CPSC 320 2017W2: Quiz 3 Sample Solution

1 Ol' McDonald is safer, he hi he hi ho

1.1 Ol' McDonald is safer, he hi he hi ho (Group)

The manager in a McDonald's comes to you with the following problem: she is in charge of a group of n workers. Each worker works one shift each day of the week (i.e., a particular worker works from the same start time to the same finish time each day, though two different workers may work different times). There can be multiple shifts happening at once. Assume that no shift starts before midnight or ends after midnight and that shifts overlap even if they just "meet" at their start or end times.

The manager is trying to choose a subset of these n workers to form a *safety committee* that she can meet with once a week. She considers such a committee to be *complete* if, for every worker X not on the committee, X's shift overlaps at least partially the shift of some worker who **is** on the committee. In this way, each worker's obedience to safety protocols can be observed by at least one person who is serving on the committee.

Example: Suppose that $n = 3$ and the shifts are

- Worker A: 0:00 to 10:00
- Worker B: 7:00 to 19:00
- Worker C: 12:00 to 23:59

then the smallest complete safety committee would consist of just worker B since the second shift overlaps both the first and the third.

1. Consider the following algorithm to produce a complete safety committee containing as few workers as possible.

   Each worker's shift is an interval. Suppose that I_x is the interval with the earliest finishing time (we will define time 0 as being midnight). We find all intervals that contain I_x's finishing time, and choose the interval I_y with the latest finishing time among those as part of the safety committee. We then delete all intervals that overlap I_y (including I_y), and repeat the operation until no intervals are left.

   Give an example that shows that this algorithm is **not** optimal.

2. Describe briefly *in words* an efficient algorithm that takes the schedule of n shifts and produces a complete safety committee containing as few workers as possible.

1.1.1 Quiz Solution

1. The following set of shifts (drawn rather than with explicit times) illustrates that the algorithm is not optimal:

```
 ----B----  -D- -E-
--A--  ------C------
```

(This greedy algorithm yields `B`, `D`, and `E`. The optimal solution is `B` and `C`. Notice how the greedy algorithm eliminates `C` from consideration for the safety committee because it is covered, even though including it would cover *other* shifts.)

2. Suppose that I_x is the interval with the earliest finishing time (we will define time 0 as being midnight). We find all intervals **in the original, complete instance** (and not just of those still uncovered) that contain I_x's finishing time, and choose the interval I_y with the latest finishing time among those as part of the safety committee. We then delete all intervals that overlap I_y (including I_y), and repeat the operation until no intervals are left.

   (Notice that this is the same as the original algorithm except that we may use a shift that has already been covered by some other shift in the safety committee.)

1.2 Ol' McDonald is safer, he hi he hi ho (Individual)

As in the group stage: The manager in a McDonald's comes to you with the following problem: she is in charge of a group of n workers. Each worker works one shift each day of the week (i.e., a particular worker works from the same start time to the same finish time each day, though two different workers may work different times). There can be multiple shifts happening at once. Assume that no shift starts before midnight and ends after midnight and that shifts overlap even if they just "meet" at their start or end times.

The manager is trying to choose a subset of these n workers to form a *safety committee* that she can meet with once a week. She considers such a committee to be *complete* if, for every worker X not on the committee, X's shift overlaps at least partially the shift of some worker who **is** on the committee. In this way, each worker's obedience to safety protocols can be observed by at least one person who is serving on the committee.

Example: Suppose that $n = 3$ and the shifts are

- Worker A: 0:00 to 10:00
- Worker B: 7:00 to 19:00
- Worker C: 12:00 to 23:59

then the smallest complete safety committee would consist of just worker B since the second shift overlaps both the first and the third.

Moving on to new questions:

1. Describe two trivial instances for this problem and their solutions.

   (a) Instance: , Solution:

   (b) Instance: , Solution:

2. One correct and efficient algorithm to solve this problem sorts the endpoints of the shifts in increasing order and then iterates through them making greedy choices (left unspecified here) of whether to include people in the safety committee. We will call a shift s *covered* if the worker for shift s is in the safety committee, or isn't but s overlaps a shift whose worker **is** in the safety committee. A shift that is not covered will be called *uncovered*.

Which data structure(s) will this algorithm need to maintain as it is iterating through the endpoints of the shifts? Assume that p is the shift endpoint currently being processed. Fill in the box next to **all** statements that apply.

- ☐ A list of all covered shifts.
- ☐ A list of all uncovered shifts.
- ☐ The shifts $S_1, \ldots, S_j$ whose workers were added to the safety committee so far.
- ☐ A list of the covered shifts whose starting time is after S_j's finishing time, but no later than p.
- ☐ A list of the uncovered shifts whose starting time is after S_j's finishing time, but no later than p.

3. Which of the following is the best **lower bound** on the running time of the algorithm mentioned in the previous question?
   - ○ $\Omega(\log n)$
   - ○ $\Omega(n)$
   - ○ $\Omega(n \log n)$
   - ○ $\Omega(n^2)$
   - ○ $\Omega(n^2 \log n)$
   - ○ None of these is a lower bound.

4. Let $S_1, \ldots, S_k$ be the set of shifts returned by the greedy algorithm. Assume without loss of generality that $S_1, \ldots, S_k$ is sorted by finishing times. Also, let $T_1, \ldots, T_m$ be the shifts in a smallest safety committee, sorted by increasing finishing time. We will denote the starting and finishing times of shift x by $s(x)$ and $f(x)$ respectively.

   Which of the following statements would it be useful to prove as part of a proof that the greedy algorithm is optimal? Fill in the box next to **all** statements that apply.
   - ☐ k < m
   - ☐ k $\leq m$
   - ☐ k $\geq m$
   - ☐ For $j = 1, \ldots, k$, $s(S_j) \geq s(T_j)$.
   - ☐ For $j = 1, \ldots, k$, $f(S_j) \geq f(T_j)$.
   - ☐ For $j = 1, \ldots, m$, $s(S_j) \geq s(T_j)$.
   - ☐ For $j = 1, \ldots, m$, $f(S_j) \geq f(T_j)$.

1.2.1 Quiz Solution

1. Here are two trivial instances with their solutions:

   (a) The empty instance (no shifts). Solution: no shifts.

   (b) Instance: worker A works 1:00–2:00. Solution: worker A is on the safety committee.

   There are others, but instances essentially equivalent to these seem the most natural trivial instances.

2. The efficient algorithm will want to maintain the following (and potentially some other information): A list of all uncovered shifts, the shifts $S_1, \ldots, S_j$ whose workers were added to the safety committee so far, and a list of the uncovered shifts whose starting time is after S_j's finishing time, but no later than p.

3. The best **lower bound** on the running time of the (still-unspecified!) greedy algorithm is $\Omega(n \lg n)$. We have to sort as a pre-processing step!

4. These statements would be useful to prove in establishing that the greedy algorithm is optimal: $k \leq m$, and for $j = 1, \ldots, m$, $f(S_j) \geq f(T_j)$.

2 Recurrences resolve runtimes

2.1 Recurrences resolve runtimes (Group)

A *recurrence relation* is a function definition that expresses the value of the function for an argument n in terms of its values for arguments smaller than n. Here is a well known example of the recurrence relation for Fibonacci numbers:

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

Knowing that $F(0) = 0$ and $F(1) = 1$, we can then compute

- $F(2) = F(1) + F(0) = 1 + 0 = 1$
- $F(3) = F(2) + F(1) = 1 + 1 = 2$
- $F(4) = F(3) + F(2) = 2 + 1 = 3$
- $F(5) = F(4) + F(3) = 3 + 2 = 5$

etc.

When we are given a recursive algorithm, one (and, frequently, the only) way to determine its worst case running time is by writing a recurrence relation for it. In this week's reading quiz, you learn how to solve the recurrence relations that arise from a specific class of algorithms called *divide and conquer* algorithms. This tutorial quiz gives you practice obtaining these recurrence relations. Take `mergesort` as an example, with annotations about its runtime on the right:

```
define mergesort(A, left, right):
  if left < right:                          // O(1) time, divides into two cases
      mid = floor ((left + right)/2)        // O(1) time
      mergesort(A, left, mid)               // first recursive call
      mergesort(A, mid+1, right)            // second recursive call
      merge(A, left, mid, right)            // linear time call to helper
```

Suppose that the sublist A[left ... right] of A that we are sorting contains n elements (that is, right $-$ left $+1 = n$). The `if` guides us to build two cases for the recurrence. In the case when $n = 1$, nothing happens and so the running time is constant. Otherwise, the function performs two recursive calls and a call to `merge`. The time taken by each recursive call is described by the recurrence relation itself. The first recursive call is on $\lceil n/2 \rceil$ elements, and the second on $\lfloor n/2 \rfloor$ elements for $T(\lceil n/2 \rceil)$ and $T(\lfloor n/2 \rfloor)$ time. Finally we know `merge` runs in $\Theta(n)$ time. We thus get the recurrence

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n \geq 2 \\ \Theta(1) & \text{if } n \leq 1 \end{cases}$$

Write recurrence relation describing the worst case running time of the following functions:

1.
```
define hanoi(n, from, to, using):
    if n > 0:
       hanoi(n-1, from, using, to)
       print ("Move disc from peg " + from + " to peg " + to)
       hanoi(n-1, using, to, from)
```

2. Assume that the call `ElmerJ(A, first, n, x)` runs in $\Theta(n \log n)$ time.

```
define BugsB(A, first, n):
    if n < 3:
       return A[first] - 1

    x = BinarySearch(A, first, n, BugsB(A, first + floor(n/3), floor(n/3)))
    ElmerJ(A, first, n, x)
    return BugsB(A, first, floor(n/6)) * BugsB(A, first + floor(n/2), floor(n/4))
```

3. In the following example, assume that `A` is a matrix (a 2-dimensional array), that `Get(A, i, j, k, l)` returns the square submatrix of A containing rows `i` through `j` and columns `k` through `l`, that `Put(C, i, j, k, l, X )` stores `X` into the part of C containing containing rows `i` through `j` and columns `k` through `l`. Both operations run in time proportional to the number of elements in the submatrix. You may also assume that `Add(X,Y)` returns the sum of matrices `X` and `Y`, and that `Sub(X,Y)` returns the difference of matrices `X` and `Y`. `Add` and `Sub` run in time proportional to the number of elements in the matrices `X` and `Y`.

```
// Assumption: n is a power of 2.
define strassen(A, B, n):
    C = new matrix(n, n)
    if n = 1:
        C[0, 0] = A[0, 0] * B[0, 0]
    else:
        A11 = Get(A, 0,   n/2-1, 0, n/2-1);  A12 = Get(A, 0,   n/2-1, n/2, n-1)
        A21 = Get(A, n/2, n-1,   0, n/2-1);  A22 = Get(A, n/2, n-1,   n/2, n-1)

        B11 = Get(B, 0,   n/2-1, 0, n/2-1);  B12 = Get(B, 0,   n/2-1, n/2, n-1)
        B21 = Get(B, n/2, n-1,   0, n/2-1);  B22 = Get(B, n/2, n-1,   n/2, n-1)

        P1 = strassen(Add(A11, A22), Add(B11, B22), n/2)
        P2 = strassen(Add(A21, A22), B11, n/2)
        P3 = strassen(A11, Sub(B12, B22), n/2)
        P4 = strassen(A22, Sub(B21, B11), n/2)
        P5 = strassen(Add(A11, A12), B22, n/2)
        P6 = strassen(Sub(A21, A11), Add(B11, B12), n/2)
        P7 = strassen(Sub(A12, A22), Add(B21, B22), n/2)

        Put(C, 0,   n/2-1, 0,     n/2-1, Add(Add(P1, P7), Sub(P4, P5)))
        Put(C, 0,   n/2-1, n/2,   n-1,   Add(P3, P5))
        Put(C, n/2, n-1,   0,     n/2-1, Add(P2, P4))
        Put(C, n/2, n-1,   n/2,   n-1,   Add(Add(P1, P6), Sub(P3, P2)))
   endif
   return C
```

2.1.1 Quiz Solution

1. The Hanoi problem:

```
define hanoi(n, from, to, using):
  if n > 0:                             // divides into two cases; the n <=0 case is trivial
     hanoi(n-1, from, using, to)        // recursive call; runtime described by the recurrence!
     print ("Move disc ..." ... to)     // O(1)
     hanoi(n-1, using, to, from)        // recursive call; runtime described by the recurrence!
```

$$T(n) = \begin{cases} O(1) & \text{if } n = 0 \\ 2T(n-1) + O(1) & \text{if } n > 0 \end{cases}$$

2. The BugsB problem. The fact that we call `BugsB` to produce one of the arguments to `BinarySearch` may be confusing, but there is **nothing** exciting going on there. Similarly, nothing exciting happens when we multiply the result of two such calls together. To see why, let's rewrite this:

```
define BugsB(A, first, n):
    if n < 3:
       return A[first] - 1

    x = BinarySearch(A, first, n, BugsB(A, first + floor(n/3), floor(n/3)))
    ElmerJ(A, first, n, x)
    return BugsB(A, first, floor(n/6)) * BugsB(A, first + floor(n/2), floor(n/4))
```

Like this:

```
define BugsB(A, first, n):
    if n < 3:                                         // divides off a base case where n < 3
       return A[first] - 1

    temp = BugsB(A, first + floor(n/3), floor(n/3))   // recursive call on size floor(n/3)
    x = BinarySearch(A, first, n, temp)               // Theta(lg n) runtime
    ElmerJ(A, first, n, x)                            // Theta(n lg n) runtime
    temp1 = BugsB(A, first, floor(n/6))               // recursive call: T(floor(n/6))
    temp2 = BugsB(A, first + floor(n/2), floor(n/4))  // recursive call: T(floor(n/4))
    return temp1 * temp2                              // O(1)
```

$$T(n) = \begin{cases} O(1) & \text{if } n < 3 \\ T(\lfloor\frac{n}{3}\rfloor) + T(\lfloor\frac{n}{6}\rfloor) + T(\lfloor\frac{n}{4}\rfloor) + \Theta(n \lg n) & \text{if } n \geq 3 \end{cases}$$

3. The Strassen problem: We won't annotate the original code for this one but just note that there are a **large** number of operations done that all take time proportional to $\frac{n}{2} \cdot \frac{n}{2} \in \Theta(n^2)$ that all get rolled together and abstracted away in our function. Similarly, the seven different recursive calls are on many different specific matrices, but what matters is that they're all called on subproblems of the same **size**.

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 7T(\frac{n}{2}) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

2.2 Recurrences resolve runtimes (Individual 1)

Fill in the blanks in the recurrence relations describing the worst case running time of each of the following functions. You may ignore floors and ceiling when you write the terms of the recurrence relations.

1. Assume that the call `merge(A, left, mid, right)` runs in $\Theta(\texttt{right} - \texttt{left})$ time.

```
define skewedsort(A, left, right):
    if left + 1 < right:
        mid = floor ((2*left + right)/3)
        skewedsort(A, left, mid)
        skewedsort(A, mid+1, right)
        merge(A, left, mid, right)
    else if left < right:
        if A[left] > A[right]:
            swap(A[left], A[right])
```

$$T(n) = \begin{cases} T(\boxed{\quad\quad\quad}) + T(\boxed{\quad\quad\quad}) + \boxed{\quad\quad} & \text{if } n \geq \boxed{\quad} \\ \boxed{\quad\quad} & \text{if } n \leq \boxed{\quad} \end{cases}$$

2. This one is a bit tricky, so be careful.

```
define pointless(A, n)
    // A is an array with n elements
    if n > 1:
        return pointless(A, n-1) * pointless(A, n-2)
    else:
        return A[n-1]
```

$$T(n) = \begin{cases} \boxed{\quad\quad\quad\quad\quad\quad} + \Theta(1) & \text{if } n \boxed{\quad\quad} \\ \boxed{\quad\quad\quad} & \text{if } n \boxed{\quad\quad} \end{cases}$$

3. Assume that the call `ALittleBitOfThis(A, size)` runs in $\Theta(1)$ time, and the call `ALittleBitOfThat(A, size, i)` (**correction:** `size` should have been `n`) runs in $\Theta(\log n)$ time,

```
define recurrence(A, size, n)
    // A is an array with size elements, and 0 <= n < size.
    if n = 0:
        ALittleBitOfThis(A, size)
    else:
        i = n
        while i > 0:
            i = floor(i/2)
            ALittleBitOfThat(A, n, i)
            recurrence(A, size, n-1)
            ALittleBitOfThat(A, n, i)
```

$$T(n) = \begin{cases} \boxed{} + \Theta(\boxed{}) & \text{if } n \boxed{} \\ \boxed{} & \text{if } n \boxed{} \end{cases}$$

2.2.1 Quiz Solution

1. $$T(n) = \begin{cases} T(\lceil n/3 \rceil) + T(\lfloor 2n/3 \rfloor) + \Theta(n) & \text{if } n \geq 3 \\ \Theta(1) & \text{if } n \leq 2 \end{cases}$$

2. $$T(n) = \begin{cases} T(n-1) + T(n-2) + \Theta(1) & \text{if } n > 1 \\ \Theta(1) & \text{if } n \leq 1 \end{cases}$$

3. $$T(n) = \begin{cases} \log n T(n-1) + \Theta(\log^2 n) & \text{if } n > 0 \\ \Theta(1) & \text{if } n = 0 \end{cases}$$

2.3 Recurrences resolve runtimes (Individual 2)

Fill in the blanks in the recurrence relations describing the worst case running time of each of the following functions. You may ignore floors and ceiling when you write the terms of the recurrence relations.

1. Assume that a call to `LinearTimeAlgorithm(A, first, n1, second, n2)` runs in $O(n1+n2)$ time.

```
define bizarre(A, first, n)
    if (n > 1):
        bizarre(A, first, floor(n/2))
        bizarre(A, first + floor(n/2), floor(n/3))
        bizarre(A, first + n - floor(n/4), floor(n/4))
        LinearTimeAlgorithm(A, first, floor(n/2),
                                first + floor(n/2), n - floor(n/2) - floor(n/4))
```

$$T(n) = \begin{cases} T(\boxed{}) + T(\boxed{}) + T(\boxed{}) + \boxed{} & \text{if } n > \boxed{} \\ \boxed{} & \text{if } n \leq \boxed{} \end{cases}$$

2. This one is a bit tricky, so be careful.

```
define pointless(A, n, item)
    // A is an array with n elements; item is an array element
    if n > 0
        x = binarysearch(A, n, item)
        pos = pointless(A, n-1, pointless(A, n-2, x))
    else:
        return 1
```

$$T(n) = \begin{cases} \boxed{} + \Theta(\log n) & \text{if } n\boxed{} \\ \boxed{} & \text{if } n\boxed{} \end{cases}$$

3. Assume that the call `ALittleBitOfThis(A, size)` runs in $\Theta(1)$ time, and the call `ALittleBitOfThat(A, n, i)` runs in $\Theta(\sqrt{n})$ time,

```
define recurrence(A, size, n)
    // A is an array with size elements, and 0 <= n < size.
    if n <= 1:
        ALittleBitOfThis(A, size)
    else:
```

```
for i = n-1 downto 0:
    ALittleBitOfThat(A, size, i)
    recurrence(A, size, n-2)
    ALittleBitOfThat(A, size, i)
```

$$T(n) = \begin{cases} \boxed{} + \Theta(\boxed{}) & \text{if } n \boxed{} \\ \boxed{} & \text{if } n \boxed{} \end{cases}$$

2.3.1 Quiz Solution

1. It's tempting to think that the fact that we call `LinearTimeAlgorithm` on just $\frac{3}{4}n$ elements is important, but bear in mind that 3/4 is just yet another constant multiplied into that linear factor. We can't "tell the difference" between that and, say, one "primitive operation" taking 3/4 as long as another.

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lfloor n/3 \rfloor) + T(\lfloor n/4 \rfloor) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n \leq 1 \end{cases}$$

2.

$$T(n) = \begin{cases} T(n-1) + T(n-2) + \Theta(\log n) & \text{if } n > 0 \\ \Theta(1) & \text{if } n = 0 \end{cases}$$

3. It's awkward that `size` makes an appearance in the final function, but that's actually not too uncommon. What would likely be happening in a "real" algorithm is that we have an original input size and, at various points in the recursion as we break the problem down into smaller pieces, we still need to refer back to that original problem size.

$$T(n) = \begin{cases} nT(n-2) + \Theta(n\sqrt{\texttt{size}}) & \text{if } n > 1 \\ \Theta(1) & \text{if } n \leq 1 \end{cases}$$

3 Playing the Blame Game

A distributed computing system composed of n nodes is responsible for ensuring its own integrity against attempts to subvert the network. To accomplish this, nodes in the system can assess each others' integrity, which they always do in pairs. A node in such a pair with its integrity intact will correctly assess the node it is paired with to report either "intact" or "subverted". However, a node that has been subverted may freely report "intact" or "subverted" regardless of the other node's status.

The goal is for an outside authority to determine which nodes are intact and which are subverted. If $n/2$ or more nodes have been subverted, then the authority cannot necessarily determine which nodes are intact using any strategy based on this kind of pairing. However, if more than $n/2$ nodes are intact, it is possible to confidently determine which are which.

Throughout this problem, we assume that **more** than $n/2$ of the nodes are intact. Further, we let one "round" of pairings be any number of pairings overall as long as each node participates in at most one pairing. (I.e., a round is a matching that may not be perfect.)

1. Imagine that nodes a and b have been paired. No matter what report we receive, both nodes could have been subverted because subverted nodes may respond arbitrarily to a pairing.

   Fill in the circle next to **all other** possible situations corresponding to a given report from the nodes:

a **reports** b is	b **reports** a is	Could be:		
intact	intact	☐ both intact	☐ a intact, b subverted	☐ a subverted, b intact
intact	subverted	☐ both intact	☐ a intact, b subverted	☐ a subverted, b intact
subverted	intact	☐ both intact	☐ a intact, b subverted	☐ a subverted, b intact
subverted	subverted	☐ both intact	☐ a intact, b subverted	☐ a subverted, b intact

   (Reminder: both subverted is **always** a possibility.)

2. Imagine that we've found **one** node that is definitely intact and that, having done so, we now only make a pairing when at least one node in the pair is known to be intact. Give good asymptotic lower- and upper-bounds on the number of rounds required to determine for every other node whether it is intact or subverted.

   Asymptotic lower-bound on rounds: ☐

   Asymptotic upper-bound on rounds: ☐

   (We can now have as a goal finding one intact node.)

3. Now, imagine that we are searching for one definitely-intact node among a set of nodes of which **more** than half are intact. We find a way to discard from the search $k > 0$ nodes with the guarantee that at least half of the discarded nodes are subverted (i.e., at least as many discards are subverted as intact).

   Give **exact** lower- and upper-bounds on k in terms of n such that **more** than half of the new search space is guaranteed to be intact.

   Exact lower-bound on k: ☐

   Exact upper-bound on k: ☐

4. Again imagine that we are searching for one definitely-intact node among a set of nodes of which **more** than half are intact. This time, we find a way to discard from the search $k > 0$ nodes that may be any mixture of intact and subverted nodes such that there are k nodes **still in the search space** in exactly the same mixture of intact and subverted nodes. (There are no other nodes discarded, but there may be others kept in the search space.)

   Give **exact** lower- and upper-bounds on k in terms of n such that **more** than half of the new search space is guaranteed to be intact. (Note that we **do** care about $\lfloor$floors$\rfloor$ and $\lceil$ceilings$\rceil$ in your answer, though substantial partial credit is available without them.) **Consider only the case** where $n \equiv 1 \mod 4$, i.e., $n - 1$ is exactly divisible by 4.

Exact lower-bound on k: $\boxed{}$

Exact upper-bound on k: $\boxed{}$

3.1 Quiz Solution

1.

a **reports** b is	b **reports** a is	Could be:
intact	intact	both intact (or both subverted)
intact	subverted	b intact and a subverted (or both subverted)
subverted	intact	a intact and b subverted (or both subverted)
subverted	subverted	a intact and b subverted or b intact and a subverted (or both subverted)

To illustrate why, let's look at just the second row. If a is intact, then b is intact, too (since a reports b is intact), which means that a is subverted (since b reports a is subverted), and that's a contradiction. So, a cannot be intact. By similar reasoning, however, we can find that b *can* be intact. (Of course, b may also be subverted.)

2. Asymptotic lower-bound on rounds: $\Omega(\lg n)$. If we "get lucky" over and over again, doubling our number of intact nodes each time.

   Asymptotic upper-bound on rounds: $O(n)$. If we "get unlucky" and discover **all** the subverted nodes (and there are a linear number of them) before we finally find an intact node to help speed things up.

3. Exact lower-bound on k: 1. (Because we know $k > 0$.)

   Exact upper-bound on k: $n - 1$. (Because we must have at least one node left to have a majority intact.)

4. Give **exact** lower- and upper-bounds on k in terms of n such that **more** than half of the new search space is guaranteed to be intact. (Note that we **do** care about $\lfloor$floors$\rfloor$ and $\lceil$ceilings$\rceil$ in your answer, though substantial partial credit is available without them.) **Consider only the case** where $n \equiv 1 \mod 4$, i.e., $n - 1$ is exactly divisible by 4.

   Exact lower-bound on k: $\lfloor \frac{n}{2} \rfloor$

   Exact upper-bound on k: $\lfloor \frac{n}{2} \rfloor$

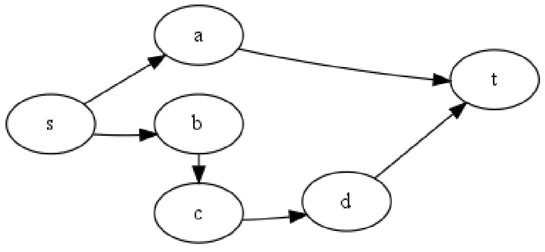
   Strange. How in the world could we pair off $\lfloor \frac{n}{2} \rfloor$ with another $\lfloor \frac{n}{2} \rfloor$ nodes so we know that everything on the "left" has the same status as its partner on the "right"? Even when we do, how do we **know** that we're left with a **majority** intact nodes? What's with the leftover node?

   Feels like someone should assign themselves to figure this out.

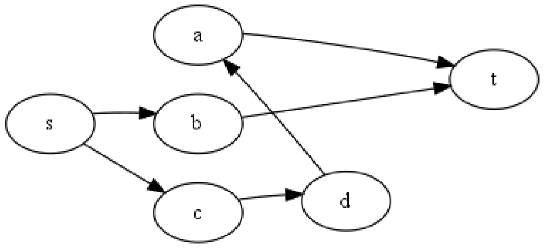
4 Mixed Nets

You're working on the routing for an anonymization service called a "mixnet" in which a network of computers pass messages through a sequence of handoffs from one source computer to another target computer.

To represent this, you have a weakly-connected, directed acyclic graph (DAG) $G = (V, E)$ composed of designated source and target vertices $s, t \in V$ and a set of $p > 0$ simple paths (along which p messages pass) each of which starts at s, ends at t, and includes at least one vertex in between s and t. The paths are also vertex disjoint besides s and t (i.e., no two paths share any other vertex). There are no other vertices or edges in the graph. For example, here are two different graphs both over the same set of vertices and both with $p = 2$:



Graph 1



Graph 2

1. Let $n = |V|$. Give exact lower- and upper-bounds on p in terms of n.

   Exact bounds: $\boxed{} \le p \le \boxed{}$

2. Let $m = |E|$. Give exact lower- and upper-bounds on p in terms of m.

   Exact bounds: $\boxed{} \le p \le \boxed{}$

3. In a valid instance of this problem, the p paths are vertex disjoint besides the shared start at s and end at t. Are they therefore also *edge-disjoint* (i.e., no two paths include the same edge)?

   ○ ALWAYS
   ○ SOMETIMES
   ○ NEVER

4. In a valid instance of this problem, is there **any** path from s to t?

   ○ ALWAYS
   ○ SOMETIMES
   ○ NEVER

   Your mixnet actually involves a single set of computers (with a designated start and target computer) and two entirely separate sets of paths among those computers, as with the two sample graphs on the previous page. At some point as each message passes along its path among the first set of paths, it switches to using one of the second set of paths instead (never switching back).

   Specifically, you have an overall graph made up of two subgraphs like those specified in the previous part, where one subgraph's vertices is an exact copy of the other's, i.e., $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, where a vertex $v_1 \in V_1$ if and only if there is a vertex $v_2 \in V_2$. (s_1 and t_1 are the start and target vertices in G_1 and their corresponding vertices s_2 and t_2 are the start and target in G_2.) Each subgraph is based on its own set of p paths, but p is the same for both. There is **also** a directed edge (v_1, v_2) for each vertex $v_1 \in V_1$ **except** s_1 and t_1 leading *from* G_1 *to* G_2. (There is no edge from s_1 to s_2 or t_1 to t_2.)

   So, for the example on the previous page, the overall graph would include both Graph 1 (with each node subscripted like a_1) and Graph 2 (with each node subscripted like a_2) plus 4 more edges: $(a_1, a_2), (b_1, b_2), (c_1, c_2), (d_1, d_2)$.

   Your goal is to find p vertex-disjoint paths in the overall graph that start at s_1 in G_1 and end at t_2 in G_2. Thus, no two paths visit the exact same vertex (besides s_1 and t_2). **Additionally**, you want to ensure that no two paths even visit the same vertex in *different* subgraphs (i.e., no path contains v_1 if any other path contains v_2).

5. In a valid instance of this problem, is there **any** path from s_1 to t_2?

   ◯ ALWAYS

   ◯ SOMETIMES

   ◯ NEVER

6. In a valid instance of this problem, is there **any** path from t_1 to t_2?

   ◯ ALWAYS

   ◯ SOMETIMES

   ◯ NEVER

7. Fill in the box next to **all** the edges used in a solution to the problem based on the examples on the previous page:

   ☐ (a_1, a_2)
   ☐ (b_1, b_2)
   ☐ (c_1, c_2)
   ☐ (d_1, d_2)

8. Fill in the box next to **all** the edges in a solution to the problem based on the examples on the previous page that has the right number of well-formed paths but violates one of the other constraints on solutions:

   ☐ (a_1, a_2)
   ☐ (b_1, b_2)
   ☐ (c_1, c_2)
   ☐ (d_1, d_2)

9. Given a valid instance of this problem, does it necessarily have a valid solution (i.e., a solution with p vertex-disjoint paths where no two paths even visit the same vertex in *different* subgraphs)?

   ◯ ALWAYS

   ◯ SOMETIMES

   ◯ NEVER

4.1 Quiz Solution

1. Exact bounds: $1 \leq p \leq n-2$. (For the lower bound: we stated $p > 0$, and a single path can "consume" any number of vertices. For the upper bound: each path requires at least one "intermediate" node plus there's the extra start and target nodes.)

2. Exact bounds: $1 \leq p \leq \frac{m}{2}$. (For the lower bound: we stated $p > 0$, and a single path can "consume" any number of edges. For the upper bound: Since a path must have at least three vertices (start, at least one intermediate, and terminal), every path "consumes" at least two edges.)

3. The paths are **ALWAYS** edge-disjoint. (They don't share any of the vertices in their "middles", and they're at least three vertices long. So, there's no neighbouring pair of vertices shared between any two paths, which means no edges shared.)

4. There is **ALWAYS** a path from s to t.

5. There is **ALWAYS** a path from s_1 to t_2.

6. There is **NEVER** a path from t_1 to t_2.

7. In the problem based on the example graphs on the previous page, the subgraph 1 to subgraph 2 edges used in the correct solution are: $(a_1, a_2), (b_1, b_2)$. (So the two paths are: s_1, a_1, a_2, t_2 and s_1, b_1, b_2, t_2.)

8. In the problem based on the example graphs on the previous page, the subgraph 1 to subgraph 2 edges used in a solution that has the right number of well-formed paths but violates on of the other constraints are either: $(a_1, a_2), (c_1, c_2)$ or $(a_1, a_2), (d_1, d_2)$. (For the first of these two options, the two paths are: s_1, a_1, a_2, t_2 and $s_1, b_1, c_1, c_2, d_2, a_2, t_2$, but these share the vertex a_2, which is not allowed.)

9. A valid instance **SOMETIMES** has a valid solution (i.e., a solution with p vertex-disjoint paths where no two paths even visit the same vertex in *different* subgraphs).

   For example, consider if we let G_1 be Graph 2 and G_2 be Graph 1 above. Then there is no longer any valid solution (because both paths **must** visit node c in one graph or the other, leading to a conflict).

5 Cover Charge

In the "minimum edge cover" problem, the input is a simple, undirected, connected graph $G = (V, E)$ with $|V| \geq 2$, and the output is the smallest possible set E' such that $E' \subseteq E$ and for all vertices $v \in V$, there is an edge $\{v, u\}$ (which is the same as $\{u, v\}$) in E'. That is, every vertex in the graph is the endpoint of some edge in E'.

Consider the following greedy algorithm for this problem:

```
Sort the vertices in increasing order by degree
Mark all vertices as uncovered
Let the cover E' be empty.
While there are vertices remaining, pick the next one v:
    If v is uncovered:
        If there is any neighbour u of v that is uncovered:
            Find the uncovered neighbour u of v with lowest degree
            Add {u, v} to E' and mark u and v as covered
        Else:
            Pick an arbitrary edge {u, v}, add it to E', and mark v as covered
```

1. Does this greedy algorithm produce an edge cover (whether or not it is minimal)?

   ◯ ALWAYS

   ◯ SOMETIMES

   ◯ NEVER

2. Does this greedy algorithm produce a minimal edge cover?

   ◯ ALWAYS

   ◯ SOMETIMES

   ◯ NEVER

3. Assuming that the graph is represented as an adjacency list and that we can determine the degree of a vertex in constant time, give a good big-O bound on the runtime of this greedy algorithm in terms of $n = |V|$ and $m = |E|$.

   Big-O Bound: $\boxed{\quad}$

A maximum matching in a graph $G = (V, E)$ is the largest set E'' such that $E'' \subseteq E$ and there are no three vertices $v_1, v_2, v_3 \in V$ such that $\{v_1, v_2\}$ and $\{v_1, v_3\}$ are in E''. That is: E'' "marries off" as many vertices as possible without having any one vertex "married" to two or more vertices.

In this part, assume you have a graph $G = (V, E)$ and a maximum matching for the graph E''.

4. Under what conditions is E'' itself a minimal edge cover? Fill in the circle next to the **best** answer.

   ◯ E'' is a perfect matching

   ◯ E'' is a stable matching

   ◯ $\{|E''| = \frac{|E|}{2}\}$

   ◯ none of these guarantees E'' is a minimal edge cover

5. Let E'' be a maximum matching in $G = (V, E)$, $v \in V$ be a vertex that is **not** covered by E'', and $\{u, v\} \in E$ (i.e., there's an edge from u to v). Is u covered by E''?

   ◯ ALWAYS

   ◯ SOMETIMES

   ◯ NEVER

5.1 Quiz Solution

1. This greedy algorithm **ALWAYS** produces an edge cover (whether or not it is minimal).

   To see that, try imagining that some vertex will not be covered when the algorithm finishes. Well, no vertex becomes uncovered again after being covered. So, when that vertex is picked at some iteration of the loop, it must have been uncovered. But then (whether or not it has an uncovered neighbor) it gets covered!

2. This greedy algorithm **SOMETIMES** produces a minimal edge cover.

   Left as an exercise for you to find examples!

3. Big-O Bound: $O(n \lg n + m)$

   We sort in pre-processing. Then, over the course of all loop iterations, we consider each edge once (well, twice if you prefer to think of it that way, once from each side).

4. E'' is itself a minimal edge cover when E'' is a perfect matching.

   A perfect matching ensures every vertex is matched, which means every vertex is incident on an edge in the matching.

   The word "stable" is meaningless in this problem.

   For the last option, try this graph `a -- b -- c` and see how the numbers work out.

5. u is **ALWAYS** covered by E''.

   Why is that? Feels like something someone should assign themselves to figure out again :)