FIT2004 Exam ~ Semester 2, 2020

Marking Guide

Question 1

State a **useful invariant** for *odd_prod*. Use that invariant to **prove that *odd_prod* calculates the product of all odd numbers in *L*.**

Note that the **empty product** (i.e. multiplying no numbers together) is 1.

```
def odd_prod(L[1...n]):
  prod = 1
  i = 1
  while i < len(L)
    if L[i] % 2 == 1:
      prod *= L[i]
    i += 1
  return prod
```

Answer:

Invariant: prod is the product of all odd numbers in L[1...i-1] **(1 mark).** Give 0.5 if the index is wrong. If they use 0 indexing instead of 1 indexing for the whole question, they can get 0.5 for the invariant but full marks for the rest of the proof as long as their proof makes sense.

Proof:

Initialisation: When i=1 (L[1...0] = [], empty product = 1 as required)

Assume the invariant holds at the state of some iteration, where i = k, i.e. that prod = product of all odd elements in L[1..k-1]. If element k is odd, then we multiply prod by that element, so now prod is the product of all odd numbers in L[1...k]. If kth element is not odd, product stays the same, but since L[k] is not odd, prod is still the product of all odd elements in L[1..k]. Since we increment i at the end of the iteration, i = k+1, and so at the start of the next iteration, we again have that prod is the product of all odd elements in L[1...i-1]

At the end of the algorithm, i = n+1. So the invariant tells us that prod is the product of all odd elements in L[1...n] = L, as required.

Question 2

Consider the algorithm for division by subtraction given below (assume that both inputs are always positive integers).

```
Def DivBySub(number, divisor)
  q = 0
  r = number
  while r >= divisor
```

```
    q += 1
    r -= divisor
  return q, r
```

What is the **big-O time complexity** of this algorithm? Note that we do not want the best or worst case complexity, but rather an expression for the **exact complexity**.

The correct answer is: O(q)

Question 3

Consider the following pseudocode for **three-way merge sort.** Which of the recurrences is an expression for the **time complexity** of this algorithm?

Note that ***merge()*** is a function which takes as input three sorted lists, and returns a single sorted containing all the elements from each of the inputs, and **runs in linear time**.

```
Def merge_sort(L[1..n]):
  if len(L) <= 1:
    return L
  n = len(L)
  A, B, C = L[1..(n//3)], L[(n//3)+1..(2n//3)], L[(2n//3)+1..n]
  A, C, C = merge_sort(A), merge_sort(B), merge_sort(C)
  return merge(A, B, C)
```

The correct answer is:

$T(N) = 3T(N/3) + bN$

$T(1) = c, T(0) = c$

Question 4

Given a **list** of *N* strings, where:

- The longest string is *M* characters long.
- Total number of unique character is *C (i.e. the alphabet is of size C)*

In each of the following two scenarios, determine which of **stable insertion sort** and **stable radix sort** would be more **time efficient**. Briefly (1 sentence or so) justify your answer to each.

1. *N* is **significantly larger** than both *M* and *C*.
2. *C* is **significantly larger** than both *M* and *N*.

Answer:
Justification should:

1. say radix sort because comparison N^2 will be large (or something similar)
2. say insertion sort because a) N^2 is small and b) the comparison cost would be less than that of initializing the count-array/ buckets.

Question 5

Given a list of *N* integers in a range of between *1* to *K (assume integers take O(1) space)*.

Discuss the **optimal auxiliary space complexity** to sort this list in ascending order **using Count Sort** if:

1. The sorted list **does not need to be stable** and the **original list does not need to be maintained**.
2. The sorted list **needs to be stable** and the **original list does not need to be maintained**.

Note: Only providing the auxiliary space complexity without an explanation would result in no marks given.

Answer:

- O(K) for unstable as we just need to count the frequency and the original list value can be replaced.
- O(K+N) for stable as we need to store the relative ordering of the items in the list.

Note: We are using integer here, we can assume each integer takes O(1) space to store in the list.

Question 6

Consider a **Quick Sort** variant which **uses 2 pivots to perform 4-way partitioning:**

- *Partition 1* for items that are of **lesser** value than *pivot 1*.
- *Partition 2* for items that are of **equal** value with *pivot 1*.
- *Partition 3* for items with values **between** *pivot 1* and *pivot 2*.
- *Partition 4* for items that are of **greater or equal** value than *pivot 2*

Quick sort is then called **recursively on Partition 1, 3 and 4.**

State the **best and worst-case time complexity** for this Quick Sort variant, if the **partitioning can be performed in O(N) time**. Briefly explain when they **occur**.

Answer:

- Best case is O(N) when all items are equal to pivot 1, leaving partition 1, 3 and 4 empty without needing to recurse again.
- Worst case is O(N^2) when all items end up in partition 1, 3 or 4 with the other partitions having only 1 item.

Question 7

Consider a list of exam results, represented as a list of *N* **unsorted** positive integers with values from *0* to *100*.

You realized that more than half of the class failed the exam. Thus, you are looking to moderate the results. You would want to moderate the results according to the following:

- To pass the exam, a student requires at least 50 marks.
- Exactly 60% of the class needs to pass the exam (don't worry about rounding the number of students up or down).
- In order to increase the passing percentage to 60%, we will give some failed students bonus marks equivalent to difference between their current marks and 50.
- Priority is given to the students that are the closest to the passing point. For example a student scoring 48 would be prioritised over a student scoring 45 for the bonus marks.

Describe an **efficient algorithm using quick select** to **determine the total bonus marks to be given out**; in **O(N) time complexity** with the **assumption that you can partition a list in O(1) time**.

Answer:

Algorithm must mention:

- Quick select to get the top 60% of the original list
- Loop through this partition and sum up the difference from the passing mark of 50

Question 8

Consider the following **Dynamic Programming problem**:

Given a rod of some length *N*, what is the **optimal way to cut this rod up to obtain maximum profit**?

You are given a list of prices *P[1..n]* for each length from *1* to *n*, which is the amount of profit you will get for selling a rod section of that length.

Write down a definition of the **overlapping subproblems (NOT the optimal substructure)** for this problem.

Answer:

Memo[i] = {The maximum profit that can be obtained by optimally cutting a rod of length i}

Question 9

The objective of a good hash function is to distribute keys more or less randomly in the table.

Why then is it a **bad idea** to hash items to **random positions** in the table?

Answer:

Since hash tables need to be able to look values up, hash functions need to be deterministic so that when an item is hashed to some position, we know where to find it later.

Note: Just saying a random hash is bad/unusable is not enough, since that information is given in the question.

Question 10

What is the **best** and **worst case time complexity** for looking up an item into a **separate chaining hash table**, where the chains are implemented using **AVL trees**?

- *M* is the size of the table (i.e. the number of AVL trees)
- *N* is the number of items currently in the table

Answers:
What is the best case complexity? → O(1)
What is the worst case complexity? → O(log(N))

Question 11

Consider the **cuckoo hashing scheme** which uses two arrays to store the values in the hashtable. State the **worst case time complexities** in big-O notation for the following operations, and briefly justify each of your answers.

- **Deleting** an item from a **cuckoo hash table.**
- **Inserting** an item into a **cuckoo hash table.**

Answer:

Deletion: O(1) This is because there are only two locations the item can be, and when we find it we can delete it without any additional work.

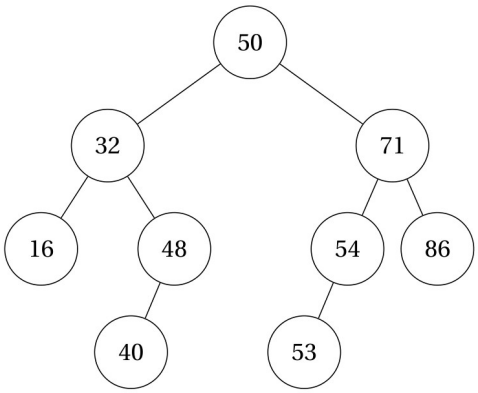
Insertion: There are a few possible answers here:

- Best answer: O(maxloop) where maxloop is some predefined cutoff. The insertion terminates after a fixed number of iterations in order to prevent an infinite loop.

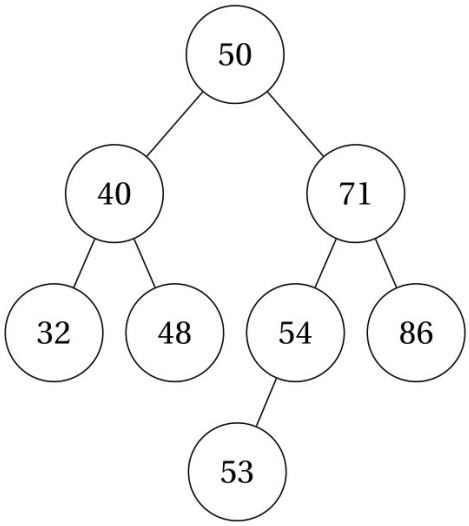
- Alternative answer: infinite/unbounded. It is possible for cuckoo hashing to keep kicking items out in a cycle which never ends.

Question 12

What is the state of this **AVL tree** when 16 is **deleted (after rebalancing)**?



Answer:



Question 13

Consider a **suffix trie** generated from a string of size *N*.

Instead of an array, each node of the trie stores the children of that node with a **sorted linked list**.

Briefly discuss:

1. A use of the trie/an operation performed on the trie in which such implementation is **less time efficient** than the array implementation**.**
2. A use of the trie/an operation performed on the trie in which such implementation is **more time efficient** than the array implementation**.**

Answer:

A possible discussion would mention:

- Insert/ Search where it is no longer possible for O(1) access and there is a cost to maintain the sorted list for insertion.
- Traversal where there is no longer a need to go through the entire array at every node.

However, there may be other correct answers.

Question 14

Consider a string S of length N.

You then perform the following step:

1. **Generate all of the suffixes** for the string, and store them in a **list**.
2. **Insert all of the suffixes** generated in step 1 into a **suffix trie**.
3. **Compress** the suffix trie generated in step 2 by compressing non-branching paths into a single edge with **[start, end] indices**

What is the **space complexity** for the **data structure in each step?**

The correct answer is:

1. O(N^2)
2. O(N^2)
3. O(N)

Question 15

Briefly discuss how a **suffix array** can be **constructed** using a **suffix trie** for any given string S.

Answer:

Discussion must mention:

- Store the suffix ID in the leaf.
- Perform an in-order traversal.

Question 16

Assume that we are constructing the **suffix array** for a string *S* using the **prefix doubling approach**. We have **already sorted** the suffixes for string S according to their **first 2 characters**; with the corresponding **rank array** shown below:

IDs from 1 to 11, ranks from left to right are: 4, 6, 5, 7, 4, 6, 3, 5, 7, 2, 1

We are now **sorting on the first 4 characters.**

For the following pairs of suffixes, describe how will you **compare them on their first 4 characters in O(1),** and **what the resulting order would be.**

1. Suffixes with ID 1 and ID 5
2. Suffixes with ID 3 and ID 5

Answer:

Description must mention:

- compare rank[1] vs rank[5]
- state same rank
- compare rank[1+2] vs rank[5+2]
- state ID5 before ID1
- compare rank[3] vs rank[5]
- state ID5 before ID3

Question 17

Consider a string *S* consisting of the following **5 characters** (the characters are presented in no particular order): *"$", "a", "a", "b", "c".*
S contains the following **2-mers:**

- *$a*
- *cb*
- *ac*

What is the **4th character (using 1-indexing)** of the **Burrows Wheeler Transform** of *S*?

The correct answer is: c

Question 18

The **Burrows Wheeler Transform** has **TWO important properties** that make it useful for **compression**. State these properties, and briefly explain why each of them are important for BWT to be useful for compression.

Answer:

Property 1: It can be inverted. This is required because without being able to invert, we could not recover the original massage**.**

Property 2: It tends to group runs of letters together. This is required because to we compress by replacing runs of letters with a single copy of that letter along with a number indicating how many letters were in that run.

Question 19

Consider the following pseudocode for **Breadth First Search**.

```
def BFS(G, s)
  visited[1...n] = False
  visited[s] = True
  q = Queue()
  q.push(s)
  while not q.isEmpty()
    current = q.pop()
    for each vertex v adjacent to u
      if not visited[v]
        visited[v] = True
        q.push[v]
```

What is the time complexity of this code in big-O notation if the graph is implemented as an **adjacency matrix**?

What is the time complexity of this code in big-O notation if the graph is implemented as an **adjacency list**?

The correct answer is:

- What is the time complexity of this code in big-O notation if the graph is implemented as an **adjacency matrix**? → O(V^2)
- What is the time complexity of this code in big-O notation if the graph is implemented as an **adjacency list**? → O(V+E)

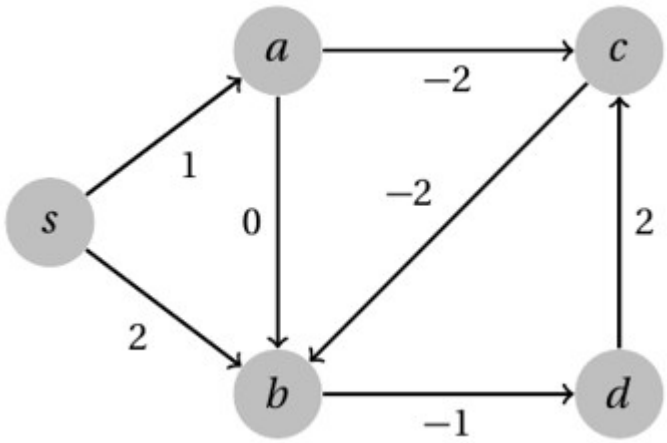
Question 20

Consider the following graph.

After some number of iterations of **Bellman-Ford**, the distance values to each vertex are as shown in the table below.

What would the distance values be **after one more iteration** of Bellman Ford? Please **choose** the appropriate value for each vertex from the **drop-down options**.

Assume that the **edges are being processed in the following order:** a->b, a->c, b->d, c->b, d->c, s->a, s->b



Vertex	Iteration		
	0	1	2
s	0	0	
a	∞	1	
b	∞	2	
c	∞	∞	
d	∞	∞	

What is the distance to **s** after iteration 2?

What is the distance to **a** after iteration 2?

What is the distance to **b** after iteration 2?

What is the distance to **c** after iteration 2?

What is the distance to **d** after iteration 2?

The correct answer is:

- What is the distance to **s** after iteration 2? → 0,
- What is the distance to **a** after iteration 2? → 1,
- What is the distance to **b** after iteration 2? → -3,
- What is the distance to **c** after iteration 2? → -1,
- What is the distance to **d** after iteration 2? → 0

Question 21

Consider a **weighted, directed graph** *G* with *V* vertices and *E* edges. The edge weights may be **negative**.

Describe how the standard **Floyd-Warshall** algorithm can **determine** if graph G **contains a negative cycle**.

Discuss a modification which could be made to the standard Floyd-Warshall algorithm to achieve the fastest possible **best case time complexity** for this.

Answer:

- The diagonal for the matrix is initialized to 0. If there is a negative value on the diagonal at any point we have detected a negative cycle

- Best case when the first intermediate vertex form a negative cycle in O(1) for a vertex back to itself. We can terminate immediately in this case.

Question 22

Consider the following pseudocode for **Dijkstra's** algorithm. The **priority queue** contains the values in *V* (i.e. the vertices in graph *G*) and is **ordered based on the values in *dist*.**

```
1: def dijkstra(G(V,E), s)
2:   dist[1..n] = inf
3:   pred[1..n] = None
4:   dist[s] = 0
5:   q = priority_queue(V[1..n], dist[1..n])
6:   while not q.is_empty()
7:     u = q.pop_min()
8:     for each edge (u,v,w)
9:       if dist[u] + w < dist[v]
10:         dist[v] = dist[u] + w
11:         update priority queue based on new distance for v
12:   return dist, pred
```

In this pseudocode, the *pred* array is **not being updated.**

1. Write down the line of pseudocode which needs to be **added to the algorithm above** in order to **update *pred* correctly,** and state between which two lines it should be inserted.
2. Write an algorithm, *path(u, v, pred)* which **determines the shortest path** from vertex *u* to vertex *v*. The algorithm *path* takes as input two vertices *u* and *v*, and the array *pred* which D**ijkstra's** algorithm produces as output.

The output of path should be a list where the **first element is *u***, the **final element is *v***, and the **elements in between are the vertices on the shortest path from *u* to *v*, in order.**

Answer:

The line that is needed is "pred[v] = u". It can go anywhere inside the innermost if block (after 9, after 10, after 11).

The algorithm should look something like this:

```
def path(u, v, pred)
  res = [v]
  current = v
  while current != u
    res.append(pred[current])
    current = pred[current]
  return reverse(res)
```

Question 23

Consider a **weighted**, **connected graph** *G* with *V* vertices and *E* edges. **All of the edges** *E* for the graph are **weighted with negative values.**

The **Kruskal minimum spanning tree algorithm** can be used to obtain the minimum spanning tree for graph *G* by sorting the edges in increasing order.

Briefly discuss and justify how the graph G can be preprocessed to work with **Kruskal algorithm** as described above to obtain the **maximum spanning tree** for *G***.**

Answer:

- Convert edges from negative to positive by multiplying -1.
- Note that the edges output by Kruskals are those which form an MST, and since the order of the edge weights is reversed by multiplying by -1, this is a maximum spanning tree in G. (their answer may not be as clear, but as long as they are expressing that idea)

Note: Question said without modifying the sort so the answer can't sort it in decreasing order.

Question 24

In a graph G, it is possible that all topological orderings of the vertices of G share some common prefix (i.e. there is no choice for these vertices, they must appear in the same order in all topological orderings of the graph). We will refer to these vertices as a **common prefix** of all topological orderings of G.
Consider a **directed acyclic graph (DAG)** G of *V* vertices and *E* edges; **represented using an adjacency matrix.** You want to **obtain a topological sort** of the vertices using the **Kahn's algorithm.**

Describe briefly how you would implement Kahn's algorithm in order to:

1. Determine when a vertex can be **added to the queue in O(1) time complexity.**
2. Determine if a vertex belongs to the **common prefix** of G**.**

Answer:

- Add an attribute to count the number of incoming edges.
- Whenever a vertex is served from the queue, check if the queue is empty and store it as an attribute. If the queue is empty, it means there were no other possible choices. Once the queue is ever non-empty, all subsequent vertices are not part of the common prefix.

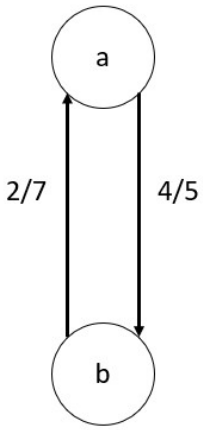
Question 25

Consider the following **pair of vertices in a flow network.**

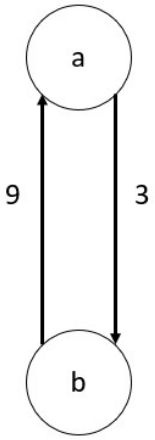
There is an **edge with capacity 5** from vertex *a* to vertex *b*, with a **flow of 4.**

There is an **edge with capacity 7** from vertex *b* to vertex *a*, with a **flow of 2**.

Which of the following scenarios **corresponds** to this pair of vertices **in the residual graph?**

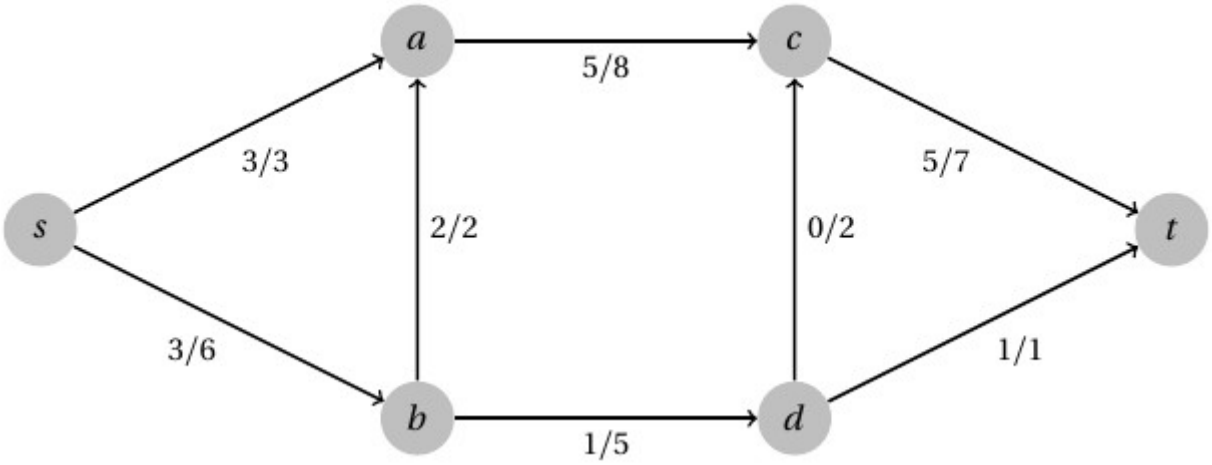


Answer:



Question 26

What is the **maximum possible flow** in the flow network **below**?



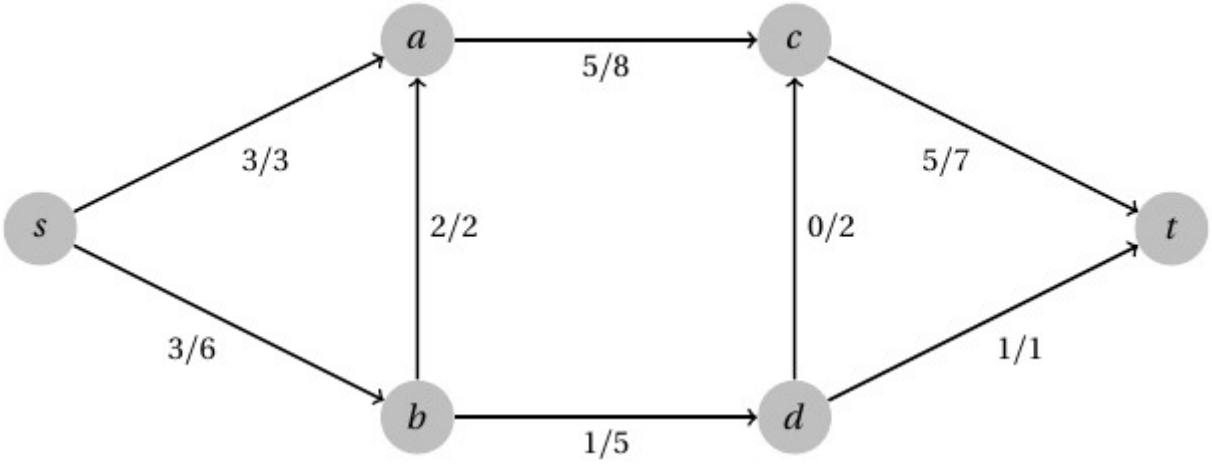
The correct answer is: 8

Question 27

A minimum cut is a partition of the vertices of a network into **two sets, *S* and *T*,** such that:

- The source is in *S*, the sink is in *T*.
- The **capacity** of the edges which are directed from *S* to *T* is **minimum**.

Which edges are in the **minimum cut** in the network shown below?



The correct answers are: s->a, b->a, d->c, d->t

Question 28

You are scheduling FIT2004 exams for the students.

- There are **5 possible timeslots** available for the exam. Each timeslot has a randomized set of questions from **a database of 1000 exam questions.**
- Each **student** could only **attend a single timeslot**; but the student can **select 3 of the timeslots as their preferred timeslots.**
- Each **timeslot** could only **fit up to 50 students.**

You come to the realization that you are not able to fit all of the students to their preferred timeslots. The exam questions will still be randomized irrespective of the timeslots.

However, **you would like to satisfy as many of the students' preferred timeslots as possible.**

1. Describe how to model this problem **as a maximum flow problem.**
2. State how you would solve the problem **algorithmically**, once it was modeled as a maximum flow problem.

Answer:

Description must mention:

- source and sink.
- student vertices.
- timeslot vertices.
- edge weighted 1 from source to student.
- edge weighted 1 from student to 3 timeslot.
- edge weighted 50 from timeslot to sink.

We could then solve the problem using Ford-Fulkerson and to get the maximum flow.
Note that the 1000 randomized exam questions are irrelevant.