CSE 3101 Design and Analysis of Algorithms
Solutions for Practice Test for Unit 1
Loop Invariants and Iterative Algorithms

Jeff Edmonds

1. (Answer in slides)
   "And the last shall be first!"
   (For each of A-K see the figure on the next page labeled with the corresponding letter. Here $n = 6$.)

   A: Precondition; The input to this problem is a linked list of n nodes in which the last node points back to the first forming a circle. The variable *last* points at the "last" node in the list. The nodes are of type $Node$ containing a field $info$ and a field *next* of type $Node$ which is to point at the next node in the list. The values $1, 2, \ldots, 6$ in figure are just to help you and cannot be used in the code.

   B: Postcondition; The required output to this problem consists of the same nodes in the same circle, except each node now points at the "previous" node instead of the "next". In other words the pointers are turned from clockwise to counter clockwise. The variable *last* still points at the same node, but this "last" node has become "first".

   C: Loop Invariant$_t$; Your algorithm for this problem will be iterative (i.e. a loop taking one step at a time). Your first task is to give the loop invariant. This is not done in words but by **drawing** what the data structure will look like after the algorithm has executed it's loop $t = 2$ times. Hint: Nodes with value 1 and 2 have their pointers fixed. Hint: You may need a couple of additional pointers to hold things in place. Give them as meaningful names as possible.

   D: Loop Invariant$_{t+1}$; The next task in developing an iterative algorithm is:

   > - Maintain the Loop Invariant:
   > We arrived at the top of the loop knowing only that the Loop Invariant is true and the Exit Condition is not.
   > We must take one step (iteration) (making some kind of progress).
   > And then prove that the Loop Invariant will be true when we arrive back at the top of the loop.
   > $\langle loop{-}invariant_t \rangle$ & $not\, \langle exit{-}cond \rangle$ & $code_{loop} \Rightarrow \langle loop{-}invariant_{t+1} \rangle$

   Towards this task, **draw** what the data structure will look like after the algorithm has executed it's loop one more time, i.e. $t+1 = 3$ times.

   E: Code C to D; In space E, give me the code to change the data structure in figure C into that in figure D. This will be the code within the loop that makes progress while maintaining the loop invariant. Assume every variable and field is public.

   F: Loop Invariant$_0$; The next task in developing an iterative algorithm is:

   > - Establish the Loop Invariant:
   > Our computation has just begun and all we know is that we have an input instance that meets the Pre Condition.
   > Being lazy, we want to do the minimum amount of work.
   > And to prove that it follows that the Loop Invariant is then true.
   > $\langle pre{-}cond \rangle$ & $code_{pre\text{-}loop} \Rightarrow \langle loop{-}invariant \rangle$

   Towards this task, **draw** what the data structure will look like when the algorithm is at the top of the loop but has has executed this loop zero times. Make it as similar as possible to the precondition so that minimal work needs to be done in G.

   G: Code A to F; In space G, give me the code to change the data structure in figure A into that in figure F. This will be the initial code that establishes the loop invariant.

   H: Loop Invariant$_n$; The next task in developing an iterative algorithm is:

- Obtain the Post Condition:
We know the Loop Invariant is true because we have maintained it.
We know the Exit Condition is true because we exited.
We do a little extra work.
And then prove that it follows that the Post Condition is true.
$\langle loop{-}invariant \rangle$ & $\langle exit{-}cond \rangle$ & $code_{post\text{-}loop} \Rightarrow \langle post{-}cond \rangle$

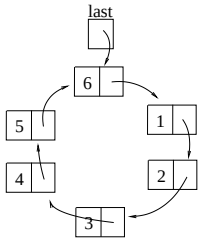
Towards this task, **draw** what the data structure will look like after the algorithm has executed it's loop n times. Make it as similar as possible to the postcondition so that minimal work needs to be done in J.

I: Exit Condition: What is the exit condition, i.e. how does your code recognize that it is in the state you drew in H, given that the algorithm does not know n (and did not count t) and does not know the values $1, 2, \ldots, 6$. Also be careful that your exit condition does not have you drop out at state D.

J: Code H to B; In space J, give me the code to change the data structure in figure H into that in figure B. This will be the final code that establishes the post condition.

K: Give the complete code; Put all the code together into a routine solving the problem.

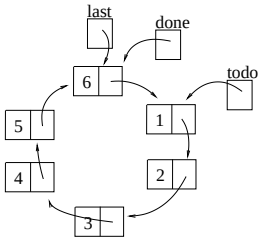
A: Precondition



G: Code A to F

```
Node done = last;
Node todo = last.next;
```

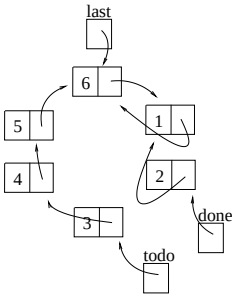
F: Loop Invariant $_0$



K: Give the complete code

```
void ReverseCycle( Node last ) {
    Node done = last;
    Node todo = last.next;
    while( todo.next!=last ) {
        Node temp = todo.next;
        todo.next = done;
        done = todo;
        todo = temp;
    }
}
```

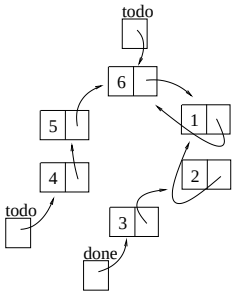
C: Loop Invariant $_t$



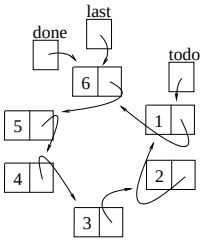
E: Code C to D

```
Node temp = todo.next;
todo.next = done;
done = todo;
todo = temp;
```

D: Loop Invariant $_{t+1}$



H: Loop Invariant $_n$



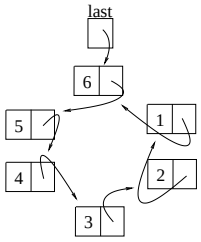
I: Exit Condition

"done==last" is good
but does not differentiate from F
if(todo.next==last) exit;

J: Code H to B

Nothing to do
except return
which will deallocate the memory
for done and todo

B: Post Condition



2. (Answer in slides) You are in the middle of a lake of radius one. You can swim at a speed of one and can run infinitely fast. There is a smart monster on the shore who can't go in the water but can run at a speed of four. Your goal is to swim to shore arriving at a spot where the monster is not and then run away. If you swim directly to shore, it will take you 1 time unit. In this time, the monster will run the distance $\Pi < 4$ around to where you land and eat you. Your better strategy is to maintain

the most obvious loop invariant while increasing the most obvious measure of progress for as long as possible and then swim for it. Describe how this works.

- Answer: I am going to complete the iterative algorithm steps in the order in which I thought of them. The disadvantage of this order is that steps that depend on later steps are not completely complete. But hopefully if you read the solution a second time you will see that each of these steps can be completed fitting them nicely together.
  - Define Loop Invariant:
    The loop invariant that you would like to maintain is that you are diametrically opposed to the monster, i.e. the line between him and you goes through the center.
    I would say this is a *Fairy Godmother* type of loop invariant. I have arrived from Mars, some amount of progress has been made already and my Fairy God Mother has set the state of the computation to be just what I want so that I feel comfortable. OK, maybe I would feel more comfortable if I was safely in my bed but this as comfortable as I could be given the amount of progress that has been made.
  - Establishing the Loop Invariant $\langle pre{-}cond \rangle$ & $code_{pre\text{-}loop} \Rightarrow \langle loop{-}invariant \rangle$:
    Initially, the loop invariant is trivially true because you are at the center.
  - Maintaining the Loop Invariant $\langle loop{-}invariant' \rangle$ & *not* $\langle exit{-}cond \rangle$ & $code_{loop} \Rightarrow \langle loop{-}invariant'' \rangle$:
    Your initial goal is simply to maintain the loop invariant as long as you can. You keep an eye on the monster. If he runs clockwise on shore, then you swim counter clockwise that the rate necessary to maintain that you are diametrically opposed to the monster. If he suddenly switches to running counter clockwise, then you switch your direction too. If you are currently near the shore, then you will have to swim almost as fast as he runs to accomplish this. The problem is that he runs four times faster than you can swim. On the other hand, if you are at the center, then you only need to tread water to accomplish this initial goal. The dividing line between these two extremes is when your current location is at radius $r \leq \frac{1}{4}$ from the center. In such a case, you are able to swim around your circle of radius r at least as fast as he can run around the lake, even though he swims four times faster because your circle has at most $\frac{1}{4}$ the circumference of his. Hence, as long as $r \leq \frac{1}{4}$, you have no problem maintaining the loop invariant.
  - Making Progress:
    Your measure progress is your distance r from the center. Your movements for maintaining the loop invariant move you parallel to the shore and hence do not make you any progress according to this measure. However, if r is strictly less than $\frac{1}{4}$, then you do not need all of your speed to maintain the loop invariant. Use your extra speed to make progress by swimming outward.
  - Define the Step. As said, some of your swimming effort is swimming in the circle perpendicular to the shore in order to keep the center of the lake between you an the monster and your remaining effort is making progress swimming towards the nearest shore. Instead of swimming in a stair case alternating between these two goals you can use linear algebra to combine these two vectors into one vector that moves you in a smooth spiral way from the center. Of course the direction of your spiral may keep changing as the monster gets frustrated and changes his direction.
  - Define Exit Condition: As said in the iterative algorithm steps, one possible reason for exiting the loop is that you are no longer able to maintain the loop invariant. In this case, we fail to make progress when $r \geq \frac{1}{4}$ circle. Hence, we make this our exit condition.
  - Ensuring that we actually Exit:
    The official rules of loop invariant algorithms is that you must make progress of at least one each time step because steps of size $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$ ... will get you close but never quite there.
    Years ago I checked the converging issue and my memory is if you swimming in a stair case alternating between swimming perpendicularly and towards shore, then the amount you have left to swim towards shore is not enough to get you to ever reach $r = \frac{1}{4}$ in a finite amount

of time, i.e. you will get closer can closer but never quite reach it. However, if you use linear algebra to combine these two vectors into one vector that spirals, then you do in fact converge to $r = \frac{1}{4}$ in a finite time. Check this if you like.
To avoid the converging issue, lets set some small number ϵ and exit when our distance from the center is $r \geq \frac{1}{4} - \epsilon$. We know that we will eventually meet this exit condition because as long as $r < \frac{1}{4} - \epsilon$, in each "time unit" we are guaranteed to make some constant amount $\mathcal{O}(\epsilon)$ amount of progress.

- Obtaining the postcondition $\langle loop{-}invariant \rangle$ & $\langle exit{-}cond \rangle$ & $code_{post\text{–}loop} \Rightarrow \langle post{-}cond \rangle$:
By the exit condition $r \geq \frac{1}{4} - \epsilon$. By the loop invariant, the monster is on the opposite shore from your current location. The post loop code is for you to swim as fast as you can directly for shore. This take you $1-r \approx \frac{3}{4}$ time units. During this time the monster can run a distance of 3. However, because he is diametrically opposed to you when you start your sprint, he needs to run a distance of $\Pi > 3$, which is too far for him. You then run to safety meeting the post condition!

In contrast, here is a wrong answer.

- 1) Basic steps: Swim in the opposite direction of monster's current location.
- 2) Loop invariant: You've been swimming to the shore while your head is in the opposite direction of monster's current location.
  * Being a picky marker, I will complain that this sounds more like a description of the algorithm than like a picture of the current state of the computation. I hope this wrong answer help to demonstrate the danger of this. Though this loop invariant does tell you what you have been *doing*, it does not tell you what you need to know, which is where you *currently* are wrt the monster.
- 3) Main steps: Look around and check monster's location while you keep swimming and change the direction if it's needed to keep your direction opposite to monster's location.
  * (Fancy talk for "Panic and swim away from the monster." Certainly this is the most natural thing for a frightened person to do. And I am not trying to prove here that this algorithm does not work (it may). I am trying to prove that this proof that this algorithm works is a faulty proof.
- 4) Measure of progress: distance to the shore is the measure of progress.
  * For sure, this is the right measure of progress.
- 5) Make progress: It makes progress because where ever monster is at shore, if you keep swimming to the shore at the opposite direction of monster's location. You get closer to the shore, which mean the distance to the shore is reduced.
  * But what if he is on the shore you are closest too. Then your algorithm swims you away from shore and you fail to make progress - in fact, just the opposite. You could well swim in circles for ever.
- 6) Maintain loop variant: You have been swimming in the opposite direction of monster's location from the previous loop. Loop invariant is maintained by checking the monster's location and keeping your head in the opposite direction of monster's location while you're swimming.
  * True, you do manage to keep doing this same algorithm and hence because your loop invariant only states that you keep doing this same algorithm, you do in fact maintain the loop invariant (for what ever that is worth).
- 7) Establishing loop invariant: You obtain loop invariant by checking the monster's direction and starting to swim to the opposite direction of monster's location.
  * Certainly a good start.
- 8) Exit condition: distance to the shore is zero.
  * The problem is that this exit condition might never be met.
- 9) Ending: By the exit condition, you are at the shore and by the loop invariant your head is is pointed in the opposite direction of monster's location. The post condition follows.

* Your head my well be pointed in the opposite direction of monster's location, but because your loop invariant says nothing about where the monster is wrt your current location, he may well be standing right beside you. On the other hand, it may be a good thing that your head is facing in the opposite direction because then you might aware of moment that he eats you.

3. Iterative Cake Cutting: The famous algorithm for fairly cutting a cake in two is for one person to cut the cake in the place that he believes is half and for the other person to choose which "half" he likes. One player may value the icing more while the other the cake more, but it does not matter. The second player is guaranteed to get a piece that he considers to be worth at least a half because he choose between two pieces whose sum worth for him is at least a one. Because the first person cut it in half according to his own criteria, he is happy which ever piece is left for him. Our goal is write an iterative algorithm which solves this same problem for n players.

   To make our life easier, we view a cake not as three dimensional thing, but as the line from zero to one. Different players value different subintervals of the cake differently. To express this, he assigns some numeric value to each subinterval. For example, if player p_i's name is written on the subinterval $[\frac{i-1}{2n}, \frac{i}{2n}]$ of cake then he might allocate a higher numeric value to it, say $\frac{1}{2}$. The only requirement is that the sum total value of the cake is one.

   Your algorithm is only allowed the following two operations. In an evaluation query, $v = Eval(p, [a, b])$, the algorithm asks a player p how much v he values a particular subinterval $[a, b]$ of the whole cake $[0, 1]$. In a cut query, $b = Cut(p, a, v)$, the protocol asks the player p to identify the shortest subinterval $[a, b]$ starting at a given left endpoint a, with a given value v. In the above example, $Eval(p_i, [\frac{i-1}{2n}, \frac{i}{2n}])$ returns $\frac{1}{2}$ and $Cut(p_i, \frac{i-1}{2n}, \frac{1}{2})$ returns $\frac{i}{2n}$. Using these the two player algorithm is as follows.

   algorithm $Partition2(\{p_1, p_2\}, [a, b])$

   ⟨*pre−cond*⟩: p_1 and p_2 are players.
   $[a, b] \subseteq [0, 1]$ is a subinterval of the whole cake.

   ⟨*post−cond*⟩: Returns a partitioning of $[a, b]$ into two disjoint pieces $[a_1, b_1]$ and $[a_2, b_2]$ so that player p_i values $[a_i, b_i]$ at least half as much as he values $[a, b]$.

   begin
   $v_1 = Eval(p_1, [a, b])$
   $c = Cut(p_1, a, \frac{v_1}{2})$
   if($Eval(p_2, [a, c]) \leq Eval(p_2, [c, b])$) then
   $[a_1, b_1] = [a, c]$ and $[a_2, b_2] = [c, b]$
   else
   $[a_1, b_1] = [c, b]$ and $[a_2, b_2] = [a, c]$
   end if
   return($[a_1, b_1]$ and $[a_2, b_2]$)
   end algorithm

   The problem that you must solve is the following

   algorithm $Partition(n, P)$

   ⟨*pre−cond*⟩: P is a set of n players.
   Each player in P values the whole cake $[0, 1]$ by at least one.

   ⟨*post−cond*⟩: Returns a partitioning of $[0, 1]$ into n disjoint pieces $[a_i, b_i]$ so that for each $i \in P$, the player p_i values $[a_i, b_i]$ by at least $\frac{1}{n}$.

   begin
   ...
   end algorithm

(a) Can you cut off n pieces of cake, each of *size* strictly bigger than $\frac{1}{n}$, and have cake left over? Is it sometimes possible to allocated a disjoint piece to each player, each worth by the receiving player much more than $\frac{1}{n}$, and for there to still be cake left? Explain.

(b) Give **all** the required steps to describe this LI algorithm. (Even if you do not know how to do a step for this algorithm, minimally state the step.)

As a big hint to designing an iterative algorithm, we will tell you what the first iteration accomplishes. (Later iterations may do slightly modified things.) Each player specifies where he would cut if he were to cut off the first $\frac{1}{n}$ fraction of the $[a, b]$ cake. The player who wants the smallest amount of this first part of the cake is given this piece of the cake. The code for this is as follows.

loop $i \in P$
  $c_i = Cut(p_i, 0, \frac{1}{n})$
end loop
i_{min} = the $i \in P$ that minimizes c_i
  $[a_{i_{min}}, b_{i_{min}}] = [0, c_{i_{min}}]$

As a second big hint, your loop invariant should include:

i. How the cake has been cut so far.
ii. Who has been given cake and how do they feel about it.
iii. How do the remaining players feel about the remaining cake.

- Answer:

(a) Extra question: Clearly the first task is impossible because summing up n things of size more than $\frac{1}{n}$ gives you at a total more than one. There can be none left over. However, the second is possible. Consider the example given above in which for each i, $Eval(p_i, [\frac{i-1}{2n}, \frac{i}{2n}])$ returns $\frac{1}{2}$. This subinterval $[\frac{i-1}{2n}, \frac{i}{2n}]$ can be allocated to this player. These intervals are disjoint, valued much more than $\frac{1}{n}$ by this player, and the interval $[\frac{1}{2}, 1]$ is left over.

(b) Define Loop Invariant:

i. First remember the *more of the output* type of loop invariant. This would be as follows. There is a set of players Q that have been served. The beginning $[0, a]$ of the cake has been partitioned into $|Q|$ disjoint pieces. Each player $p_i \in Q$ has been allocated a piece $[a_i, b_i]$ worth at least $\frac{1}{n}$ to him.

ii. Now remember the *Fairy God Mother* type of loop invariant. I have arrived from Mars, some amount of progress has been made already and my Fairy God Mother has set the state of the computation to be just what I want so that I feel as comfortable as I could be given the amount of progress that has been made. This would be as follows.
The remaining $[a, 1]$ interval of the cake is worth at least least $\frac{n-|Q|}{n}$ to each of the remaining players, i.e. to those in $P - Q$.

This is a more of the output loop invariant if you think of the output being ordered by who gets the first piece, who gets the second, and so on.

(c) Establishing the Loop Invariant $\langle pre{-}cond \rangle$ & $code_{pre\text{-}loop} \Rightarrow \langle loop{-}invariant \rangle$:
With $a = 0$ and $Q = \emptyset$, the beginning zero $[0, 0]$ of the cake has been partitioned into disjoint pieces and allocated to $|Q| = 0$ of the players. All $[0, 1]$ of the cake remains and by the precondition is worth at least least $\frac{n-|Q|}{n} = 1$ to each of the players.

(d) Define the Step (actually I will give the entire code):
algorithm $Partition(P)$
$\langle \boldsymbol{pre{-}cond} \rangle$**:** As above
$\langle \boldsymbol{post{-}cond} \rangle$**:** As above
begin
  $a = 0$ and $Q = \emptyset$
  loop

$\langle$***loop−invariant***$\rangle$**:** As above.

exit when $|Q| = n$
loop $i \in P - Q$
$c_i = Cut(p_i, a, \frac{1}{n})$
end loop
i_{min} = the $i \in P - Q$ that minimizes c_i
$[a_{i_{min}}, b_{i_{min}}] = [a, c_{i_{min}}]$
$a = c_{i_{min}}$
$Q = Q + i_{min}$
end loop
return all parts $[a_i, b_i]$ for each $i \in P$
end algorithm

(e) Maintaining the Loop Invariant $\langle loop{-}invariant' \rangle$ & *not* $\langle exit{-}cond \rangle$ & $code_{loop} \Rightarrow \langle loop{-}invariant'' \rangle$:
Let a' and P' be the values at the top of the loop and let a'' and Q'' be the values after going around again. By the code, $a'' = c_{i_{min}}$ and $Q'' = Q' + i_{min}$.

i. By $\langle loop{-}invariant' \rangle$ the beginning $[0, a']$ of the cake has been partitioned into $|Q'|$ disjoint pieces. By the code, we just partitioned off the piece $[a', a'']$. Hence, the beginning $[0, a'']$ of the cake has been partitioned into $|Q'| + 1 = |Q''|$ disjoint pieces.

ii. By $\langle loop{-}invariant' \rangle$, each player $p_i \in Q'$ has been allocated a piece worth at least $\frac{1}{n}$ to him. We just just allocated a piece of this value to another player and added him to Q'. Hence, each player $p_i \in Q''$ has been such a piece.

iii. Consider a remaining player p_i in $P - Q'' = P - Q' - i_{min}$. Because he marked it, we know that this player values the piece $[a', c_i]$ by $\frac{1}{n}$. The piece $[a', c_{i_{min}}] = [a', a'']$ cut off is smaller than this, because $c_{i_{min}}$ is smaller than c_i. Hence, player p_i values the piece $[a, a'']$ by at most $\frac{1}{n}$. By $\langle loop{-}invariant' \rangle$, the remaining $[a', 1]$ interval of the cake is worth at least $\frac{n-|Q'|}{n}$ to p_i. Hence, $[a'', 1] = [a', 1] - [a', a'']$ has value of at least $\frac{n-|Q'|}{n} - \frac{1}{n} = \frac{n-|Q''|}{n}$.

If follows that $\langle loop{-}invariant'' \rangle$ is true and that the loop invariant has been maintained.

(f) Define Exit Condition: When every player has a piece we exit.

(g) Obtaining the postcondition $\langle loop{-}invariant \rangle$ & $\langle exit{-}cond \rangle$ & $code_{post\text{–}loop} \Rightarrow \langle post{-}cond \rangle$:
By the exit condition $|Q| = n$. By the loop invariant, the beginning $[0, a]$ of the cake has been partitioned into $|Q| = n$ disjoint pieces and each player $p_i \in Q = P$ has been allocated a piece $[a_i, b_i]$ worth at least $\frac{1}{n}$ to him. Though there may be some cake left over, the post condition is satisfied.

(h) Measure of Progress: The measure of progress is number of pieces allocated. Each iteration one piece is allocated. Initially, there are only n pieces to be allocated. Hence, with sufficient progress, the exit condition will be met. The i^{th} of the n iterations takes $\mathcal{O}(n-i)$ time for a total running time of $\mathcal{O}(n^2)$.

4. Tiling Chess Board: You are given a $2n$ by $2n$ chess board. You have many tiles each of which can cover two adjacent squares. Your goal is to place non-overlapping tiles on the board to cover each of the $2n \times 2n$ tiles except for to top-left corner and the bottom-right corner. Prove that this is impossible. To do this give a loop invariant that is general enough to work for any algorithm that places tiles. Hint: chess boards color the squares black and white.

- Answer:

  Precond: The board is empty.

  Postcond: All but the corner two white tiles is covered.

  Define Loop Invariant: The number of blacks and white tiles covered is equal.

Establishing the Loop Invariant: $\langle pre-cond \rangle$ & $code_{pre-loop}$ $\Rightarrow$ $\langle loop-invariant \rangle$: By $\langle pre-cond \rangle$ there are zero white and zero blacks covered. Hence, these numbers are the same.

Step: Place a tile. This covers one white and one black square.

Maintaining the Loop Invariant: $\langle loop-invariant' \rangle$ & *not* $\langle exit-cond \rangle$ & $code_{loop}$ $\Rightarrow$ $\langle loop-invariant'' \rangle$: By $\langle loop-invariant' \rangle$, at the beginning of the iteration the number of white and black tiles covered are equal. The step covers one more white and one more black. Hence, after the iteration the numbers are still equal.

Exit: Exit when ever you like.

Ending: Obtaining that the postcondition is NEVER met.
i.e. $\langle loop-invariant \rangle \Rightarrow \neg \langle post-cond \rangle$
Prove the contra positive,
i.e. $\langle post-cond \rangle \Rightarrow \neg \langle loop-invariant \rangle$:
Assume that the post condition is true. It states that all but the corner two white tiles is covered. However, the board has an equal number of black and white tiles. Hence, the number of white and black tiles covered are not equal. Hence, the LI is not true.

5. Loop Invariants - Sorted Matrix Search: Search a Sorted Matrix Problem: The input consists of a real number x and a matrix $A[1..n, 1..m]$ of nm real numbers such that each row $A[i, 1..m]$ is sorted from left to right and each column $A[1..n, j]$ is sorted from top to bottom. The goal is to determine if the key x appears in the matrix. Design and analyze an iterative algorithm for this problem that examines as few matrix entries as possible. Careful, if you believe that a simple binary search solves the problem. Later we will ask for a lower bound and for a recursive algorithm.

   - Answer: Doing binary search in $\mathcal{O}(\log(n \times m))$ time is impossible. See the lower bound question. If you take $\mathcal{O}(n \log m)$ time doing binary search in each row than you are taking too much time. It can be done by examining $n + m - 1$ entries. Observe that the values in the matrix increase from $A[1, 1]$ to $A[n, m]$. Hence, the boundary between values that are less than or equal to x and those that are greater follows some monotonic path from $A[1, m]$ to $A[n, 1]$. The algorithm traces this path starting at $A[1, m]$. When it is at the point $A[i, j]$, the loop invariant is that we have stored the best answer from those outside of the sub-rectangle $A[i..n, 1..j]$. Initially, this is true for $[i, j] = [1, m]$, because none of the matrix is excluded. Now suppose it is true for an arbitrary $[i, j]$. The algorithm then compares $A[i, j]$ with x. If it is better than our current best answer then our current best is replaced. If $A[i, j] \leq x$, then because the values in the row $A[i, 1..j]$ are all smaller or equal to $A[i, j]$, these are worse answers and hence we can conclude that we now have the best answer from those outside of the sub-rectangle $A[i + 1..n, 1..j]$. We maintain the loop invariant by increasing i by one. On the other hand, if $A[i, j] > x$, then it is too big and so are all the elements in the column $A[i..n, j]$ which are even bigger. We can conclude that we have the best answer from those outside of the sub-rectangle $A[i..n, 1..j - 1]$. We maintain the loop invariant by decreasing j by one. The exit condition is $|i..n| = 0$ or $|1..j| = 0$ (i.e. $i > n$ or $j < 1$. When this occurs, the sub-rectangle $A[i..n, 1..j]$ is empty. Hence, our best answer, which by the loop invariant is the best from those outside of the this sub-rectangle, must be the best overall. The measure of progress $|i..n| + |1..j| - 1 = (n - i + 1) + (j) - 1$ is initially $n + m - 1$ and decrease by one each iteration. After $n + m - 1$ iterations, either the algorithm has already halted or the measure has reached zero at which point the exit condition is definitely met.

6. $d + 1$ Colouring: Given an undirected graph G such that each node has at most d neighbors, colour each node with one of $d + 1$ colours so that for each edge the two nodes have different colours. Hint: Don't think too hard. Just colour the nodes. What loop invariant do you need?
   Hint: All it will say is that you have not gone wrong yet.
   Give all the steps done in class to develop this iterative algorithm.

   - Answer:

Loop Invariant The only loop invariant that you need is that after colouring i nodes, you have not yet coloured the two nodes of an edge with the same colour.

Establishing the Loop Invariant $\langle pre{-}cond\rangle$ & $code_{pre\text{-}loop}$ $\Rightarrow$ $\langle loop{-}invariant\rangle$:
$code_{pre\text{-}loop}$: Initially, $i = 0$ nodes have been coloured, so the LI is trivially true.

Steps: Grab any uncoloured node v. It has at most d neighbors some of which might already be coloured. However, at most d different colours different colours can be already used by these neighbors. Colour v with one of the other $d + 1$ colours.

Maintaining the Loop Invariant $\langle loop{-}invariant'\rangle$ & not $\langle exit{-}cond\rangle$ & $code_{loop}$ $\Rightarrow$ $\langle loop{-}invariant''\rangle$:
By the LI', we had coloured i nodes without any bichromatic edges. We proved that the step manages to colour and $i + 1^{st}$ node so that it has a different colour than its neighbors. This reestablishes the LI with $i + 1$ nodes coloured.

Measure of Progress and Time: The measure of progress is the number of coloured nodes. Each of the n iterations will take at most $O(d)$ time for a total of at most $O(dn)$ time.

Obtaining the postcondition $\langle loop{-}invariant\rangle$ & $\langle exit{-}cond\rangle$ & $code_{post\text{-}loop}$ $\Rightarrow$ $\langle post{-}cond\rangle$:
By exit condition, $i = n$.
By LI, all these nodes have been coloured without bichromatic edges.
Hence we have coloured the entire graph without them. This establishes the post condition.

7. Loop Invariants - Connected Components: The input is a matrix I of pixels. Each pixel is either black or white. A pixel is considered to be connected to the four pixels left, right, up, and down from it, but not diagonal to it. The algorithm must allocated a unique name to each connected black component. (The name could simply be 1,2,3,..., to the number of components.) The output consists of another matrix $Names$ where $Names(x, y) = 0$ if the pixel $I(x, y)$ is white and $Names(x, y) = i$ if the pixel $I(x, y)$ is a part of the component named i. The algorithm reads the matrix from a tape row by row as follows.

```
loop y = 1...h
    loop x = 1...w
        ⟨loop−invariant⟩: ??
        if( I(x,y)=white )
            Names(x,y) = 0
        else
            ???
        end if
    end loop
end loop
end algorithm
```

The image may contain spirals and funny shapes. Connected components may contain holes that contain other connected components. A particularly interesting case is the image of a comb consisting of many teeth held together at the bottom by a handle. Scanning the image row by row, one would would first see each tooth as a separate component. As the handle is read, these teeth would merge into one.

(a) Give the classic *more of the input loop invariant* algorithm. Don't worry about its running time.

- Answer: Assuming that the input image consists only of the part of the image read in so far, the algorithm has named the connected components. This loop invariant is maintained while making progress as follows. The next pixel $I(x, y)$ is read. The white case is easy, so suppose it is black. This pixel is connected to two pixels that have been read before: $I(x-1, y)$ to its left and $I(x, y-1)$ above it. If these are both white, then $I(x, y)$ is in a new component by itself and hence is given the new name never used before. If only one of $I(x-1, y)$ and

$I(x, y-1)$ is black or the two are black but in the same component, then $I(x, y)$ become a part of this component. Finally, if $I(x-1, y)$ and $I(x, y-1)$ are both black and in the two different components, then the new pixel $I(x, y)$ joins these two components into one. Suppose $I(x-1, y)$ has been named i and $I(x, y-1)$ named i'. Every pixel in the component named i' is renamed to i.

$maxName = numberOfComponents = 0$
loop $y = 1 \ldots h$
loop $x = 1 \ldots w$
⟨***loop−invariant***⟩**:** More of the input.
if($I(x, y)$=white)
$Names(x, y) = 0$
else
if($Names(x, y-1) = 0$ and $Names(x-1, y) = 0$)
$++maxName$
$++numberOfComponents$
$Names(x, y) = maxName$
elseif($Names(x, y-1) = 0$ or $Names(x-1, y) = 0$ or ($Names(x, y-1) = Names($
$Names(x, y)$ = the nonzero one of $Names(x, y-1)$ and $Names(x-1, y)$
else
$i = Names(x, y-1)$
$i' = Names(x-1, y)$
$Names(x, y) = i$
Replace every i' in $Names$ with i.
$--numberOfComponents$
end if
end if
end loop
end loop
end algorithm

(b) This version of the question is easier in that the matrix $Names$ need not be produced. The output is simply the number of connected black components in the image. However, this version of the question is harder in your computation is limited in the amount of memory it can use. For example, you don't remember pixels that you have read if you do not store them in this limited memory and you don't have nearly enough memory to store them all. The number of components may be Θ(the pixels) so you cant store all of them either. How little memory can you get away with?

- Answer: The amount of memory needed is $\Theta(w)$. The algorithm will be the same except it only keeps the last two rows worth of $Names$. Note $numberOfComponents$ has already been added to the code to count the number of components.

(c) In this this final version, in addition to a small amount of fast memory, you have a small number of tapes for storing data. Data access on tapes, however, is quite limited. A tape is in one of three modes and cannot switch modes mid operation. In the first mode, the data on a tape is read forwards one data item at a time in order. The second mode, is the same except it is read backwards. In the third mode, the tape is written to. However, the command is simply $write(data)$ which appends this data to the end of the tape. Data on a tape cannot be changed. All you can do is to erase the entire tape and start writing again from the beginning. An algorithm must consist of a number of passes. The first pass reads the input from the input tape one pixel at a time row by row in order. As it goes, the algorithm updates what is store in fast memory and outputs data in order onto a small number output tapes. Successive passes can read what was written during a previous pass and/or the input again. The last pass must write the required output $Names$ onto a tape. You want to use as little fast memory and as few passes as possible. For each pass clearly state the loop invariant.

- Answer: The algorithm will have two passes, two intermediate tapes, and $\Theta(w)$ fast memory. During the first pass, the algorithm will again be the same. As in the last algorithm, it only keeps the last two rows worth of $Names$ in the fast memory. As rows of $Names$ is deleted from fast memory it is written to a tape $Names_{firsttry}$. The problem is that as components named i' are renamed to i, this updating will not occur on the tape. Hence, $Names_{firsttry}$ may have many names associated with each component. To keep track of these repeated names, every time the algorithm would like to rename i' to i, the instruction $\langle i' \Leftarrow i \rangle$ is written to a second tape $Instructions$.
  The loop invariant for this first pass states that what is in the fast memory is exactly the same as it would be in the second algorithm's memory and if all the instructions $\langle i' \Leftarrow i \rangle$ in $Instructions$ are followed, then what is currently in the tape $Names_{firsttry}$ would be converted into what would be in the first algorithm's $Names$. More over, the instruction $\langle i' \Leftarrow i \rangle$ appears in the location of the tape $Instructions$ corresponding to the last place in $Names_{firsttry}$ that the name i' appears. Hence, if one reads both tapes $Names_{firsttry}$ and $Instructions$ backwards, then one would always have the required set of instructions to do the required conversion.
  The second pass reads these two tapes backwards. Its loop invariant is that it has already correctly converted the part of $Names_{firsttry}$ that is has seen and stored it backwards in the final output tape $Names$. The loop invariant also states that fast memory contains the needed instructions to convert the next pixel. The step is to read the next pixel from $Names_{firsttry}$, read any new instructions from $Instructions$, convert this next pixel using the current set of instructions, and write the converted pixel into $Names$.
  One concern might be that the number of instructions read may exceed the size of the fast memory. However, as soon as a name i' no longer appears in a row of $Names_{firsttry}$, we know that it will never appear again earlier in $Names_{firsttry}$. Hence, at this point the instruction $\langle i' \Leftarrow i \rangle$ can be deleted. This ensures, that the number of instructions in fast memory never exceeds the number w of pixels in a row.

8. A *tournament* is a directed graph (see Section **??**) formed by taking the complete undirected graph and assigning arbitrary directions on the edges, i.e., a graph $G = (V, E)$ such that for each $u, v \in V$, exactly one of $\langle u, v \rangle$ or $\langle v, u \rangle$ is in E. A *Hamiltonian path* is a path through a graph that can start and finish any where but must visit every node exactly once each. Design an algorithm which finds a Hamiltonian path through it given any tournament. Because this algorithm finds a Hamiltonian path for each tournament, this algorithm, in itself, acts as proof that every tournament has a Hamiltonian path.

   - Answer:

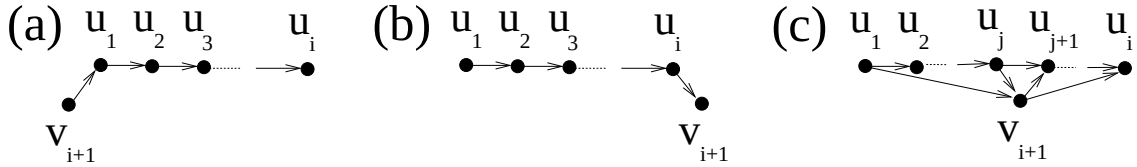
     More of the Output: It is natural to want to push forward and find the required path through a graph. The measure of progress would be the amount of the path output and the loop invariant would say "I have the first i nodes (edges) in the final path." Maintaining this loop invariant would require extending the path constructed so far by one more node. The problem, however, is that the algorithm might get stuck, when the path constructed so far has no edges leaving the last node to a node that has not yet been visited. This makes the loop invariant as stated false.

     Recursive Backtracking: One is then tempted to have the algorithm "backtrack" when it gets stuck trying in a different direction for the path to go. This is a fine algorithm. See recursive backtracking algorithms in Chapter **??**. However, unless one is really careful, such algorithms tend to require exponential time.

     More of the Input: Instead, try solving this problem using a "more of the input" loop invariant. Assume the nodes are numbered 1 to n in an arbitrary way. The algorithm temporarily pretends that the sub-graph on the first i of the nodes is the entire input instance. The loop invariant is "I currently have a solution for this sub-instance." Such a solution is a Hamiltonian path $u_1, \ldots, u_i$ that visits each of the first i nodes exactly once each, which

in turn is simply a permutation the first i nodes. Maintaining this loop invariant requires constructing a path for the first $i+1$ nodes. There is no requirement that this new path resembles the previous path. However, for this problem, it can be accomplished by finding a place to insert the $i+1^{st}$ node within the permutation of the first i nodes. In this way, the algorithm looks a lot like insertion sort.

Case Analysis: When developing an algorithm, a good technique is to see for which input instances the obvious thing works and then try to design another algorithm for the remaining cases.



(a) If $\langle v_{i+1}, u_1 \rangle$ is an edge, then the extended path is easily $v_{i+1}, u_1, \ldots, u_i$.

(b) Similarly, if $\langle u_i, v_{i+1} \rangle$ is an edge, then the extended path is easily $u_1, \ldots, u_i, v_{i+1}$.

(c) Otherwise, because the graph is a tournament, both $\langle u_1, v_{i+1} \rangle$ and $\langle v_{i+1}, u_i \rangle$ are edges. Color each node u_j red if $\langle u_j, v_{i+1} \rangle$ is an edge and blue if $\langle v_{i+1}, u_j \rangle$ is. Because u_1 is red and u_i is blue, there must be some place u_j to u_{j+1} in the path where path changes color from red to blue. Because both $\langle u_j, v_{i+1} \rangle$ and $\langle v_{i+1}, u_{j+1} \rangle$ are edges, we can form the extended path $u_1, \ldots, u_j, v_{i+1}, u_{j+1}, \ldots, u_i$.

9. An *Euler tour* in an undirected graph is a cycle that passes through each edge exactly once. A graph contains an Eulerian cycle iff it is connected and the degree of each vertex is even. Given such a graph find such a cycle.

- Answer:

More of the Output: We will again start by attempting to solve the problem using the more the output technique, namely, start at any node and build the output path one edge at a time. Not having any real insight into which edge should be taken next, we will choose them in a blind or "greedy" way (see Chapter **??**). The loop invariant is that after i steps you have some path through i different edges from some node s to some node v.

Getting Stuck: The next step in designing this algorithm is to determine when, if ever, this simple blind algorithm gets stuck and to either figure out how to avoid this situation or to fix it.

Making Progress: If $s \neq v$, then the end node v must be adjacent to an odd number of edges that are in the path. See Figure **??**.a. This is because there is the last edge in the path and then for every edge in the path coming into the node there is one leaving. Hence, because v has even degree it follows that v is adjacent to at least one edge that is not in the path. Follow this edge extending the path by one edge. This maintains the loop invariant while making progress. This process can only get stuck when the path happens to cycle around back to the starting node giving $s = v$. In such a case, join the path here to form a cycle.

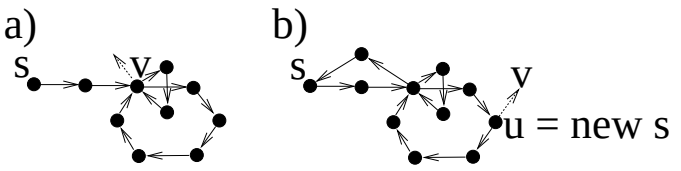


Figure 1: Euler Algorithm

Ending: If the cycle created covers all of the edges, then we are done.

Getting Unstuck: If the cycle we have created from our chosen node s back to s does not cover all the edges, then we look for a node u with in this cycle that is adjacent to an edge not in the cycle. See Figure **??**.b. Change s to be this new node u. We break the cycle at u giving us a path from u back to u. The difference with this path is that we can extend it past u along this unvisited edge. Again the loop invariant has been maintained while making progress.

***u* Exists:** The only thing remaining to prove is that when v comes around to meet s again and we are not done, then there is in fact a node u in the path that is adjacent to an edge not in the path. Because we are not done then there is an edge e in the graph that is not in our path. Because the graph is connected, there must be a path in the graph from e to our constructed path. The node u at which this connecting path meets our constructed path must be as required because the last edge $\{u, w\}$ in the connecting path is not in our constructed path.

Extended Loop Invariant: To avoid having to find such a node u when it is needed, we extend the loop invariant to state that in addition to the path, the algorithm remembers some node u other than s and v that is in the path and is adjacent to an edge not in the path.

10. (Answer in slides) The ancient Egyptians and the Ethiopians had advanced mathematics. Merely by halving and doubling, they could multiply any two numbers correctly. Say they want to buy 15 sheep at 13 Ethiopian dollars each. Here is how he figures out the product. Put 13 in a left column, 15 on the right. Halve the left value; you get $6\frac{1}{2}$. Ignore the $\frac{1}{2}$. Double the right value. Repeat this (keeping all intermediate values) until the left value is 1. What you have is

13	15
6	30
3	60
1	120

Even numbers in the left column are evil and, according the story, must be destroyed, along with their guilty partners. So scratch out the 6 and its partner 30. Now add the right column giving $15 + 60 + 120 = 195$, which is the correct answer. Give all the required steps to describe this LI algorithm. The question sheet has some hints.

- Answer:

  Specification: An input instance consists of two positive integers x and y. The output is their product.

  *** Define Loop Invariant:** The loop invariant is $\ell \times r + s = x \times y$. It is similar to the Shrinking Instance invariant.

  *** Establishing the Loop Invariant:** $\langle pre{-}cond \rangle$ & $code_{pre\text{-}loop} \Rightarrow \langle loop{-}invariant \rangle$:
  - $\langle pre{-}cond \rangle$ assures us that we have two integers x and y.
  - The code before the loop sets $\ell = x$, $r = y$, and $s = 0$.
  - This trivially establishes the loop invariant by giving $\ell \times r + s = x \times y + 0$.

  Picture: Draw an ℓ by r rectangle and an area of s on the side. The first step consists of removing one row (with area $1 \times r$) from the rectangle and moving that area to the side. The second step consists of cutting the rectangle in half top to bottom and moving the top half to the right side of the bottom half. Both of these steps keep the total area the same.

  Step: Define the Step (actually all the code):

  algorithm $EthiopianMultipling\,(x, y)$

  $\langle \boldsymbol{pre{-}cond} \rangle$**:** x and y are positive integers.

  $\langle \boldsymbol{post{-}cond} \rangle$**:** Outputs $x \times y$.

  begin

  $\ell = x$, $r = y$, $s = 0$

  loop $\langle \boldsymbol{loop{-}invariant} \rangle$**:** $\ell \times r + s = x \times y$.

exit when $\ell = 0$
if(ℓ is odd) then
$\ell = \ell - 1$
$s = s + r$
end if

$\langle$***loop−invariant***$\rangle$**:** $\ell \times r + s = x \times y$.

$\ell = \ell/2$
$r = 2 \times r$
end loop
return(s)
end algorithm

Maintaining the Loop Invariant $\langle loop{-}invariant' \rangle$ & *not* $\langle exit{-}cond \rangle$ & $code_{loop} \Rightarrow \langle loop{-}invariant'' \rangle$:
Let ℓ', r', s' be the values when at the top of the loop and assume that $\ell' \times r' + s' = x \times y$. In the first step, if ℓ' is odd then $\ell'' = \ell' - 1$ and $s'' = s' + r'$. This gives that $\ell'' \times r'' + s'' = (\ell' - 1) \times r' + (s' + r') = \ell' \times r' + s'$, which by the loop invariant is $x \times y$.
In the second step, $\ell''' = \ell''/2$ and $r''' = 2r''$. This gives that $\ell''' \times r''' + s''' = (\ell''/2) \times (2r'') + s'' = \ell'' \times r'' + s''$, which by the loop invariant is $x \times y$.

Exit: The Ethiopians exit when $\ell = 1$. But this being odd, they must add r to s. We will iterate one more time and exit when $\ell = 0$.

*** Ending:** Obtaining the postcondition $\langle loop{-}invariant \rangle$ & $\langle exit{-}cond \rangle$ & $code_{post\text{–}loop} \Rightarrow \langle post{-}cond \rangle$:
The loop invariant gives $\ell \times r + s = x \times y$ and the exit condition gives that $\ell = 0$. This gives $s = x \times y$. The code returns s.

Termination: The measure of progress can be the value of ℓ or even more elegantly $\log_2 \ell$. Progress is made each iteration because the value of ℓ is cut in half and hence $\log_2 \ell$ goes down by one. Initially, $\log_2 \ell$ is initially finite. With sufficient progress, ℓ will be one or zero and the exit condition will be met.

Time Complexity: The "size" of the input is $n = \log_2 x + \log_2 y$, which is the number of bits to write down x and y. The algorithm iterates $\log_2(x) = n$ times. No value increases beyond $x \times y = 2^n \times 2^n = 2^{2n}$ and hence is at most a $2n$ bit number, giving that all additions take $\Theta(n)$ bit operations. Hence, the total algorithm uses at most $\Theta(n^2)$ bit operations, which is quadratic in the size $2n$ of the input. This time equivalent to that needed for the high school algorithm for multiplying.
If pebbles are used then eventually the answer $x \times y$ is counted out in pebbles. This alone requires $\Omega(x \times y) = \Omega(2^{2n})$ pebble operations. This is exponential in the input size $2n$. But if the input is given to us in pebbles, it might be more fair to call the "size" the number of pebbles, which is $x + y$. In which, case the time $x \times y$ is also quadratic in the input size. Note this is equivalent in time to lay out a rectangle of x by y pebbles.

A Comparison: Though this algorithm seems strange, it is in fact exactly the same as the high school algorithm for multiplying in binary. Lets us first understand multiplication $x \times y$ in terms of bits. Let $x_4x_3x_2x_1x_0$ denote the bits of x written in binary. For example $x = 58 = 11001_2$. The bit x_i is shifted into place by multiplying it by 2^i. This gives $x = \sum_i 2^i x_i$. Plugging this in gives that $x \times y = \left[\sum_i 2^i x_i\right] \times y = \sum_i \left[2^i x_i y\right]$.

```
Y   111010
X      101
     111010     2^0 Y X_0
    000000      2^1 Y X_1
   111010       2^2 Y X_2
  100100010
```

Now lets see how the high school algorithm for multiplying in binary implements this. First the bits of y are written on the top line and the bits of x, namely $x_4x_3x_2x_1x_0$, are written

below it. On the i^{th} line below this, we multiply x_i by y and shift this over i spots. This gives $2^i y x_i$ on this i^{th} line. The algorithm then adds these lines up giving the answer $\sum_i 2^i y x_i$ as required.

Now lets consider our algorithm. Lets first see how it determines the bits of x. The variable ℓ starts at x and comes down by a factor of two rounded down each iteration and hence after i iterations is $\ell_i = \lfloor \frac{x}{2^i} \rfloor$. Similarly $r_i = 2^i y$. An iteration adds in r_i to s iff ℓ_i is odd. $\ell_i = \lfloor \frac{x}{2^i} \rfloor$ is odd iff the i^{th} bit of x is $x_i = 1$. For example, $x = 13$ in binary is $x = 1101_2$. Here, $x_0 = 1$ because $\ell_0 = x = 13$ is odd and $x_1 = 0$ because $\ell_1 = \lfloor \frac{x}{2} \rfloor = 6$ is even. Hence, the bit x_i indicates whether or not r_i is added to s. Hence, our answer, which is the final value of s, is $\sum_i r_i x_i = \sum_i 2^i y x_i$ as required.

11. (Answer in slides) Multiplying using Adding: My son and I wrote a JAVA compiler for his grade 10 project. We needed the following. Suppose you want to multiply $X \times Y$ two n bit positive integers X and Y. The challenge is that the only operations you have are adding, subtracting, and tests like $\leq$. The standard high school algorithm seems to require looking at the bits of X and Y and hence can't obviously be implemented here. The Ethiopian algorithm requires dividing by two, so can't be implemented either. A few years after developing this algorithm, I noticed that it is in fact identical to this high school multiplication algorithm, but I don't want to tell you this because I don't want you thinking about the algorithm as a whole. This will only frighten you. All you need to do is to take it one step at a time. I want only want you to establish and maintain the loop invariant and use it to get the post condition. (The flavor is very similar to what was done in the Ethiopian problem on the practice test.) To help, I will give you the loop invariants, the measure of progress, and the exit condition. I also want you to explain what the time complexity of this algorithm is, i.e. the number of iterations and the total number of bit operations as a function of the size of the input.

We have values i, x, a, $u[]$, and $v[]$ such that

Useful Arrays: Before the main loop, I will set up two arrays for you with the following values. Then they will not be changed.

LI0': $u[j] = 2^j$, (for all j until $u[j] > X$)

LI0": $v[j] = u[j] \times Y$. Note $v[j] - u[j] \times Y = 0$

My Code: For completion, I include my code, but it is not necessary for you to understand it.

$u[0] = 1$
$v[0] = Y$
$j = 0$
while($u[j] \leq X$)
    $u[j+1] = u[j] + u[j]$
    $v[j+1] = v[j] + v[j]$
    $j = j + 1$
end while

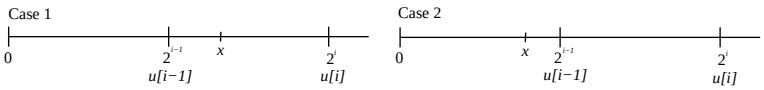
For Main Loop: These are the loop invariants for you to deal with.

LI0: Neither X nor Y change.

LI1: $X \times Y = x \times Y + a$ (i.e. Shrinking Instance)

LI2: $x \geq 0$

LI3: $x < 2^i = u[i]$ (below are two cases)



Measure of progress: i. It decreases by 1 each iteration.

Exit Condition: $i = 0$

Use the steps laid out in class to complete the description of the algorithm.

- Answer:

  Establishing the Loop Invariant $\langle pre{-}cond\rangle$ & $code_{pre\text{–}loop}$ $\Rightarrow$ $\langle loop{-}invariant\rangle$:

  $code_{pre\text{–}loop}$:

  $i = j$
  $x = X$
  $a = 0$

  LI1: $x \times Y + a = X \times Y + 0 = X \times Y$.

  LI2: $x = X \geq 0$ by pre-condition

  LI3: $x = X < u[j] = u[i]$ by exit condition of while loop that constructs $u[j]$.

  Steps:

  loop
  $\langle loop{-}invariant\rangle$
  exit when $i = 0$
  $i = i - 1$;
  if$(u[i] \leq x)$
  $x = x - u[i]$
  $a = a + v[i]$
  end if
  end loop

  Maintaining the Loop Invariant $\langle loop{-}invariant'\rangle$ & *not* $\langle exit{-}cond\rangle$ & $code_{loop}$ $\Rightarrow$ $\langle loop{-}invariant''\rangle$:

  Let x' be the value of x when we are at the top of the loop and let x'' be that after going around the loop one more time. (Same for i and a)

  There are two cases:

  if$(u[i''] \leq x')$: The code in the step gives $i'' = i' - 1$, $x'' = x' - u[i'']$, and $a'' = a' + v[i'']$.

  Maintain LI1: $x'' \times Y + a'' = (x' - u[i'']) \times Y + (a' + v[i'']) = x' \times Y - u[i''] \times Y + a' + v[i''] = (x' \times Y + a') + (v[i''] - u[i''] \times Y) = (X' \times Y$ by LI1$) + (0$ by LI0"$) = X \times Y$.

  Maintain LI2: By the if statement $u[i''] \leq x'$ and hence $x'' = x' - u[i''] \geq 0$.

  Maintain LI3: $x'' = x' - u[i''] = x' - 2^{i'-1} < 2^{i'} - 2^{i'-1}$ (by LI3) $= 2^{i'-1} = 2^{i''} = u[i'']$.

  else The code in the step gives $i'' = i' - 1$, $x'' = x'$, and $a'' = a'$

  Maintain LI1&2: Trivial

  Maintain LI3: By the else of the if statement, $u[i''] > x'$.

  Obtaining the postcondition $\langle loop{-}invariant\rangle$ & $\langle exit{-}cond\rangle$ & $code_{post\text{–}loop}$ $\Rightarrow$ $\langle post{-}cond\rangle$:

  By exit condition, $i = 0$.
  By LI2&3, $0 \leq x < u[i] = 1$, and hence $x = 0$.
  By LI1, $X \times Y = x \times Y + a = 0 \times Y + a = a$.
  If follows that the postcondition is met simply by outputting a.

  Time Complexity: The "size" of the input is $n = \log_2 X + \log_2 Y$, which is the number of bits to write down X and Y. The number of iterations is the initial value of i, which is set so that $X = x < u[i] = 2^i$, giving $i = \lceil \log_2 X \rceil \leq \mathcal{O}(n)$. Each iteration requires add/subtract operations which take $\mathcal{O}(n)$ bit operations. Hence this algorithm will require $\mathcal{O}(n^2)$ bit operations.

  A Comparison: Though this algorithm seems strange, it is in fact exactly the same as the high school algorithm for multiplying in binary (and hence the same as the Ethiopian algorithm). Lets us first understand multiplication $X \times Y$ in terms of bits. Let $X_4X_3X_2X_1X_0$ denote the bits of X written in binary. For example $X = 58 = 11001_2$. The bit X_i is shifted into place by multiplying it by 2^i. This gives $X = \sum_i 2^i X_i$. Plugging this in gives that $X \times Y = \left[\sum_i 2^i X_i\right] \times Y = \sum_i \left[2^i X_i Y\right]$.

```
Y   111010
X      101
     111010     2^0 Y X_0
    000000      2^1 Y X_1
   111010       2^2 Y X_2
  100100010
```

Now lets see how the high school algorithm for multiplying in binary implements this. First the bits of y are written on the top line and the bits of X, namely $X_4X_3X_2X_1X_0$, are written below it. On the i^{th} line below this, we multiply X_i by Y and shift this over i spots. This gives 2^iYX_i on this i^{th} line. The algorithm then adds these lines up giving the answer $\sum_i 2^iYX_i$ as required.

Now lets consider our algorithm. Lets first see how it determines the bits of X. Let x_i denote the value of x in the iteration in which i is i. Our first value of i is the number of bits in X, here 5, because it is the smallest for which $X = x < u[i] = 2^i = 100000_2$. A loop invariant that we will maintain is that x_i will always be the integer formed from the right most i bits $X_{i-1} \ldots X_2X_1X_0$ of $X = 11001_2$, namely $x_5 = 11001_2$, $x_4 = 1001_2$, $x_3 = 001_2$, $x_2 = 01_2$, $x_1 = 1_2$, and $x_0 = 0_2$. We establish this loop invariant by setting $x_5 = x_i = X$. We set $u[i-1] = 10000_2$ so when written in binary it is zero everywhere except a 1 in the location of the bit X_{i-1}. The iteration tests whether $u[i-1] \leq x_i$ which tells us whether or not $X_{i-1} = 1$. For example, we know that $X_4 = 1$ because $u[4] = 10000 \leq 11001_2 = x_4$ we know that $X_2 = 0$ because $u[2] = 100 > 001_2 = x_3$. We maintain this loop invariant about x_i being the right i bits of X by removing its left most bit using $x_{i-1} = x_i - u[i-1]$. For example, $x_4 = x_5 - u[4] = 11001_2 - 10000_2 = 1001_2$ and $x_2 = x_3 = 001_2 = 01_2$. Each iteration also adds $v[i-1]$ into a iff $u[i-1] \leq x_i$ and hence iff $X_{i-1} = 1$. Hence, our answer, which is the final value of a, is $\sum_i v[i]X_i = \sum_i 2^iYX_i$ as required.