

CS213 2023 Tutorial Solutions

CS 213/293 UG TAs

2023

Contents

Tutorial 1	2
Tutorial 2	7
Tutorial 3	12
Tutorial 4	16
Tutorial 5	22
Tutorial 6	25
Tutorial 7	27
Tutorial 8	32
Tutorial 9	36

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

```
1 for  $j \leftarrow 1$  to  $n - 1$  do
2    $key \leftarrow A[j]$ ;
3    $i \leftarrow j - 1$ ;
4   while  $i \geq 0$  do
5     if  $A[i] > key$  then
6        $A[i + 1] \leftarrow A[i]$ ;
7     end
8     else
9       break;
10    end
11     $i \leftarrow i - 1$ ;
12  end
13   $A[i + 1] \leftarrow key$ ;
14 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 evaluating false only when the iterator i reaches 0. This will happen when the input is in decreasing order.

Counting the operations for the outer iterator j (which goes from 1 to $n - 1$):

1. Each iteration will have 1 comparison, 3 assignments, 2 memory accesses and 1 increment.
2. if-condition will evaluate to true $j - 1$ times (all elements to the left are larger) and false 1 time (along with 1 comparison, 1 assignment and 1 decrement for each inner iteration)
3. For the true case, we have 2 memory accesses and 1 jump; for the false case, we have just 1 jump.

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

$$(C + 4A + 2M) \times (n - 1) + (C + J + A) \times (n \times (n - 1)/2) \\ + (C + A + 2M + J) \times ((n - 1) \times (n - 2)/2) + (J) \times (n - 1).$$

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

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 $\mathcal{O}(1)$ 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 $f(n) = a_0n^0 + \dots + a_dn^d$ and $g(n) = b_0n^0 + \dots + b_en^e$ 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

$$\begin{aligned} \forall n \geq n_0, a_0n^0 + \dots + a_dn^d &\leq cb_0n^0 + \dots + b_en^e \\ \forall n \geq n_0, \sum_{i=0}^e (a_i - cb_i)n^i + a_{i+1}n^{i+1} + \dots + a_dn^d &\leq 0 \end{aligned}$$

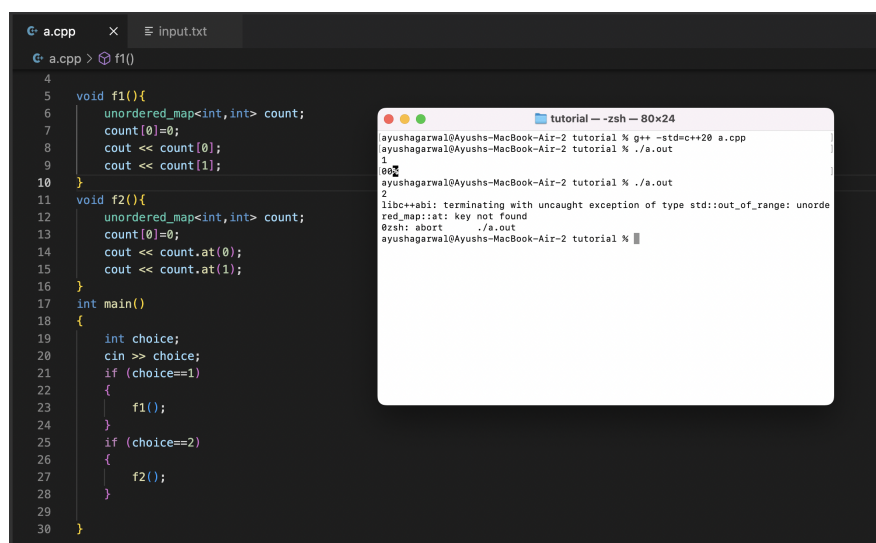
By definition of limit

$$\begin{aligned} \lim_{n \rightarrow \infty} \sum_{i=0}^e (a_i - cb_i)n^i + a_{i+1}n^{i+1} + \dots + a_dn^d &\leq 0 \\ \implies a_d &\leq 0 \end{aligned}$$

Assuming $a_d > 0^2$, this results in a contradiction; thus, our original proposition is proved.

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

Solution: The at function can throw an exception when the accessed element is not in range. The [] operator does not check the range, and the behaviour upon access is undefined when the element is not in range. The at function can throw an exception when the accessed element is not in range. The [] operator calls the default constructor of the value type and then returns the allocated object. This may be considered bad behaviour because a read action causes changes in the data structure, which is undesirable. The map will be filled with many dummy entries if there are many out-of-range reads to the map. On the other hand, throwing exceptions is not ideal from a programmer's perspective. They have to constantly add code that handles exceptions. If such a code is not added, then the exceptions may cause failure of the entire system. See the figure below:



```
G++ a.cpp x input.txt
G++ a.cpp > f1()
4
5
6 void f1(){
7     unordered_map<int,int> count;
8     count[0]=0;
9     cout << count[0];
10    cout << count[1];
11 }
12 void f2(){
13     unordered_map<int,int> count;
14     count[0]=0;
15     cout << count.at(0);
16     cout << count.at(1);
17 }
18 int main()
19 {
20     int choice;
21     cin >> choice;
22     if (choice==1)
23     {
24         f1();
25     }
26     if (choice==2)
27     {
28         f2();
29     }
30 }
```

```
tutorial --zsh -- 80x24
ayushagarwal@Ayushs-MacBook-Air-2 tutorial % g++ -std=c++20 a.cpp
ayushagarwal@Ayushs-MacBook-Air-2 tutorial % ./a.out
1
00
ayushagarwal@Ayushs-MacBook-Air-2 tutorial % ./a.out
2
libc++abi: terminating with uncaught exception of type std::out_of_range: unordered_map::at: key not found
0zsh: abort
ayushagarwal@Ayushs-MacBook-Air-2 tutorial %
```

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 leak occurs when a program allocates some memory and stops referencing it. C++ does not automatically deallocate memory when a program does not reference a part of memory. However, the language supports smart pointers. Smart pointers are classes that count references to an object. If the number of references hits zero, the memory is deallocated. shared_ptr allows one to allocate memory without worrying about memory leaks. unique_ptr is like shared_ptr but it does not allow a programmer to have two references to an address. weak_ptr allows one to refer to an object without having the reference counted. This kind of pointer is needed in case of cycles among pointers. Due to the cycle, the reference counter never reaches zero even if the programs stop referencing the memory. weak_ptr is used to break the cycle. auto_ptr is now a deprecated class. It was replaced by shared_ptr. Please refer to the code snippets shown below

```
main.cpp
1 // auto_ptr_auto_ptr.cpp
2 // compile with: /EHsc
3 #include <memory>
4 #include <iostream>
5 #include <vector>
6 using namespace std;
7
8 class Int
9 {
10 private:
11     bool bIsConstructed;
12     int x;
13 public:
14     Int(int i)
15     {
16         cout << "Constructing " << (void*)this << endl;
17         x = i;
18         bIsConstructed = true;
19     };
20     ~Int()
21     {
22         cout << "Destructing " << (void*)this << endl;
23         bIsConstructed = false;
24     };
25     Int &operator++()
26     {
27         x++;
28         return *this;
29     };
30 };
31
32 int main()
33 {
34     {
35         auto_ptr<Int> pi ( new Int( 5 ) );
36     }
37     {
38         Int* pi2=new Int(5);
39     }
40 }
```

Output

```
/tmp/8Mb3AJHf.o
Constructing 0x1a9beb0
Destructing 0x1a9beb0
Constructing 0x1a9beb0
```

```
G: a.cpp 1 X a.out
G: a.cpp > main()
1 #include <iostream>
2 #include<unordered_map>
3 using namespace std;
4
5
6 int main()
7 {
8     unique_ptr<int> a1;
9     unique_ptr<int> a2=a1;
10
11 }
```

tutorial -- -zsh -- 80x24

```
ayushagarwal@Ayushs-MacBook-Air-2 tutorial % g++ -std=c++20 a.cpp
a.cpp:9:21: error: call to implicitly-deleted copy constructor of 'unique_ptr<int>'
    unique_ptr<int> a2=a1;
                        ^
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/c++/v1/__memory/unique_ptr.h:213:3: note: copy constructor is implicitly deleted because 'unique_ptr<int>' has a user-declared move constructor
    unique_ptr(unique_ptr&& __u) _NOEXCEPT
    ^
1 error generated.
ayushagarwal@Ayushs-MacBook-Air-2 tutorial %
```

```
G: a.cpp X a.out
G: a.cpp > main()
1 #include <iostream>
2 #include<unordered_map>
3 using namespace std;
4
5
6 int main()
7 {
8     shared_ptr<int> a1;
9     shared_ptr<int> a2=a1;
10
11 }
```

tutorial -- -zsh -- 80x24

```
ayushagarwal@Ayushs-MacBook-Air-2 tutorial % g++ -std=c++20 a.cpp
ayushagarwal@Ayushs-MacBook-Air-2 tutorial % ./a.out
ayushagarwal@Ayushs-MacBook-Air-2 tutorial %
```

```
a.cpp x a.out
a.cpp > main()
1 #include <iostream>
2 #include <memory>
3
4 std::weak_ptr<int> gw;
5
6 void observe()
7 {
8     std::cout << "gw.use_count() == " << gw.use_count() << " ";
9     // we have to make a copy of shared pointer before usage:
10    if (std::shared_ptr<int> spt = gw.lock())
11        std::cout << "spt == " << *spt << '\n';
12    else
13        std::cout << "gw is expired\n";
14 }
15
16 int main()
17 {
18     {
19         auto sp = std::make_shared<int>(42);
20         gw = sp;
21
22         observe();
23     }
24
25     observe();
26 }
```

```
tutorial --zsh -- 80x24
ayushagarwal@Ayushs-MacBook-Air-2 tutorial % g++ -std=c++20 a.cpp
ayushagarwal@Ayushs-MacBook-Air-2 tutorial % ./a.out
gw.use_count() == 1; *spt == 42
gw.use_count() == 0; gw is expired
ayushagarwal@Ayushs-MacBook-Air-2 tutorial %
```

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) 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 2 5 6.

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

Solution: The idea in this problem is to check the history till the i^{th} day and compute the span based on the blocks of past sets of consecutive days. Lets define a **valid stretch** to be "a stretch of consecutive days so far where the price never exceeded the current price". The invariant that we need to maintain here is:

The span of the current day is the longest valid stretch

To do this, at the position i , we make a linear pass over the prices till i^{th} day and do the following:

- we check all the positions where the price did not exceed the current price and take the maximum value out of these
- we also maintain a counter for the length of the most recent valid stretch

Based on the above computations, we can conclude that the span of the i^{th} day is the maximum of the span value obtained in linear iteration (maximum among spans of past days with non-exceeding prices) and the length of the most recent valid stretch (including the current day).

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.

Alternate interpretation: This is the more common version of the problem, where we start from current day and go backwards.

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

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.

Both implementations can be found [here](#).

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 $sz - 1$, i.e., 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 $n = \alpha^i$ consecutive pushes. So, total cost of expansion is (refer to slides for detailed explanation):

$$\begin{aligned}
 &(\alpha + 1) \cdot (\alpha^0 + \alpha^1 + \alpha^2 \dots + \alpha^{i-1}) \\
 &\quad \frac{\alpha + 1}{\alpha - 1} \cdot (\alpha^i - 1) \\
 &\quad \frac{\alpha + 1}{\alpha - 1} \cdot (n - 1)
 \end{aligned}$$

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* \rightarrow *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* \rightarrow *next* = *prev*
- *prev* = *curr*
- *curr* = *next*
- *next* = *next* \rightarrow *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. *The mess table queue problem: There is a common mess for k hostels. Each hostel has some N_1, \dots, N_k students. These students line up to pick up their trays in the common mess. However, the queue is implemented as follows:*

If a student sees a person from his/her hostel, she/he joins the queue behind this person.

This is the “enqueue” operation. The “dequeue” operation is as usual, at the front. Think about how you would implement such a queue. What would be the time complexity of enqueue and dequeue?

Do you think the average waiting time in this queue would be higher or lower than a normal queue? Would there be any difference in any statistic? If so, what?

Solution: First, note that such a queue would consist of at most K sub-queues, each one containing students from the same hostel. New students who show up to join the queue get attached to the end of one of these hostel sub-queues.

We implement this with the following:

- A global linked list of all students currently in the queue (each student is a node in the list). This linked list is initially empty.
- Pointers head and tail, to the front and end of the queue respectively. Both are initially null.
- An array of pointers sqend of size K . sqend[i], for $0 \leq i \leq K-1$, has a pointer to the last student in the sub-queue of hostel $i + 1$, if that sub-queue currently exists in the queue; otherwise, sqend[i] is null. (The hostel number is taken as $i + 1$ here and not i only because of 1-indexing for hostel numbers but 0-indexing in arrays.) All sqend[i]'s are initially null. Note that enqueueing requires pointers to all the sub-queue ends, because it may occur at a number of places in the queue, whereas dequeueing only requires a single pointer at the front.

Note that enqueueing requires pointers to all the sub-queue ends, because it may occur at a number of places in the queue, whereas dequeueing only requires a single pointer at the front.

Enqueueing: When a new student s shows up to join the queue:

- Obtain the hostel number h of s . This is a constant $\mathcal{O}(1)$ time operation if the hostel numbers are stored within the student nodes themselves.
- If sqend[h-1] is not null, i.e., there is a sub-queue for hostel h in the current queue, then insert s in the mess queue, immediately after the student to which sqend[h-1] points.
- If sqend[h-1] is null, then there is no sub-queue for the student's hostel, and he/she must be attached at the end of the entire queue. Using the tail pointer, insert s at the end. It may so happen that tail is null too. This means the entire queue is empty; in this case, simply make both head and tail point to s .
- If s has been inserted at the end of the queue, then we have to change tail to point at s . This should be done if either sqend[h-1] is null (no sub-queue already present for s), or sqend[h-1] and tail point to the same student (sub-queue for s present at the end).
- Update sqend[h-1] to point at s .

All the above operations take constant $\mathcal{O}(1)$ time, so enqueueing is a constant time operation.

Dequeueing: Dequeueing a student only happens at the front:

- Use the head pointer to obtain the student s to be removed, and his/her hostel number h .
- Change head to point to the student immediately after s in the queue. If there are no more students in the queue, head becomes null.
- If the new head of the queue is either null, or has a hostel number different from h , that means the sub-queue of hostel h no longer exists after dequeueing s ; set $sqend[h-1]$ to null.
- If the new head of the queue is null, i.e., it's empty, set tail to null too.

Now, think about the average waiting time in such a queue:

$$\text{average waiting time} = \frac{\text{sum of waiting times for all students in queue (total load)}}{\text{total number of students in queue}}$$

We make the (reasonable) assumption that dequeueing at the front of the queue takes place at a constant rate, say 1 person per unit time. Then, the waiting time for any student in the queue is equal to the number of students in front of him/her

- In an ordinary queue of size n , a newcomer gets attached at the end. Then, the waiting time for the newcomer is n , and the waiting times of all the other students remain unchanged. The sum of waiting times (total load of the queue) increases by n .
- In a mess queue of size n which operates as in this problem, a newcomer may get attached at one of many positions in the queue. Suppose the newcomer is inserted after the first k students in the queue. Then, the waiting time for the newcomer is k , and there is an increase of 1 in the waiting times of all the $n - k$ students behind the newcomer. Thus, the increase in sum of waiting times in the queue is $k + (n - k) = n$.

Because the increase is the same in both cases, it can be easily seen that the average waiting time would also be the same; that is, the average waiting time in a queue is not influenced by the arrival process distribution. For more on this, see Little's Law.

Tutorial 3

1. 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: We assume that the hash function is perfect - that is, it hashes each element to a uniformly random element of the target space. The probability of such a hash function not incurring a collision is simpler - every element hashes to a different element of the space: the probability is simply

$$q = \frac{k! \binom{n}{k}}{n^k} \quad (1)$$

The probability of a collision is then $p = 1 - q$.

Alternately: We write the probability as a product of conditionals, basically, imagine picking the objects one by one. The probability of no collision when the i th element is inserted is $(n - (i - 1))/n$ since the previous $(i - 1)$ elements occupy different places. The probability of no collision is then the product of these numbers -

$$q = 1 \cdot \left(1 - \frac{1}{n}\right) \cdot \left(1 - \frac{2}{n}\right) \cdots \left(1 - \frac{k-1}{n}\right) = \frac{k! \binom{n}{k}}{n^k} \quad (2)$$

as before.

For the second question, one notes that if a collision happened during the insertion of the i th of N elements, there is a collision even at the end. And if there was no collision throughout the insertion, there will not be any at the end too. Thus the probability of a collision occurring during insertion is simply the probability above with $k = N$.

We estimate it by bounding it suitably from above and below. On the one hand, since $1 - x \leq e^{-x}$ for $x \geq 0$, we get from Equation (2) that

$$q \leq \exp\left(-\sum_{i=0}^{k-1} \frac{i}{n}\right) = \exp\left(-\frac{k^2 - k}{2n}\right)$$

so that

$$p \geq 1 - \exp\left(-\frac{k(k-1)}{2n}\right)$$

If $k < \sqrt{2n}$, the term in the exponent is less than 1 and we can use $1 - e^{-x} \geq x/2$ (holds good for $0 \leq x \leq 1$ to deduce

$$p \geq \frac{k(k-1)}{4n}$$

Next, to upper-bound the probability. Let A_{ij} denote the event that elements i and j collided. It is easy to see $P(A_{ij}) = 1/n$. The probability, then, of a collision existing is

$$p = P\left(\bigcup_{1 \leq i < j \leq n} A_{ij}\right) \leq \sum_{1 \leq i < j \leq n} P(A_{ij}) = \frac{\binom{k}{2}}{n} = \frac{k(k-1)}{2n}$$

Finally, we get the desired bounds: Let $\lambda := \frac{k(k-1)}{2n}$. Then

$$\lambda/2 \leq p \leq \lambda$$

or $p = \Theta(\lambda)$.

Remark. This is, in essence, the **birthday problem**.

2. Let $C(i)$ be the chain of array indices that are queried to look for a key k in linear probing where $h(k) = i$.
1. How does this chain extend by an insertion, and how does it change by a deletion?
 2. 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?
 3. Is there a way of not using tombstones?

Solution: Clearly, $C(i)$ is precisely the set of cells starting at i and ending with the first empty cell. Let $C'(i)$ denote the new chain starting at i after an insert/delete.

1. Say k was inserted into the table at position t . If t is the cell at the end of $C(i)$, $C'(i) = C(i) \cup C(t+1)$. If not, we stop looking for hash value i at the (empty) end of $C(i)$ itself, so $C'(i) = C(i)$.

In the case of a deletion, search indices do not reduce because we always move over tombstones as though they were actual keys. Thus $C'(i) = C(i)$.

2. Yes, we could make it work. We get rid of tombstones, and instead of looking for an empty cell to end our search, we stop when we reach an end marker. We do not use tombstones. An end marker is always empty.

- When we need to insert an element, we reach this end during probing and insert the element there. **If the next element is also empty, we put an end marker there. Otherwise, we leave it as is.**
- During searches, we search till we reach the desired key or an end marker.
- During deletion (of a key with hash value j , say), we find the element and turn the cell into an empty cell. **We then iterate backwards till j until to find an element with hash value j , maintaining the index of the latest empty spot we found. We stop when we locate such an element or reach j , and we put up an end marker in the latest empty spot found.**

The bold text describes how we maintain the invariant that an end marker describes the end of a chain and that we search till we reach an end marker.

Efficiency: Deletions are slower due to iteration over a chain. But there are no tombstones, and so searching is faster. Thus, if searches are priority with not many deletes, this may prove faster.

3. The method of end markers described above is one such way. Another is as follows: Suppose we delete key k with hash value j . When we empty a cell in chain $C(j)$ (on deletion), we "reel in" the rest of the chain $C(j)$. We search ahead of the now-empty cell in the cluster for the first key k' that has the same hash value j and move that into our empty cell. That leaves us with another empty cell, and repeat the process (we search ahead, and try to find a key k'' , and so on) till we get to the end of the chain. We return here and the last cell that became empty can remain empty (why does this work?). This has linear ($\mathcal{O}(m)$) complexity in the worst case for deletion, but is generally faster. Of course, later searches are faster because there are no tombstones lying around.

Remark. Tombstones are clearly useful in fast hash table implementations. To get partially around the problem of search with too many tombstones, what is done is the following: Once α (the load factor) becomes large (we count the tombstones too), we remove all the elements, empty the array and then rehash and insert the elements into the array.

3. Let $m = 11, h_1(k) = (k \bmod 11), h_2 = 6 - (k \bmod 6)$
 Let us use the following hash function for an open addressing scheme.

$$h(k, i) = h_1(k) + i \times h_2(k)$$

What will be the state of the table after the following sequence of insertions?

41, 22, 44, 59, 32, 31, 74

Solution: Let us trace the execution of inserting the above elements into a hash table.

- For element 41, we have the hash index as $(41 \bmod 11)$ which equals 8 (unused), hence 41 gets stored at index 8.
- For element 22, we have the hash index as $(22 \bmod 11)$ which equals 0 (unused), hence 22 gets stored at index 0.
- For element 44, we have the hash index as $(44 \bmod 11)$ which equals 0 (occupied). Hence, our update step is $6 - (44 \bmod 6)$ which equals 4. Hence, we check $(44 \bmod 11) + 4 = 4$ (unused) and thus, 44 is stored at index 4.
- For element 59, we have the hash index as $(59 \bmod 11)$ which equals 4 (occupied). Hence, our update step is $6 - (59 \bmod 6)$ which equals 1. Hence, we check $(59 \bmod 11) + 1 = 5$ (unused) and thus, 59 gets stored at index 5.
- For element 32, we have the hash index as $(32 \bmod 11)$ which equals 10 (unused), hence 32 gets stored at index 10.
- For element 31, we have the hash index as $(31 \bmod 11)$ which equals 9 (unused), hence 31 gets stored at index 9.
- For element 74, we have the hash index as $(74 \bmod 11)$ which equals 8 (occupied). Hence, our update step is $6 - (74 \bmod 6)$ which equals 4. Hence, we check $(8 + 4) \bmod 11$ which equals 1 (unused), hence 74 gets stored at index 1.

The final table looks like this

22, 74, —, —, 44, 59, —, —, 41, 31, 32

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

Solution: Idea is to use double-hashing. Instead of hashing just on basis of name or address, we will use both of them to hash. Consider a 2-d array $arr[i][j]$, where i represents the index corresponding to hash value of name and j represents index corresponding to hash value of address. We will use the technique of chaining in the case of collisions. All the steps will be same as that of one-dimension, just the hash value has changed to a tuple. Time complexity of various operations will be:

- Insertion : $\mathcal{O}(1)$
- Deletion : $\mathcal{O}(1 + \alpha)$

- Search : $\mathcal{O}(1 + \alpha)$

For a particular name or address, we will iterate over the corresponding row or column accordingly and check for all the elements for counting or deleting all entries.

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 $0, \dots, r - 1$.

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 $Seq(T) = \{Seq(w) \mid w \in T\}$ 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: Let $p_0 = 0$ be root of tree

$$w = p_0 p_1 p_2 \dots p_n$$

$$S_w = \{x \mid x = p_0 p_1 p_2 \dots p_i q \quad \& \quad 0 \leq i \leq n - 1 \quad \& \quad 0 \leq q \leq p_{i+1}\} \cup \{0\}$$

We can prove S_w is the set which is guaranteed to be present if w is present by following two arguments,

- 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 $l_0 l_1 l_2 \dots l_n$ is present then all the words of form $l_0 l_1 l_2 \dots l_{n-1} q$ where $0 \leq q \leq l_n$ will also be present since these are just the child of the node represented by sequence $l_0 l_1 l_2 \dots l_{n-1}$
- There exist a tree T st. that $Seq(T) = w \cup S_w$. This is the tree where p_i is no of children of node represented by sequence $p_0 p_1 p_2 \dots p_{i-1}$. All nodes whose sequence are not of the form $p_0 p_1 p_2 \dots p_i$ are leaf nodes

Let A_w denotes no of nodes in the sub-tree considering w as the node. Then w appears as a prefix for A_w no of words in $Seq(T)$

Some properties are $Seq(T)$ are

1. Size of this set will be n (Trivial!!!)
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.

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

Solution: The definition of $LCA(n_1, n_2, T)$ is a node in $ancestors(n_1) \cap ancestors(n_2)$ that has the largest level. This function can be written using a recursive definition by breaking into several cases and returning the appropriate answers. The cases can be broken as follows:

- if the root is *NULL* then we return *NULL*
- if the root is one of the nodes then the root is the answer (the other node will surely have the root as an ancestor)
- we recursively compute the LCA of these nodes from the left and right sub-trees (more precisely $L = LCA(v, w, T \rightarrow left)$ and $R = LCA(v, w, T \rightarrow right)$)
- if both L and R are not *NULL* then the root is the largest level ancestor
- otherwise the one which is not *NULL* is the LCA (*NULL* if both are *NULL*)

Explanation: The first two cases are fairly obvious. If the tree does not exist, there is no LCA. The LCA for root and any other node is the root itself. The recursive computation allows us to search for the largest common ancestor in the sub-trees attached to the root. If both the left and right child have the LCA of the sub-trees (think of it as both being part of the set of ancestors of both the given nodes), then the root is also an ancestor for both nodes. Otherwise, the largest level ancestor will be the LCA of the sub-tree that exists. If either of them does not exist, there is no common ancestor.

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 $f(\text{root}(T))$. What is the time complexity of the program?

Algorithm 2: Computing $f(n)$

Data: Tree T having n nodes

```

1 ModifiedPostOrderWalk( $p$ )
2  $f \leftarrow [1, 0, \dots, 0]$ 
3 if  $p.\text{isLeaf}()$  then
4    $\mid$  return  $f$ ;
5 for  $n' \in \text{children}(p)$  do
6    $f_c \leftarrow \text{ModifiedPostOrderWalk}(n')$ 
7   for  $j \leftarrow 1$  to  $n - 1$  do
8      $\mid$   $f[j] \leftarrow f[j] + f_c[j - 1]$ ;
9 return  $f$ ;

```

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 $\text{depth}(n)$ 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 $f(n)[i] = \sum_{m \in \text{children}(n)} f(m)[i - 1]$.
Recursive equation of $f(n)$ then can be written as:

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

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

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

Pseudocode:

As you all know, we will be doing some kind of traversal to get $f(\text{root}(T))!!!$

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 $[1, 0, \dots, 0]$, 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.

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 $f = (f_1, f_2, \dots, f_n)$ and postorder walk be $l = (l_1, l_2, \dots, l_n)$

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 $f' = (f_2, f_3, \dots, f_{k-1})$ as preorder walk of left subtree of r and $f'' = (f_k, f_3, \dots, f_n)$ 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 $l' = (l_1, l_2, \dots, l_k)$ as postorder walk of left subtree of r and $l'' = (l_{k+1}, l_{k+2}, \dots, l_{n-1})$ 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. For a given binary tree, let $prevloc(T, n)$ give the node n' such that $label(n')$ will appear just before $label(n)$ in the inorder printing of T . Give a program to compute $prevloc$.

Solution: The algorithm for in-order traversal of a tree is provided below as Algorithm 3

Observations:

1. The first node l whose label gets printed in inorder traversal is the leftmost node of the tree.
2. The last node r whose label gets printed in inorder traversal is the rightmost node of the tree.

Let's make cases on left child of node n

1. $n \rightarrow$ left child exists.

Claim: The rightmost node of left subtree of n is the required node

Proof: According to definition, $prevloc(T, n)$ is the node m such that $label(m)$ will be printed just before $label(n)$. According to algorithm 3, $label(n)$ gets printed after label of all nodes of left subtree of n gets printed and thus the node l in left subtree of n whose label gets printed last during inorder traversal should be $prevloc(T, n)$ and according to **observation 2**, this is the rightmost node of left subtree of node n .

2. $n \rightarrow$ left child doesn't exist.

- (a) There doesn't exist any ancestor a of n , (a may or may not be n) such that a is right child of its parent p i.e. all ancestors of n are left child of their parent if their parent exists.

Claim: $prevloc(T, n)$ doesn't exist

Proof: According to **observation 1**, then n is the **first** node whose label gets printed and thus $prevloc(T, n)$ doesn't exist.

- (b) There exists an ancestor a of n , (a may or may not be n) such that a is right child of its parent p and then according to well-ordering principle there exists an ancestor b such that it is closest (in terms of distance or number of edges) to n and is right child of its parent q or in simple words b is the ancestor of n which is right child of its parent and any ancestor of n strictly in between n and b are left child of their parent

Claim: The node q is the required node

Proof: When algorithm 3 is called on node q , it first calls algorithm 3 on its left subtree, then prints $label(q)$ and then calls algorithm 3 on its right subtree.

Can you think of the node which will get its label printed first in the right subtree R of q ?

So the answer is simple, the leftmost node of R , right?

Which node is it?

It is our node n , why?

Consider the subtree R as new tree T' then node n in T' has all its ancestors left child of their parent and thus from **Case 2(a)**, it is the first node to get printed when algorithm 3 is called on T' or equivalently R and hence q is $prevloc(T, n)$.

Time Complexity

In the worst case, you will have to go either to the root or to a leaf, and thus time complexity comes out to be $O(h)$ where h is height of tree or we can write $O(\log(n))$ in case our tree is balanced.

Conclusion

We have covered all the cases, try to write a C++ program for it as well as the following fun problem.

Fun problem: For a given binary tree, let $\text{nextloc}(T, n)$ give the node n' such that $\text{label}(n')$ will appear just after $\text{label}(n)$ in the in-order printing of T . Give a program to compute nextloc .

Algorithm 3: InOrderTraversal(n)

Data: Node n

```
1 InOrderTraversal( $n$ )
2 if  $n$  is not null then
3   InOrderTraversal( $n \rightarrow \text{left child}$ );
4   visit( $n$ );
5   InOrderTraversal( $n \rightarrow \text{right child}$ );
```

Tutorial 5

1. (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.

2. Given a BST T and a key k , the task is to delete all keys $b < a$ 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 provided deletes all keys $b < a$ from T and returns the root of the modified BST.
- (b) The complexity of the procedure is $\mathcal{O}(\log n)$. This is because we are moving from the root to a leaf of the tree in worst case. So, the worst-case complexity is proportional to the height of the tree.
- (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 4: Algorithm For Q2

Data: Node *node*, value *k*
Result: Updated node or *null*

```
1 if node is null then
2   | return null
3 end
4 if node.key ≥ k then
5   | node.left ← Algorithm(node.left, k);
6   | return node
7 end
8 else
9   | return Algorithm(node.right, k)
10 end
```

3. 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: First, we make a few relevant observations

- The expected height of a tree obtained by inserting a random permutation of $[n]$ equals the expected height obtained by inserting a random permutation of any n natural numbers. This stems from the fact that we can create a bijection between this set of natural numbers and the first n natural numbers, the insertion of which will result in the same tree structure. Think what this bijection will be.
- The height of a binary tree equals

$$H(T) = 1 + \max(H(T - \text{right}), H(T - \text{left}))$$

where $H(T)$ represents the height of a tree with root T .

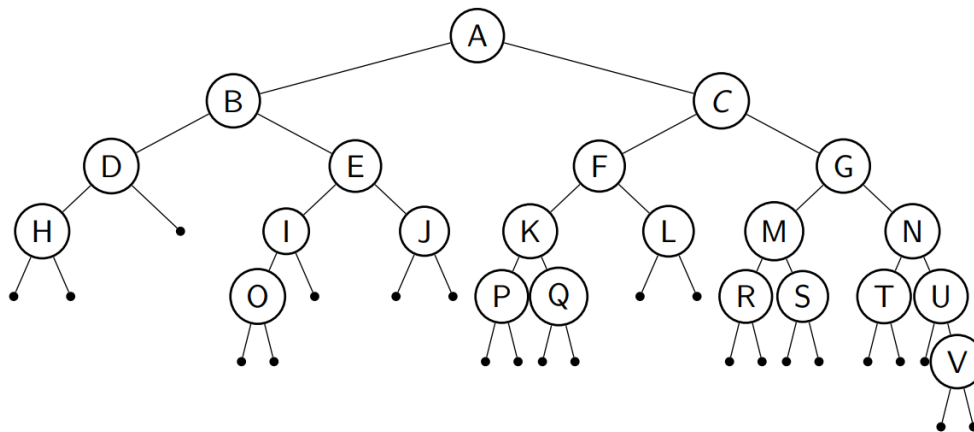
- In a random permutation of $[n]$, the first element can be any $k \in [n]$ with equal probability $\left(\frac{1}{n}\right)$
- Let $K(x)$ denote the probability mass function of the height of a tree with x randomly permuted values.

Hence, we can now write the recursion for $H(n)$

$$H(n) = 1 + \frac{1}{n} \sum_{i=1}^n E \max(K(i-1), K(n-i))$$

Base Case: $H(0) = 0$

4. Consider the tree below. Can it be coloured and turned into a red-black tree? If we wish to store the set $1, \dots, 22$, 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

Tutorial 6

1. Compute array h for pattern "babbaabba".

Solution: For each j , $h[j]$ is the length k of the longest proper prefix $P[0..k-1]$ such that

- (a) $P[0..k-1]$ is also a suffix of $P[0..j-1]$. That is, $P[0..k-1] = P[j-k..j-1]$.
- (b) $P[j] \neq P[k]$.

By convention, condition (a) vacuously passes when $k \leq 0$ and condition (b) vacuously passes when $k = -1$ (equivalent to setting $p[-1] = \text{null}$). Thus if no such $k \geq 0$ exists, we set $h[j] = -1$ (why? Think about how much you can shift in the KMP algorithm).

Note that h is one-indexed, but P is zero-indexed. We also set $h[0] = -1$ by convention. The table is as follows:

j	0	1	2	3	4	5	6	7	8	9
$P[j]$	b	a	b	b	a	a	b	b	a	∅
		b	a	b	b	a	a	b	b	a
			b	a	b	b	a	a	b	b
				b	a	b	b	a	a	b
					b	a	b	b	a	a
						b	a	b	b	a
							b	a	b	b
								b	a	b
									b	a
										b
$h[j]$	-1	0	-1	1	0	2	-1	1	0	2

Let's see how to evaluate each $h[j]$. First, write out a table with the columns $0..|P|$, and then all the prefixes of P , right-aligned (to match them with suffixes more easily). Now, pick your favourite $j \in \{0, \dots, |P|\}$. Go over to the j th column, and search for rows whose j th column has a character different from $P[j]$ (to satisfy condition (b)). Then, for each of those rows, match the prefix of that row with the suffix of P ending at $j-1$ just above it. If it matches, the length of the prefix (excluding j) is $h[j]$. See an example, marked for $j = 5$ in red.

2. Is the following version of *KMPtable* correct?

Algorithm 10.3: KMPtableV2(string P)

$i := 1; j := 0; h[0] := -1;$

while $i < |P|$ **do**

$h[i] := j;$

while $j \geq 0$ and $P[j] \neq P[i]$ **do**

$j := h[j];$ // Moving forward the pattern in minimum steps as in KMP

$i := i + 1; j := j + 1;$

$h[|P|] := j;$

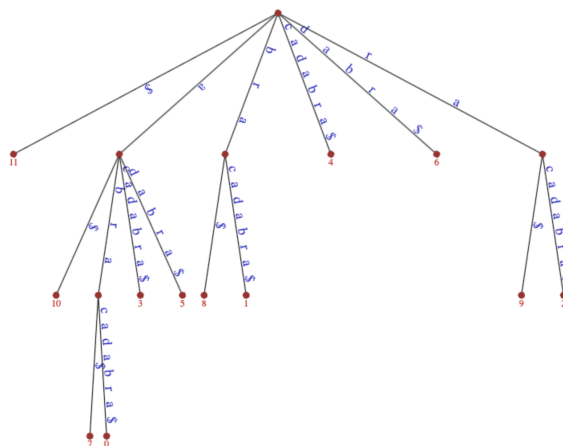
return h

Solution: Yes. The h table is constructed exactly like running a KMP of P over P (think!), and thus this will work.

3. Suppose that there is a letter z in P 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: Essentially, the character z splits the KMP computation into two parts which can be done *parallelly*. Suppose that $z = P[k]$, and $p = p_1zp_2$. It is clear that for every $i > k$, any prefix of p that is also a suffix of $p[0]...p[i]$ cannot have size larger than k , since otherwise it would necessitate that z occur somewhere after k too. Thus, one simply needs to match p_1z, p_2 with p_1 to compute the complete h -table. This can be achieved by computing both parallelly, since the two computations are independent.

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:

Using the given suffix tree, we can find the 2 occurrences of "abr", one as the prefix of suffix "abra\$" and other as prefix of suffix "abracadabra\$"

Credits: [here](#)

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 $\mathcal{O}(|T|^2)$? Give a worst-case example.

Solution:

1. In a ordinary Trie, each character in the text T can correspond to at-most one node in the trie structure. Also, no. of edges can be atmost T one corresponding to each character in the text T , joining each character to it's next character. Hence, trie can be stored in space complexity of $\mathcal{O}(|T|)$
2. Suffix tree can require $\mathcal{O}(|T|^2)$ space because we may have to make k nodes corresponding to a character that is at k^{th} position from the start. Example $a^n b^n$, it will require $(n + 1)^2$ nodes excluding \$ (For proof [here](#))

Tutorial 7

1. 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. 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.

2. Suppose we have a 2D 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.

3. (a) In a Huffman code instance, show that if there is a character with a frequency greater than $\frac{2}{5}$ then there is a codeword of length 1.

Solution: Let us assume, without loss of generality, that character X has a frequency of at least $\frac{2}{5}$ and has the highest frequency among all characters. For it to not have an encoding of size 1, it must get merged with a character or composite character, not at the final step. Hence, when character X gets merged, there must be at least three characters or composite characters left over and X must be amongst the smallest 2 frequencies. Additionally, the composite character(s) having frequency greater than that of X must not have a frequency greater than twice that of a character. This is not possible - hence contradiction

(b) Show that if all frequencies are less than $\frac{1}{3}$ then there is no codeword of length 1.

Solution: Let us assume the frequency of all characters are under $1/3$. Let us assume the character wlog X, having the highest frequency, has an encoding of size 1 (if we chose any other character, we get a contradiction on the optimality of the encoding). In such a case, we will have a composite character having frequency greater than $2/3$ being merged with a character having frequency less than $1/3$. This is not possible, as the composite character that was formed would have been formed if and only if both components had frequency lesser than $1/3$, which would not add up to a frequency greater than $2/3$. Hence, proved by contradiction.

4. 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 or b , then the next output will either be a or 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: The Huffman encoding will end up encoding each character using 1 bit (most common) and 2 bits (for the other two). This will take greater than 1 bit to encode a character on average.

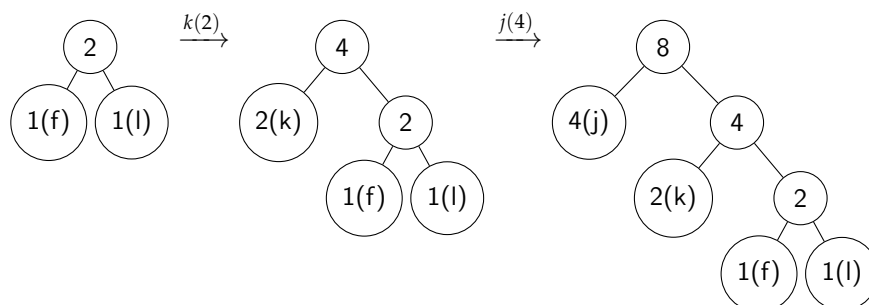
We ask ourselves the natural question. Can we do better? We can express the starting letter using 2 bits (00 - a , 01 - b and 11 - c) followed by a 0 if the current letter is the same as the previous letter and 1 if it is different.

5. Given the following frequencies, compute the Huffman code tree.

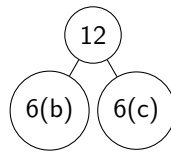
a	d	g	j	b	e	h	k	c	f	i	l
20	7	8	4	6	25	8	2	6	1	12	1

Solution: We first arrange the characters in decreasing order of frequencies: $e, a, i, g, h, d, b, c, j, k, f, l$. When we merge two characters, we can treat it as if we get a new composite character which has frequency as the sum of the frequency of the two characters. The tree will be built as follows:

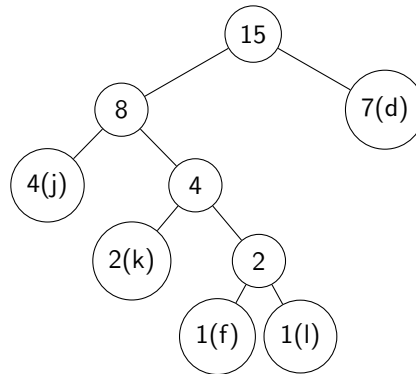
First sub-tree: Starting from f (1) and l (1) (the least two frequencies).



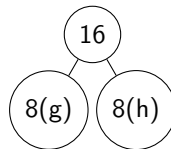
Second sub-tree: Starting from b (6) and c (6) (the new least two frequencies).



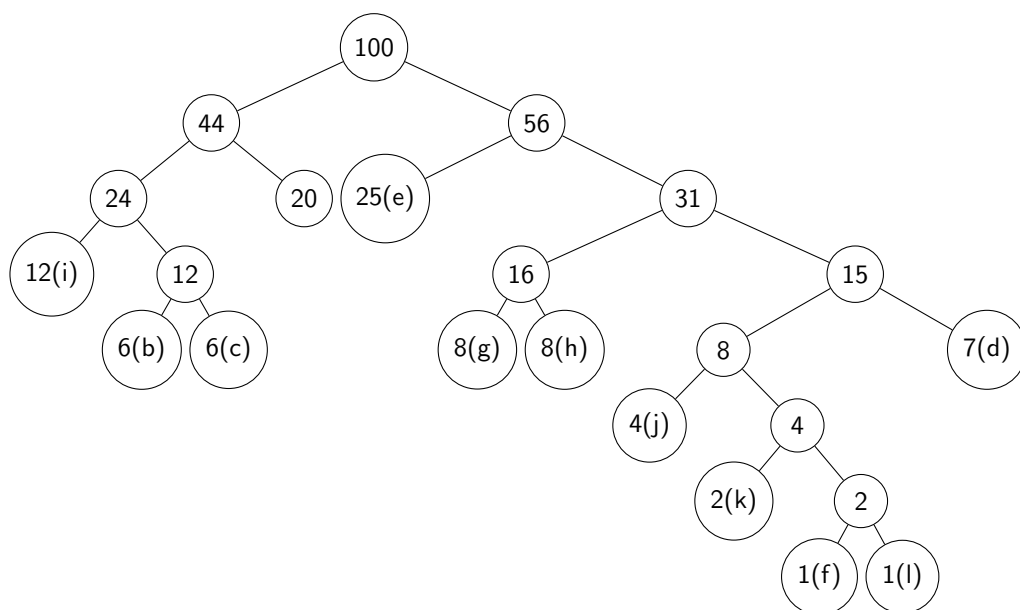
Adding d (7) to first sub-tree (8).



Third sub-tree: Starting with g (8) and h (8) (the new least two frequencies).



Proceeding like this and merging the least two frequencies into a tree, we get the complete Huffman code tree. Naturally, it is not unique as the choice of the least two frequencies need not be unique. Below is one Huffman code tree (completing the remaining merges).



6. (a) Modify the partition such that it detects that the array is already sorted.
 (b) Can we use this modification to improve quick sort?

Solution: Do the following modifications to the partition algorithm mentioned in slides:

- Make a new variable *isSorted* and initialize it to true.
- Make two new variables lb and ub initialized to $A[l]$ and $A[u]$ respectively.
- while increasing i, if $A[i + 1] < lb$, make *isSorted* False or else update lb to $A[i + 1]$
- while decreasing j, if $A[j - 1] > ub$, make *isSorted* False or else update ub to $A[j - 1]$
- If anytime condition of swap becomes True make *isSorted* False.

This can help in improving quicksort since anytime we know that array is sorted, we don't need to any further call sub-parts recursively, we can just return the array.

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

Solution: Radix sort is a **stable sort**. So, the only modification we need to do on going from right to left is we need to call the RadixPartition function on the entire array instead of on the two sub-parts since priority is decided by the left bits.

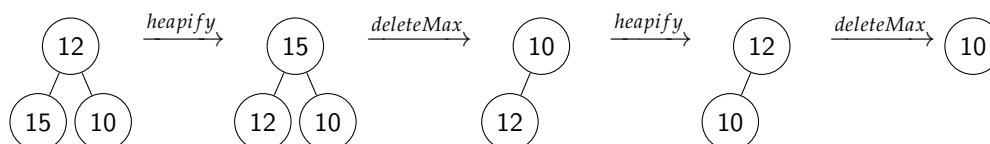
Algorithm 5: *RadixSortRec*(A, b, l, u)

```

 $k \leftarrow 0$ 
while  $k < b$  do
  RadixPartition( $A, k, l, u$ )
   $k \leftarrow k + 1$ 
end while
  
```

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

Solution: Let us take the array [12, 15, 10]. We first create a binary tree (level-wise) out of it. The heapify procedure is $\mathcal{O}(\log(n))$ and deleteMax is constant time for a max-heap.



The last node is just the base case ($n = 1$). We can call deleteMax directly.

9. Prove that $\mathcal{O}(\log(n!)) = \mathcal{O}(n \log n)$.

Solution: First, observe that to prove $a_n = \mathcal{O}(b_n)$, we need to show that we can find a c and n_0 such that $a_n \leq cb_n$ (or basically $a_n/b_n \leq c$ if $b_n > 0$) for all $n \geq n_0$.

To show that the two functions $f(n) = \log(n!)$ and $g(n) = n \log n$ have the same limiting/asymptotic behaviour, we need to show that (1) $f(n) = \mathcal{O}(g(n))$ and (2) $g(n) = \mathcal{O}(f(n))$. Note that both functions are always positive (for all $n > 1$).

(1) $\log(n!) = \mathcal{O}(n \log n)$. Consider $\frac{\log(n!)}{n \log n}$. This reduces to a form that can be bounded easily, using $c = 1$. The proof is as follows.

$$\begin{aligned} \frac{\log(n!)}{n \log n} &= \frac{\log(n \times n-1 \times \cdots \times 1)}{n \log n} = \frac{\log n + \log(n-1) + \cdots + \log(1)}{n \log n} \\ &< \frac{\log n + \log n + \cdots + \log n}{n \log n} = 1. \end{aligned}$$

(2) $n \log n = \mathcal{O}(\log(n!))$. The upper bound for $\frac{n \log n}{\log(n!)}$ is a bit non-trivial to find (it can be shown that a constant c will always exist for all $n \geq n_0$). Instead, we can use an existing result which is Stirling's approximation for an expression of $\log(n!)$ that is easier to handle. So we use $\log(n!) = n \log n - n + \mathcal{O}(\log n)$. Given this, we can directly conclude that $n \log n = \log(n!) + n - \mathcal{O}(\log n) = \mathcal{O}(\log(n!))$ (dominating function).

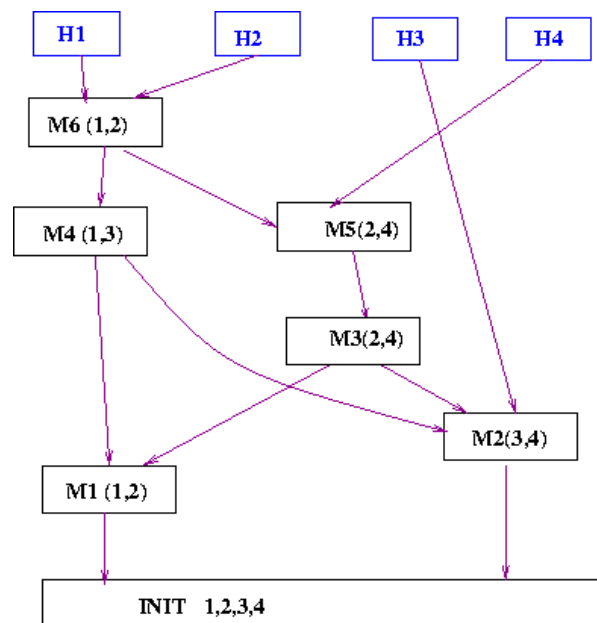
In fact, using Stirling's approximation, we can directly conclude that $\mathcal{O}(\log(n!)) = \mathcal{O}(n \log n)$. You can refer to the [Wikipedia page](#) for the details of it.

Tutorial 8

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 E_1 if their names appear on the same webpage and (ii) in E_2 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 E_1 , 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 E_2 , 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 E_2 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 $V = \{[M_i, S_i] | i\text{-th meeting label, } S_i = \text{people who participated}\}$. 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 $\text{header}(p)$ 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 6: Insert Function

```

1 Function Insert( $k, S_k$ ):
2   New meeting  $m$ ;
3    $m.S \leftarrow S_k$ ;
4    $m.id \leftarrow k$ ;
5    $m.next \leftarrow []$ ;                                // Adjacency list
6   foreach  $p$  in  $S_k$  do
7      $m.next \leftarrow \text{append}(m.next, \text{header}(p))$ ;
8    $\text{header}(p) \leftarrow m$ ;

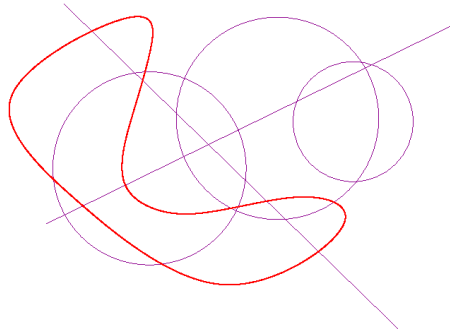
```

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

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 can prove this by induction on the number of objects (say n). Base Case ($n = 1$): Either a line or a circle exists which divides the plane into 2 regions adjacent to each other ($K_{1,1}$ is obtained).



Inductive Hypothesis: The graph obtained using n objects is bipartite. Proof: Let there be an arbitrary graph G obtained using any $n + 1$ objects. Pick any object and remove it. Let the resulting graph be G' . We know by inductive hypothesis that G' is bipartite i.e. it doesn't contain any odd length cycle. Now add the $(n + 1)^{st}$ object (say p) to get G . This object might be a line or a circle. Nevertheless, it will divide the plane into 2 parts (2 parts on either side of a line or 2 parts inside and outside a circle). We need to prove that the G contains no odd length cycle.

Consider an odd-length cycle in the new graph. We can construct a geometric version of it as shown. Now remove the last object. By induction, it must then be an even cycle. Now consider any even length cycle that intersects the new object p . It will intersect even regions present on the cycle.

Case 1: It intersects 2 regions of $2L$ length cycle. Each region will be divided into 2 smaller regions, resulting in a new cycle of length $2L + 2$. So the new cycle will always be even.

Case 2: It intersects 1 region of $2L$ length cycle. The new cycle will remain as is because one of the 2 new regions obtained will never participate in the cycle.

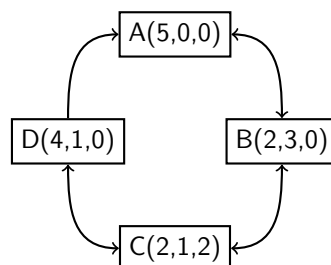
Case 3: It doesn't intersect any regions of a cycle. This can be safely ignored.

We also need to prove that newly formed cycles are always even in length. New cycles are obtained only when p divides a set of regions forming a path, making exactly 2 copies of such a path, each on either side of p . Let the length of any such path be L . All the resulting cycles obtained will always be of length $2 + 2k$, which is even (where $1 \leq k \leq L$).

Therefore, no new cycles of odd length are present in G . Hence G is bipartite.

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 $c = (c_1, c_2, c_3)$ 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.



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 $\text{width}[u]$ 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.

Tutorial 9

1. 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 $\text{BFSCONNECTED}(\text{Graph } G = (V, E), \text{Vertex } r, \text{int id})$.

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 $\text{BFSCONNECTED}(\text{Graph } G = (V, E), \text{Vertex } r, \text{int id})$, 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 $r.p_0 \dots p_k.p_{k+1}$. 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.

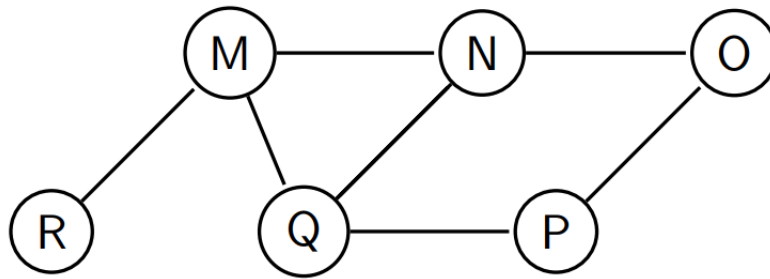
2. Suppose that there is an undirected graph $G(V,E)$ where the edges are colored either red or blue. Given two vertices u and v . It is desired to (i) find the shortest path irrespective of colour, (ii) find the shortest path, and of these paths, the one with the fewest red edges, (iii) a path with 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:

- (i) normal BFS will work.
- (ii) 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.

3. 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.

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

Algorithm 7: Modified DFS

```

1 time  $\leftarrow$  0;
2 Function DFS( $u$ ,  $parent$ ):
3   visited[ $u$ ] = true;
4   arrival[ $u$ ] = evv[ $u$ ] = time++;
5   for  $v$  in adj[ $u$ ] do
6     if not visited[ $v$ ] then
7       children[ $u$ ]++;
8       DFS( $v$ ,  $u$ );
9       evv[ $u$ ] = min(evv[ $u$ ], evv[ $v$ ]); // update earliest visitable vertex time
10      if  $parent$  is not null && evv[ $v$ ]  $\geq$  arrival[ $u$ ] then
11        AP[ $u$ ] = true; // no back edge from  $v$  to ancestors of  $u$ 
12      else
13        if  $v \neq parent$  then
14          evv[ $u$ ] = min(evv[ $u$ ], arrival[ $v$ ]); //  $v$  could have a smaller arrival time
15  if  $parent$  is null && children[ $u$ ] > 1 then
16    AP[ $u$ ] = true; // root with more than one child

```

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 $\text{DFS}(v)$ 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 ($\text{arrival}[a] > \text{arrival}[b]$ 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 $\text{DFS}(v)$ is more than $\text{arrival}[u]$ (for any such v), then u is an articulation point. The time complexity will be $\mathcal{O}(V + E)$.

Algorithm 8: Two-Vertex Connected

```

1 Function twoVC():
2   parent  $\leftarrow$  null;
3   for  $u$  in  $V$  do
4     if not visited[ $u$ ] then
5        $\text{DFS}(u, \text{parent})$ ;
6   for  $u$  in  $V$  do
7     if  $AP[u]$  then
8       return false;

```

5. Let $G(V, E)$ be a graph. We define a relation on edges as follows: two edges e and f are related (denoted by $e \sim f$ 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_a and e_b 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_a and r_b 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 P_1 into P_2 you can get the desired path which contain p as an edge. Augmentation had to be carefully handled! (why? it can lead to walks.. Make cases and show)
- For 2-edge connected component, the given equivalence relation turns into edges which are not separated by a Articulation Point

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

Solution: Modification: Run DFS on G and partition them into different sets depending upon whether they are visited in the same call of $DFS(v)$ or not. Let's call this partition as P_1 . And then run DFS on G^R and partition them into different sets depending upon whether they are visited in the same call or not. Let's call this partition 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](#).

7. Give similar modification for the algorithm SC.

Solution: For every node n find a node n' using the SC algorithm such that it's arrival time is least but it's hasn't departed yet (departure time greater than that of n). All the nodes which have same n' are present in a SCC.

8. 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 $v.\text{level} = u.\text{level} + 1$.

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.

9. 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)

```
1  $v.visited := True$ ;  
2  $v.arrival := time++$ ;  
3 for  $w \in G.adjacent(v)$  do  
4   if  $w.visited == False$  then  
5     DFSREC( $G, w$ )  
6   else  
7     if  $condition$  then  
8       throw "Found Cycle"  
9  $v.departure := time++$ ;
```

Solution:

$$w.arrival < v.arrival \ \& \ w.departure == DEFAULT \ \& \ w! = v.parent$$