assignment1

Bertram Lee

February 2022

Question 1

1. 3 4 5 9
2. 4 3 1 2
3. $T(n) = 5T(\frac{n}{2}) + c$. Since $c = O(n^{\log_2 5 - \epsilon})$ for $\epsilon = \log_2 5 > 0$, by Master Theorem, $T(n) = \Theta\left(n^{\log_2 5}\right)$
4.

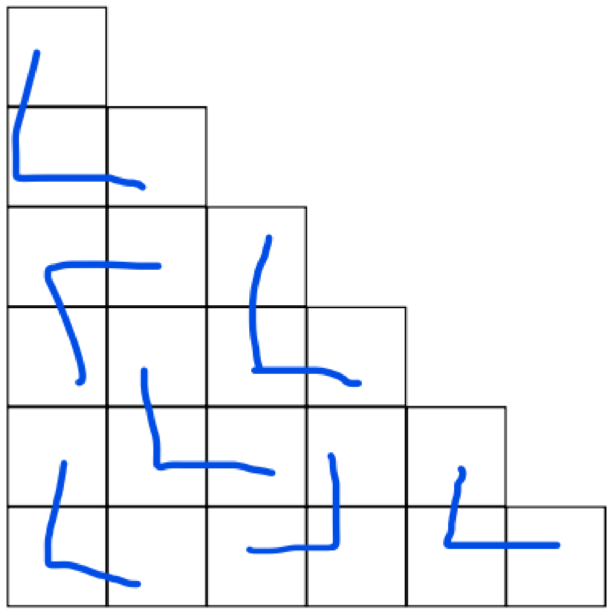
```
procedure SILLY-SORT-FIXED(A, i, j)
    n ← j − i
    if n = 2 then
        if A[i] > A[j − 1] then
            Swap A[i] and A[j − 1]
        end if
    else if n > 2 then
        m ← n/4
        SILLY-SORT-FIXED(A, i, i + 2m)
        SILLY-SORT-FIXED(A, i + m, i + 3m)
        SILLY-SORT-FIXED(A, i + 2m, j)
        SILLY-SORT-FIXED(A, i, i + 2m)
        SILLY-SORT-FIXED(A, i + m, i + 3m)
        SILLY-SORT-FIXED(A, i, i + 2m)
    end if
end procedure
```

5. $T(n) = 6T(\frac{n}{2}) + c$. Since $c = O(n^{\log_2 6 - \epsilon})$ for $\epsilon = \log_2 6 > 0$, by Master Theorem, $T(n) = \Theta\left(n^{\log_2 6}\right)$

Question 2

1. Total number of tiles in $T_n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$. For $n = 3k + 1$, where $k \geq 1$, Total number of tiles in $T_n = \frac{(3k+1)(3k+2)}{2} = 3\frac{3k(k+1)}{2} + 1$. $k(k+1)$ is even, so $\frac{k(k+1)}{2}$ is an integer. Hence $T_n = 3\frac{3k(k+1)}{2} + 1$ is not divisible by 3 and cannot be constructed by smaller triangles with size of 3.

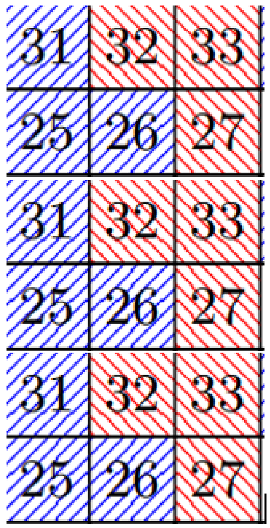
2. graphical solution



3. We shall separate the problem into two cases, when k is even and when k is odd.

   1. k is even:

   From answer above, tiling exist for $k = 2, n = 6$. Assume tiling is possible for T_n, where $n = 3k$, k is even and $k \geq 2$. From T_{3k}, we can construct $T_{3(k+2)}$ by adding a rectangle of width 6 and length $3k$ below T_{3k} and append a T_6 to the right of the rectangle. All we need to do is to prove the rectangle can be constructed using smaller triangles of size 3. By combining 6 smaller triangles, we are able to build a small rectangle of width 6 and length 3(shown below). Since $3k$ is divisible by 3, we can concatenate these smaller rectangles to build rectangle of width 6 and length $3k$. This shows tiling exist for $k + 2$ when k is even. By induction, tiling for $n = 3k$ exist for all even $k \geq 2$.

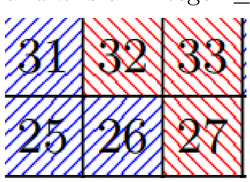


2. k is odd:

   Tiling is possible for $k = 3$, $n = 9$, as shown in question 2. Assume tiling is possible for T_n, where $n = 3k$, k is odd and $k \geq 3$. The method of constructing $T_{3(k+2)}$is exactly the same as the case when k is even - by appending a rectangle of width 6 and length $3k$ and a T_6 to it's right. By induction, tiling for $n = 3k$ exist for all odd $k \geq 3$.

   Combining these two cases, we proved tiling is possible for $n = 3k$, where k is a positive integer and $k \geq 2$

4. We shall show that for every T_n, where $n = 3k$, there exist a T_{3k+2}. The method is similar to above last question, append a rectangle of width 2 and length $3k$ below to T_{3k} , and append a small triangle to the right of the rectangle. A smaller rectangle of width 2 and length 3 can be constructed as shown below. Combine k rectangles to get a rectangle of width 2 and length $3k$. Since we already proved that tiling exist for T_n, where $n = 3k$ and k is an integer ≥ 2, the result follows.

Question 3

1. The graph is acyclic
2. initialize $visited[v] = false$ for all $v = 0$ to $n - 1$

```
2: procedure NUMBER-OF-PATHS(n)
3:     nPaths ← 0
4:     if n is empty then
5:         return nPaths
6:     else
7:         nNode ← 1
8:         create queue Q
9:         push n to Q
10:        while Q is not empty do
11:            v ← dequeue(Q)
12:            visited[v] ← true                    ▷ mark vertex v as visited
13:            nNode ← nNode + 1
14:            for all unvisited and adjacent node of v do
15:                u ← unvisited and adjacent node of v
16:                if edge(v,u) > D then
17:                    nPaths ← nPaths + number-of-paths(u)
18:                else
19:                    enqueue(Q, u)
20:                end if
21:            end for
22:        end while
23:        return nPaths + nNode(nNode−1)/2
24:    end if
25: end procedure
```

The algorithm is based on the fact that for a connected acyclic graph, the number of paths is equal $\binom{k}{2}$, where k is the number of nodes in the connected acyclic graph. The algorithm finds the number of connected vertices by breadth first traversal, implemented by a queue. $nNode$ is the number of node that can be traversed from node n (traversable here means the only path from n to the other node does not contain an edge with distance > D). Nodes that are not traversable from node n are passed as argument recursively as shown in line 16, which returns the number of paths in that seperate graph (seperat e graph as in two graphs are seperated by a edge length $> D$). The function returns $nPaths + \frac{nNode(nNode-1)}{2}$, where $nPaths$ is the number of paths found recursively in other seperate graphs, and $\frac{nNode(nNode-1)}{2}$ is $\binom{nNode}{2}$, which is the number of paths in the local graph.

3. Assume the graph uses a adjacency matrix representation. The algorithm runs in $O(n^2)$, since every node will be visited and pushed to the queue exactly once, and for each node visited, it requires to compare n times to find which nodes are adjacent and which node isn't, just like breadth first traversal in adjacency matrix graph representation. calculating number of paths is $O(1)$ for each separate graph.

Question 4

1. We shall make three functions, find-center, find-edge-node and find-node-contain-price, and a global boolean array $asked[]$ which stores whether we queried the vertex or not.

   GLOBAL VARIABLES
   initialize $asked[v] = false$ ▷ stores had the vertex been queried
   initialize $visited[v]$ ▷ stores the previously travelled node, allow back tracking
   $nNode \leftarrow 0$ ▷ number of node in graph
   $pathLength \leftarrow 0$ ▷ length of the diameter of acyclic graph
   GLOBAL VARIABLES END
   procedure FIND-NODE-CONTAIN-PRICE(v) ▷ v is any node of graph
       $c \leftarrow$ FIND-CENTER(v)
       if $nNode = 1$ **then**
           return c
       end if
       $(c, w) \leftarrow$ QUERY(c)
       $asked[c] \leftarrow true$
       if query returned Yes **then**
           return c
       else
           return FIND-NODE-CONTAIN-PRICE(w)
       end if
   end procedure

   procedure FIND-CENTER(v)
       $start \leftarrow$ FIND-EDGE-NODE(v)
       $end \leftarrow$ FIND-EDGE-NODE(start)
       $c = end$
       for $i \leftarrow 1$ to $\frac{pathLength}{2}$ **do**
           $c \leftarrow visited[c]$
       end for
       return c
   end procedure

   procedure FIND-EDGE-NODE(v)
       reinitialize $visited$ array
       $nNode \leftarrow 0$
       $pathLength \leftarrow 0$
       create queue Q
       push v to Q
       $i \leftarrow 0$
       $k \leftarrow 1$
       while Q is not empty **do**
           $v \leftarrow$ pop Q

$nNode \leftarrow nNode + 1$
$k \leftarrow k - 1$
for all vertices adjacent to v and is both not visited and not asked **do**
$u \leftarrow$ vertices adjacent to v and is both not visited and not asked
push u to Q
$visited[u] = v$
$i \leftarrow i + 1$
end for
if $k = 0$ **then**
$pathLength \leftarrow pathLength + 1$
$k \leftarrow i$
$i \leftarrow 0$
end if
end while
return v
end procedure

First		Second		Conclusion
Query	Feedback	Query	Feedback	
c	yes			c
	c			a
	cb			b
	cd	d	yes	d
			de	e
			df	f
	cg	g	gi	i
			gj	j
			gh	h
			Yes	g

ıplate strategy for Question 4.2.

2.

3. use the same algorithm as part 1

4. use the same algorithm as part 1. The principle uses a first breadth first traversal to find the edge of a graph, and from that edge, run another breadth first traversal. The longest path will be the diameter of the graph, and the middle of this path will be the center. We query the center and repeat the process with the appropriate sub-graph until either the node is found or there is only one node in sub-graph. Assume the graph uses a adjacency list representation. The algorithm takes $O(n \log n)$. find-edge-node runs in $O(n + e)$, since it is basically a breadth first traversal and an initialization of a array of size n. But for an acyclic graph, $e < n$, so $O(2n) = O(n)$. find-center preforms two find-edge-node and one for loop which loops $pathLength/2$ times. Combining the cost it runs in $O(n)$. Note that find-node-contain-price is being called recursively, and that find-

center and find-node-contain-price function in subsequent recursive calls does not visit any nodes across nodes that have already been asked, so each calls of find-node-price cuts the graph in half, so the runtime of find-node-contain-price is $T(n) = T(\frac{n}{2}) + O(n)$, where $O(n)$ is the cost of find-center. By Master's Theorem, this is $\Theta(n)$, which is $O(n \log n)$