the rightmost bit it becomes $[00, 10, 11, 01]$. After the second partition, we want it to become $[00, 01, 10, 11]$ only, with the relative ordering among the numbers with the first bit 0 remaining as before.

Partitioning (quick sort and radix sort partition) algorithms that execute in place do not guarantee the same. For this reason, we choose a "partitioning" algorithm that populates a new array first with all the numbers (otherwise in order) from the original that has the b^{th} bit as 0, then all the numbers (in the same order otherwise) with the b^{th} bit 1. Note that $num\&(1 << b)$, the bitwise AND equals $(1 << b)$ if the b^{th} bit of num is 1; and it equals 0 otherwise.

Algorithm 19: ModifiedRadixPartition(int* A_old, int n int b)

```
A_new ← new int [n];
j ← 0;
foreach i in $[0, 1 \ldots n-1]$ do
    if (A[i] & (1 n)) == 0 then
        A_new[j] ← A[i];
        j ← j+1;
    end
end
foreach i in $[0, 1 \ldots n-1]$ do
    if (A[i] & (1 n)) != 0 then
        A_new[j] ← A[i];
        j ← j+1;
    end
end
delete [] A;
return A_new;
```

Algorithm 20: ModifiedRadixSort(int* &A, int n, int num_bits)

```
foreach b in $[0, 1 \ldots num\_bits - 1]$ do
    A ← ModifiedRadixPartition(A, n, b);
end
```

3. *Draw the execution tree for the Heap sort. Let us suppose the input array is of size 3.*

Solution: The BuildHeap section does Heapify(2) and Heapify(1) (optional), which are single-step functions that do nothing since the sub-heaps starting at index 1 and 2 are singletons i.e. nothing to build.

BuildHeap(0), first compares A[0] with A[1]. If A[0] is found to be the larger element, then it compares with A[2], and if A[0] is larger then the heap is already built; else, there is one swap. If A[1] is found to be larger, then it is, in turn, compared with A[2], and the larger of the two is swapped with A[0].

This gives us a (complete binary) execution tree of depth 2. Then, A[0] is swapped with A[2], wherein the largest element is popped from the heap (deterministic step). After this, the Heapify compares A[0] and A[1], ensuring with or without a swap that A[0] is larger. This gives two execution options. Then, A[0] and A[1] are again swapped (deterministic, when A[0] is popped from the heap). Then, A[0] is popped from the heap, making it empty and the array sorted.

We notice that the execution tree has 8 leaves and is a complete binary tree of depth 3, assuming that we compress the deterministic steps (popping, swapping after a case is evaluated, etc.). This is valid since 8 is greater than 6, the number of possible permutations of A[0], A[1], and A[2]; each shows up in the leaves at least once.

4. *The quickselect algorithm finds the kth smallest element in an array of n elements in average-case time $O(n)$. The algorithm, which uses the quicksort partition is as follows:*

Algorithm 21: QuickSelect(Array G, int k)

```
n ← A.size();
p ← random(1, n);
swap(A[1], A[p]);
p ← partition(A, p);
if $p == k$ then
    return A[p];
end
if $p > k$ then
    return QuickSelect(A[1...p-1], k);
end
else
    return QuickSelect(A[p+1...n], k-p);
end
```

1. *Suppose the pivot was always chosen to be the first element of A. Show that the worst-case running time of quickselect is then $O(n^2)$.*
2. *Show that the expected running time of (randomized) quickselect is $O(n)$.*

Solution: Worst case analysis: Let us consider a sorted array (in increasing order) of size n, with $k = n$; that is, we are supposed to find the largest element in the array with this algorithm. For this purpose, consider that the array is 1-indexed.

If $p = 1$ instead of a random pivot, then the returned value after a partition would be 1 since the index i stops at 1 and j too does stop at 1 where $A[i] = A[j] = A[p] = A[1]$. There is no change in the array, as there is no swap before the return. Since $k = n > 1$, we return $\text{QuickSelect}(A[2 \dots n], n - 1$.

Let the time taken for partition of an array of size n be bounded by $an + b$, where a and b are appropriate constants. Partition involves $n + 1$ increments/decrements overall and less than that many $O(1)$ operations for swapping elements and checking $i \geq j$ so this bound is true. Further, the QuickSelect algorithm involves $O(1)$ operations for getting size, pivot, and comparing $p == k$ and the if branch.

Let those $O(1)$ operations take a total time bounded by c. Then, for an array of size n, the time taken for Quickselect, $Q(n)$ is bounded by $c + an + b + Q(n-1)$. The last term is for the recursive call. If $n = 1$, then k has to be 1, and the if condition would anyway evaluate to true, so the base case time is bounded by $c + a + b$.

Expanding the recursive sum, we get $Q(n) \leq (an + d) + (a(n-1) + d) + \cdots + (a + d)$, where $d = b + c$. Using the sum of the AP formula, we get $Q(n) \leq an(n+1)/2 + dn$. Further, if we trace the exact same argument but with lower bounds on the same example, we get a different set of constants but the fact that $Q(n) \geq (en^2 + fn + g)$. Therefore, the worst-case bound $O(n^2)$ is a tight bound. **Q.E.D.**

WRONG Expected runtime analysis: A lot of students will be thinking of forming an inductive argument that looks like this:

- Let statement $S[n]$ be that $Q_E(n)$, expected quick-select time be $O(n)$.
- The base case is true, since $Q_E(1)$ is constant, or $O(1)$.
- If $S[n]$ is true, then time for $Q_E(n+1)$ = $O(n)$ for partition and $Q_E(n)$ for the recursive step. If $Q_E(n) \in O(n)$, then $O(n)$ + $O(n) = O(n)$ so $Q_E(n+1) \in O(n)$.
- $S[1]$ and $S[n] \implies S[n+1]$ means $\forall n \;\; S[n]$ i.e. $Q_E(n) \in O(n)$.

This is severely flawed since this same argument could be used as 'proof' that quicksort or mergesort also is $O(n)$ or that the worst case is $O(n)$. Heck, you can also 'prove' that bubble sort is $O(n)$ this 'way'. We all know that it is laughably wrong.

The reason why this proof is flawed is that we cannot just add complexities in recursive or inductive proofs. We should pick constants a, b, and if we are to prove that it is of $O(n)$, that for all n, the average/expected time is bounded by $an + b$, and if we are to use induction, we are to prove that the EXACT SAME constants for the bound are true during $S[n] \implies S[n+1]$.

CORRECT Expected runtime analysis: Let us use a strong inductive argument: $Q_E(n')$ for a random array, random value of p and any k and $n' \leq n - 1$ be bounded by $\alpha n + \beta$. Note that this is the expected or average time, not the worst case. Let this argument be $S[n-1]$. We are to prove that $S[n-1] \implies S[n]$.

The partition algorithm returns a value p after partitioning A, such that all elements of $A[1 \ldots p]$ are less than or equal to a pivot (an element in A) and all elements of $A[p+1 \ldots n]$ are greater than or equal to that pivot value. As a result of randomization, it is equally likely that the p can take any value ranging from 1 to n. Let us evaluate $Q_E(n, k)$.

Now, $Q_E(n, k, p)$ where $p < k$ is $an + d + Q_E(n-p, k-p) \leq an + d + \alpha(n-p) + \beta$. $Q_E(n, k, p)$ where $p > k$ is $an + d + Q_E(p-1, k) \leq an + d + \alpha(p-1) + \beta$. Lastly, $Q_E(n, k, k)$ is the time for partition (and the overhead) which is just $an + d$.

Let us find $(1/n)\sum_{p=1}^{n} Q_E(n,k,p)$. For this,

$$\begin{aligned}
\sum_{p=1}^{n} Q_E(n,k,p) &\le \sum_{p=1}^{n}(an+d) + \sum_{p=1}^{k-1}(\alpha(n-p)+\beta) + \sum_{p=k+1}^{n}(\alpha(p-1)+\beta) \\
&= (an^2+dn) + (n-1)\beta + \frac{(k-1)(2n-k)}{2}\alpha + \frac{(n-k)(n+k-1)}{2}\alpha \\
&= (an^2+dn) + (n-1)\beta + \alpha\left(\frac{1}{2}(2nk+k-2n-k^2+n^2-k^2+n-k\right) \\
&\le (an^2+dn) + n\beta + \frac{\alpha}{2}(2nk+n^2-2k^2) \\
\implies Q_E(n,k) &\le (an+d+\beta) + \frac{\alpha}{2}\left(2k+n-\frac{2k^2}{n}\right)
\end{aligned}$$

The term $2k - 2k^2/n$ has a maxima at $k = n/2$, with its value as $n/2$. Plugging it in, we get for any value of k, $Q_E(n) \le (an+d+\beta) + \alpha n$. This is not good enough.

For the expected value, though, k can take any value, so we assume it takes any value with probability $1/n$ each. Then, $Q_E(n) \le 1/n \sum_{k=1}^{n}\left(an+d+\beta+\frac{\alpha}{2}(2k+n+2k^2/n)\right)$. The sum of the second term evaluates to $n(n+1)+n^2-(n+1)(2n+1)/3 = (4n^2-1)/3$. Therefore, $Q_E(n) \le an+d+\beta+2\alpha n/3$.

If we choose $\alpha \ge 3a$ then we get $Q_E(n) \le \alpha n + d + \beta$. The last problem is the extra constant, d. For this, we can just substitute a with $a+d$ in all the previous steps: a larger constant for the time for partition plus the overheads. If $\alpha \ge 3(a+d)$ then we get $Q_E(n) \le \alpha n + \beta$. That is, $S[n-1] \implies S[n]$.

And, $S[1]$ is obviously true since we can choose α and β to satisfy the lower bounds; from these two, we can say $\forall n \;\; S[n]$ i.e. $Q_E[n] \in O(n)$. **Q.E.D.**

Follow-up: Checking correctness of QuickSelect: Consider the 1-indexed array array $[61, 51, 56, 69, 90, 84, 86, 45, 88, 61, 21]$ with $p = 1$ (the pivot index) and $k = 5$. This is the exact same array used in the lecture slides to explain the Partition algorithm. The returned value is $j = 5$, as in slide 19. Here, $j = k$ so the algorithm returns $A[5] = 45$. Verify that this is the second smallest element, *not* the fifth smallest.

Corrected version of QuickSelect algorithm: once the partition returns index p: ensure that $A[p]$ is the p^{th} largest element in the array, by finding $max(A[1\ldots p])$ and swapping that with $A[p]$, then proceeding. This does not increase the time complexity; it only increases a (and therefore α).

Algorithm 22: CorrectedQuickSelect(Array G, int n, int k)

```
p ← random(1, n);
swap(A[1], A[p]);
p ← partition(A, p);
m ← argmax(A[1...p]);
swap(A[m], A[p]);
if $p == k$ then
    return A[p];
end
if $p > k$ then
    return CorrectedQuickSelect(A[1...p-1], p-1, k);
end
else
    return CorrectedQuickSelect(A[p+1...n], n-p, k-p);
end
```

Tutorial 12

1. *Give an algorithm to undo the last union operation assuming there is path compression or not?*

Solution: Every call to the Union function makes only these 2 deterministic changes:-

1. y.parent = x
2. x.size = x.size + y.size

So to revert this change, we need to do this:-

1. x.size = x.size - y.size
2. y.parent = y

We just need to store some reference/pointer to nodes x and y from the latest Union call. We'll add a line to the Union function. See Algorithm 23 below.

Algorithm 23: Ctrl + Z operation

```
Data: There's a new variable $latest$ (initially $NULL$) that shall store $x, y$ as said above.
Function Union($x, y$):
    $x \leftarrow$ FindSet($x$)
    $y \leftarrow$ FindSet($y$)
    if $x.size < y.size$ then
        SWAP($x, y$)
    end
    $y.parent \leftarrow x$
    $x.size \leftarrow x.size + y.size$
    $latest \leftarrow (x, y)$ // New line
end
Function Undo():
    if $latest = NULL$ then
        return
    end
    $(x, y) \leftarrow latest$
    $x.size \leftarrow x.size - y.size$
    $y.parent \leftarrow y$
end
```

2. *Show that complexity bound of $O(n\ log(n))$ in the running time of the union-find algorithm is tight, both based on weighted union and forest (no path compression)?*

Solution: Weighted Union: Bounds and Tightness

Let $X = \{x_1, x_2, \ldots x_k\}$ be one union with a linked-list-based data structure, with $k < n$. Let $Y = \{y_1, y_2, \ldots y_h\}$ be another. Let a call be made to $union(x_i, y_j)$ for appropriate i and j. What happens is that in constant time, the heads of each linked list (data structures for X and Y) are found.

Case: $k \geq h$: Then, the tail of the data structure for X gets its next pointer reset to the first node (after head) of Y [constant time]. Then, for each node in Y, the head pointer gets set to the head of X [time: $O(h)$]. Lastly, the head of Y is deleted [constant time], so there is one set $X \cup Y$.

Case: $k < h$: The reverse happens. Notice that the new data structure (new set) has size $h + k \geq 2k$.

An important observation is that whenever the head pointers for any element in X are changed (to that of Y or Z, etc.), then *all* elements in the set $Z \supseteq X$ [and therefore X] get their pointers changed (with head deleted), and the new super-set formed has size at least double that of X.

But, no superset can have a size of more than n. Thus, merging can occur at max $log_2(n/k)$ times. Starting from scratch with $k = 1$, the doubling (and reset of the head pointer for x_1) occurs at max $log_2(n)$ times.

The worst-case scenario is when each merge has $k = h$, a.k.a. sets of equal sizes. Thus, the singleton $\{x_1\}$ then becomes a subset of a set of size 2, later size 4, 8, ... This means that the head pointer gets reset exactly $log_2(n)$ times.

Moreover, there are $n/2$ merges of sets of size 1, $n/4$ merges of sets of size 2 following that, and so on. Let the merge take time $an + b$, where a and b are appropriate constants. Then, the net time taken would be

$$\frac{n}{2}(a+b) + \frac{n}{4}(2a+b) + \cdots + [log(n) \text{ terms}] = \frac{na}{2} log(n) + bn$$

This proves the tightness of the weighted union. For the upper bound, all equalities are replaced by $\leq$ inequalities.

Solution: Forest: Bounds and Tightness

Let us consider the forest with calls such that there is a complete binary tree X with $n/2$ nodes formed at first, then all the remaining calls are of type $union(x_i, y)$ where i is an appropriate index for x_i to be a leaf in X, and y a free element (singleton or not part of any set).

In each of the remaining $n/2$ calls, the $FindSet(x_i)$ takes $log(n) - 1$ recursive calls to reach the root of X, and $FindSet(y)$ immediately returns itself. Note that the size of X is much more than of $\{y\}$, and therefore, y gets merged with its parent set to the root of X.

This means that the time taken is $a \cdot log(n) + b$ where a, b are appropriate constants. For $n/2$ iterations, the time taken is $(an/2) \cdot log(n) + (bn/2)$. Then, we add the time to construct the perfect tree, which cannot be less than αn and cannot be more than $\beta n log(n)$ for appropriate constants α, β. Either way, the net time taken is $\Theta(n \cdot log(n))$.

This is a tight example. To prove the bounds themselves, we first prove the bounds on the height of a tree with respect to its size using an inductive argument. We leave it as an exercise for you to prove that that bound is $log_2(n) + 1$. Then, given that in each call, the heights of both sets cannot exceed $c \cdot log(n)$ for some constant c, the time complexity overall is bounded by $O(n \cdot log(n))$.

3. *Give an implementation of printSet(x) function in UnionFind (with path compression) that prints the set containing x. You may add one field in each node and must not alter the asymptotic running times of the other operations.*

Solution: See Algorithm 24 below.

Algorithm 24: Union-Find with *printSet(x)*

```
Data: Union-Find data structure with an additional field setElements, where
      setElements[root] stores all elements in the set rooted at root.
Function FindSet($x$):
    if $x.parent \neq x$ then
        $x$.parent ← FindSet($x$.parent)
    end
    return $x$.parent
end
Function Union($x$, $y$):
    $rootX$ ← FindSet($x$) $rootY$ ← FindSet($y$) if $rootX \neq rootY$ then
        if $rootX.size < rootY.size$ then
            SWAP($rootX$, $rootY$)
        end
        $rootY$.parent ← $rootX$ $rootX$.size ← $rootX$.size + $rootY$.size
          $rootX$.setElements ← $rootX$.setElements ∪ $rootY$.setElements // Merge
          elements
        $rootY$.setElements ← ∅ // Clear $rootY$'s set
    end
end
Function PrintSet($x$):
    $rootX$ ← FindSet($x$) // Find root of $x$
    return $rootX$.setElements // Return all elements in $x$'s set
end
```

4. *Modify proof of theorems 17.1 and 17.3 to support non-unique edges.*

Solution: Modified Proof for Theorem 17.1 (Correctness of Kruskal's Algorithm): Kruskal's algorithm correctly returns a minimum spanning tree (MST) even if edge lengths are not unique.

Assumption: Sort the edges in non-decreasing order of weights. For ties, resolve them consistently (e.g., lexicographically by edge endpoints). Let $e_1, e_2, \ldots, e_m$ be the sorted edge list.

Proof: We use proof by contradiction:

- Suppose the algorithm outputs a spanning tree T_K that is *not* an MST.
- Let T^* be an MST of G.
- Let i be the smallest index such that e_i is in T_K but not in T^*.
- Adding e_i to T^* forms a cycle C in T^*. Let e' be the maximum-weight edge in C.
- If e' has the same weight as e_i, the algorithm might select e' or e_i based on the tie-breaking rule.
- Removing e' from C yields another valid spanning tree T', and the total weight of T' is the same as T^*.

Thus, T_K must be a valid MST, even with non-unique weights.

Modified Proof for Theorem 17.3 (Minimum Edge of a Cut is in the MST): For any cut C in the graph, at least one minimum-weight edge in the cut will be part of an MST.

Proof: We use proof by contradiction:

- Let C be a cut of G for some subset of vertices $S \subseteq V$, and let e be a minimum-weight edge in C.
- Suppose e is not part of any MST T.
- By the cut property, at least one minimum-weight edge of C must be included in every MST.
- If there are multiple edges with the same weight as e, any of these edges can replace e while still maintaining the spanning tree and optimality properties.
- Since e can always be replaced without increasing the tree weight, there exists an MST containing e, contradicting our assumption.

Thus, at least one minimum-weight edge of the cut is part of the MST, even with non-unique edge weights.

5. *Kruskal's algorithm can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted into order. Show that for each minimum spanning tree T of G, there is a way to sort the edges of G in Kruskal's algorithm so that the algorithm returns T.*

Solution: Let $G = (V, E)$ be a graph with edge weights $L(e)$ for $e \in E$, and let T be a minimum spanning tree of G.

1. Sorting Edges: - Sort edges by their weights $L(e)$ in non-decreasing order. - For edges with the same weight, modify the sorting order such that edges in T are prioritized.

2. Kruskal's Algorithm Behavior: - During Kruskal's execution, edges are processed in the sorted order. - Since edges in T are processed first, they will be included in the tree unless adding an edge causes a cycle. - Edges not in T but with the same weight will not disrupt the construction of T, as T satisfies MST properties.

3. Result: - By prioritizing the edges in T, Kruskal's algorithm will construct T.

Example: Consider a graph $G = (V, E)$ with the following edges and weights:

Edge	Weight
$a \to b$	1
$b \to c$	2
$a \to c$	2

Possible MSTs: $T_1 = \{a \to b, b \to c\}$ and $T_2 = \{a \to b, a \to c\}$.

To construct T_1, sort edges as $(a \to b), (b \to c), (a \to c)$. To construct T_2, sort edges as $(a \to b), (a \to c), (b \to c)$.

6. *a. What is anti-arborescence?*
b. Show that Kruskal's and Prim's algorithm will not find a minimum spanning arborescence for a directed graph.
c. Give an algorithm that works on a directed graph.

Solution: An anti-arborescence is formed by reversing the edges of an arborescence.

Prim's algorithm assumes that all vertices are connected. But in a directed graph, every node is not reachable from every other node. So, Prim's algorithm fails due to this reason. In Kruskal's algorithm, In each step, it is checked that if the edges form a cycle with the spanning-tree formed so far. But Kruskal's algorithm fails to detect the cycles in a directed graph as there are cases when there is no cycle between the vertices (as concluded from the union-find method) but Kruskal's Algorithm assumes it to cycle and don't take consider some edges due to which Kruskal's Algorithm fails for directed graph.

Chu-Liu-Edmonds' Algorithm takes the following steps to find a MST in a directed graph:

1. Initially, every vertex is considered a subtree.
2. For each subtree, keep 1 incoming edge with the minimum weight.
3. If there is no cycle, go to 5.
4. If there is a cycle,
   (a) Merge subtrees with the cycle into one and update scores for all incoming edges to this merged subtree, and goto 2.
   (b) For each vertex in the subtree, add the weight of its outgoing edge chain to its incoming edges not in the subtree.
5. Break all cycles by removing edges that cause multiple parents.

7. *Prove that Boruvka's algorithm returns an MST.*

Solution: To prove that the given algorithm correctly computes the **Minimum Spanning Tree (MST)**, we must show:

1. The resulting tree is a **spanning tree**, i.e., it connects all vertices and is acyclic.

2. The resulting tree is a **minimum spanning tree**, i.e., its total weight is minimized.

Definitions and Key Properties:

Let $G = (V, E)$ be a connected, weighted, undirected graph.

- mst: The set of edges forming the growing MST.
- components: A set of connected components, initialized as $\{\{v\} \mid v \in V\}$.
- c.outgoingEdges.min(): For each component c, this finds the minimum-weight edge connecting c to a vertex in a different component.

The algorithm selects these edges iteratively, merging components, until all vertices belong to a single connected component.

Step 1: The Resulting Tree is a Spanning Tree

Connectivity: Initially, each vertex is its own component. At every iteration:

- The algorithm selects the minimum-weight outgoing edge for each component.
- Each selected edge connects two different components.

As edges are added, the number of components decreases. Eventually, all vertices belong to a single component, ensuring that the graph is connected.

Acyclicity:

- The algorithm only adds edges between distinct components.
- Since it never connects vertices within the same component, no cycles are formed.

Thus, the result is a connected, acyclic graph—a spanning tree.

Step 2: The Resulting Tree is a Minimum Spanning Tree

To prove that mst is a **minimum spanning tree**, we use the **Cut Property**:

Cut Property: For any cut (partition of V into two disjoint subsets S and $V \setminus S$), the minimum-weight edge crossing the cut is part of the MST.

- At each iteration, the algorithm selects the minimum-weight outgoing edge e for each component c.
- The edge e is the minimum-weight edge crossing the cut between c and the rest of the graph.
- By the cut property, e must be part of the MST.

Since every edge added satisfies the cut property, all edges in mst are guaranteed to be part of the MST.

Step 3: Termination

The algorithm terminates when all vertices are in a single connected component. At this point:

- The graph contains $|V| - 1$ edges, as required for a spanning tree with $|V|$ vertices.
- All edges in mst satisfy the cut property and maintain connectivity and acyclicity.

Thus, mst is both a spanning tree and a minimum-weight tree.

Inductive Proof of Minimum Weight

Base Case: Initially, mst $= \emptyset$, which trivially satisfies the properties of an MST.

Inductive Step:

- Assume after k iterations, mst is part of an MST.
- In the $k + 1$-th iteration, the algorithm adds the minimum-weight edge e connecting two components.
- By the cut property, e is part of the MST.
- The merged components still satisfy the properties of an MST.

By induction, the final mst is an MST.

Tutorial 13

1. *Modify Dijkstra's algorithm to compute the number of shortest paths from s to every vertex t.*

Solution: Remember that in Dijkstra's, we used a variable sp, with $sp[t]$ denote the length of the shortest path from s to t for any vertex t. The modification here is that for each vertex t, sp stores two values – $sp[t][0]$ represents the length of the shortest path and $sp[t][1]$ the number of paths from s to t of this length.

Initially, $sp[s]$ is initialized to $\{0, 1\}$ since there is one path (the null path). For every other $t \neq s$, $sp[t]$ is set to $\{\infty, 0\}$ representing no path or unreachable node.

During each iteration, we visit a vertex v from the priority queue, similar to Dijkstra's – the unvisited vertex with the shortest path length from s (we mark that visited, then). During the iteration, for all t that are neighbors of v,

- If $d(v, t) + sp[v][0] > sp[t][0]$, then the paths $s \dots v - t$ are suboptimum. Ignore.
- If $d(v, t) + sp[v][0] = sp[t][0]$, then set $sp[t][1] \leftarrow sp[v][1] + sp[t][1]$.
- If $d(v, t) + sp[v][0] < sp[t][0]$, then set $sp[t] \leftarrow \{sp[v][0] + d(v, t), sp[v][1]\}$.

In the second case, the order of visiting vertices (and ignoring vertices already visited) ensures that there is no path of length $sp[t][0]$ that passes through v before this iteration. Then, we add the paths that pass through v as the penultimate vertex. Note that there could be more paths via v, but where v is not penultimate, those paths get added in later iterations.

The third case: we found a better path(s) than any known. So, we ignore all previous paths and set $sp[t]$ to the new optimum, along with a number equal to the number of paths up to v from s.

Since the path lengths to the vertices (in order we visit in the algorithm) are monotonically increasing, there cannot be any better paths to v or more paths of the same length from s to v. This ensures correctness. An important point to be made is that this fails if there are edges of weight zero (exact 0). All edges should have positive weights.

We leave it as an exercise for you to write the algorithm/pseudocode/code given this.

2. *Show an example of a graph with negative edge weights and show how Dijkstra's algorithm may fail. Suppose that the minimum negative edge weight is $-d$. Suppose that we create a new graph G' with weights w', where G' has the same edges and vertices as G, but $w'(e) = w(e) + d$. In other words, we have added d to every edge weight so that all edges in the new graph have edge weights non-negative. Let us run Dijkstra on this graph. Will it return the shortest paths for G?*

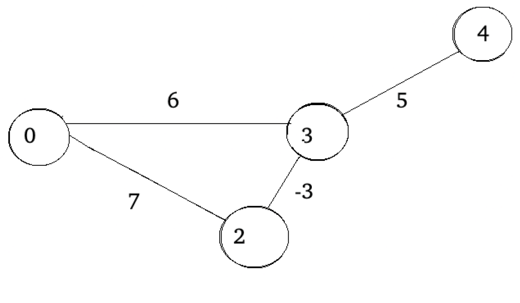


Figure 20: Failure of Dijkstra's Algorithm

Solution: Look at the graph shown on the previous page. Normal Dijkstra's algorithm reports the shortest path from vertex 0 to vertex 4 as $6 + 5 = 11$ before it even evaluates the existence of edge $(2, 3)$. If it does not update the shortest path to vertex 3, the algorithm directly fails (wrong answer).

A similar failure (wrong answer) occurs for vertex 4 if we update the path to vertex 3 but stop there; we need to push vertex 3 back to the priority queue and mark it unvisited. This, however, makes it inefficient.

If we add $d = 3$ to all edge weights, then the shortest path of the new graph will be $0 - 3 - 4$, of length 11 (17 after addition of d to all edge weights), versus $10 + 8 = 18$ (which is 9 before addition) for path $0 - 2 - 3 - 4$. Once again, we get the wrong path (not the shortest) on adding d to all edge weights.

To really get the shortest path, we should do the modification mentioned above (inefficient). A better approach is to update during each iteration, the shortest path lengths after evaluating **all** edges (u, v); with

$$sp[v] \leftarrow min(sp[v], d(u, v) + sp[u])$$

We do this for a number of iterations equal to the number of vertices (or until it terminates), and then we get the shortest paths with negative edges allowed.

The algorithm described above is **Bellman-Ford's Algorithm**, and the updating of the values of sp real-time and using already stored values to speed up computation (dynamic programming). You will read more about this in the fourth semester.

As far as correctness is concerned, this works unless there are *negative loops* (loop with the sum of edge weights negative), in which case no algorithm works (there is no shortest path to any vertex reachable from the loop).

3. *Let $G(V, E)$ be a representation of a geography with V as cities and (u, v) an edge if and only if there is a road between the cities u and v. Let $d(u, v)$ be the length of this road. Suppose that there is a bus plying on these roads with fare $f(u, v) = d(u, v)$. Next, suppose that you have a free coupon that allows you one free bus ride. Find the least fare paths from s to another city t using the coupon for this travel.*

Solution: First, we solve the *all-source cheapest path* (Dijkstra) from s. Then, we solve this all-source cheapest path from t, the destination. This gives the cheapest path from each stop to the destination.

Note that in these two subproblems, we do not assume the existence of a free ticket. We assume the lengths are stored in $from\_s$ and to_t, respectively.

We initialize the result path costs r (what we require) to the cheapest path from s to t without any free ticket coupon use. Clearly, adding the free ticket will not increase the minimum cost. Then, for each edge (u, v) we set

$$r \leftarrow min(r, from\_s[u] + to\_t[v] + 0)$$

Zero is the cost of the edge (u, v), zeroed due to the use of the free ticket. If it is an undirected graph, repeat for (v, u) also. Then, we return r.

The time complexity is $O(E\ log(V) + V)$. This can easily be verified and is not proved.

Regarding correctness, every path that utilizes the free ticket has to utilize it across one route (edge). For each edge, we compute the cheapest path utilizing it in $O(1)$ time using results already computed paths to its start and from its end. The cheapest among all these is the cheapest path in all.

4. *Suppose $w(u, v)$ is the width of the road between the cities u and w. Given a path π, the width $w(\pi)$ is the minimum of widths of all edges in π. Given a pair of cities s and t, is it possible to use Dijkstra to determine the largest width of all paths π from s to t?*

Solution: This is there in one of the practice problems posted on Piazza. Solve that if you get time; it also teaches new concepts in C++.

Here, instead of a min-heap, we maintain a **max-heap**. And, $pw[t]$ refers to the "path with the largest value of (the narrowest road width in the path from s to t)". We initialize $pw[s]$ to ∞ and $pw[v] = 0 \quad \forall v \neq s$.

For each edge when it is evaluated, say (u, v), we set

$$pw[v] \leftarrow max(min(pw[u], w(u, v)), pw[v])$$

We traverse the entire graph from s onwards and, at the end, return $pw[t]$.

The proof of correctness is a mirror image of the proof for Dijkstra's, just that all inequalities are reversed, and all sums are replaced by *min*-s. The time complexity is the same, $O(E\ log(V) + V)$.

5. *Same as problem 4. For a path π, define $hop(\pi) = max(d(e))$ for all e in π, with $d(e)$ being its length. Thus, if one is traveling on a motorcycle and if fuel is available only in cities, then $hop(\pi)$ determines the fuel capacity of the tank of your motorcycle needed to undertake the trip. Now, for any s and t, we want to determine the minimum of $hop(\pi)$ for all paths π from s to t. Again, can Dijkstra be used?*

Solution: Yes, the solution is very similar to the previous one. Here, $fc[t]$ refers to the "path with the shortest value of (the longest edge or road in the path from s to t)". And, $fc[s]$ is initialized to 0 and $fc[v] = \infty \quad \forall v \neq s$.

Here, whenever we evaluate an edge (u, v), we set

$$fc[v] \leftarrow min(max(fc[u], d(u, v)), fc[v])$$

We traverse the entire graph using min-heap (based on fc values of unvisited vertices, of course). At the end, we return $fc[t]$.

We leave it as an exercise to use the points, as well as your knowledge of Dijkstra's algorithm, to write the pseudocode for the algorithms for the above problems (3 to 5).