NOI 2014 – solutions to contest tasks

School of Computing
National University of Singapore

Scientific Committee

22 March 2014

Scientific Committee

Nathan Azaria

Chang Ee-Chien (Chair)

Chin Zhan Xiong

Gan Wei Liang

Jonathan Irvin Gunawan

Steven Halim

Raymond Kang

Ooi Wei Tsang

Frank Stephan

Sung Wing Kin, Ken

Harta Wijaya

Overview

- 4 Tasks. Each has a relatively easy subtask.
- Feedback provided for each test instance.
- Changes from previous years
  - Using CMS
  - Grading Rule: Best submission/tie-break using submission time
- During re-evaluation, due to differences in running time, some submissions obtained higher marks. We pick the consistent runs as the final results. All submissions are considered “tokenized”.

Task 1: ORCHARD

Task 2: SIGHTSEEING

Task 3: CATS

Task 4: OBELISK

Task 1: ORCHARD

Chang Ee-Chien, Frank Stephan, Jonathan Irvin Gunawan

Presented by Frank Stephan

Overview

Alex and Bert inherit an orchard.

The orchard consists of apple trees and banana trees planted in a rectangular array.

Apple trees should go to Alex and banana trees to Bert.

Principle of division:

- Alex and Bert cut out a rectangle from orchard for Bert;
- Every further adjustment afterwards (moving a piece of land with one tree from one person to another) costs SGD 1 lawyer fees.

Example

```
1 1 0 1 0        B B A A A        B B A B A
1 1 1 0 0        B B A A A        B B B A A
0 1 0 0 0  =>    B B A A A  =>    A B A A A
1 1 0 0 0        B B A A A        B B A A A
1 0 0 0 0        A A A A A        B A A A A
```

Original Orchard

Rectangle to Bert

1 tree to Alex
3 trees to Bert

Overall costs: SGD 4

Algorithm

```
alpha[i,j] is 0 (apple) or 1 (banana) where
i goes from 0 to n-1 and j from 0 to m-1;
beta[0,j] = 0;
beta[i+1,j] = alpha[0,j]+...+alpha[i,j];
h = number of banana trees; costs = h-1;
for intervals {imin,...,imax} of i-indices do
  { current = h;
    for j = 0,1,...,m-1 do
      { k = imax+1-imin +
            2*beta[imin,j]-2*beta[imax+1,j];
        if (h + k < current + k)
        then { current = h + k; }
        else { current = current + k; }
        if (current < costs)
        then { costs = current; } } }
```

Performance of Algorithm

The algorithm has runtime $\mathbf{O(n^2 \cdot m)}$; in the special case that $\mathbf{n = 1}$ the algorithm runs in time linear in $\mathbf{m}$.

An implementation detail is that the two-dimensional arrays are coded in one-dimensional arrays of length $\mathbf{(n+1) \cdot m}$ in order to meet the space requirements.

Easily $\mathbf{O(\min\{m^2 \cdot n, n^2 \cdot m\})}$ possible.
However, as $\mathbf{n \leq m}$ throughout the assessment tasks, this optimisation was not needed.

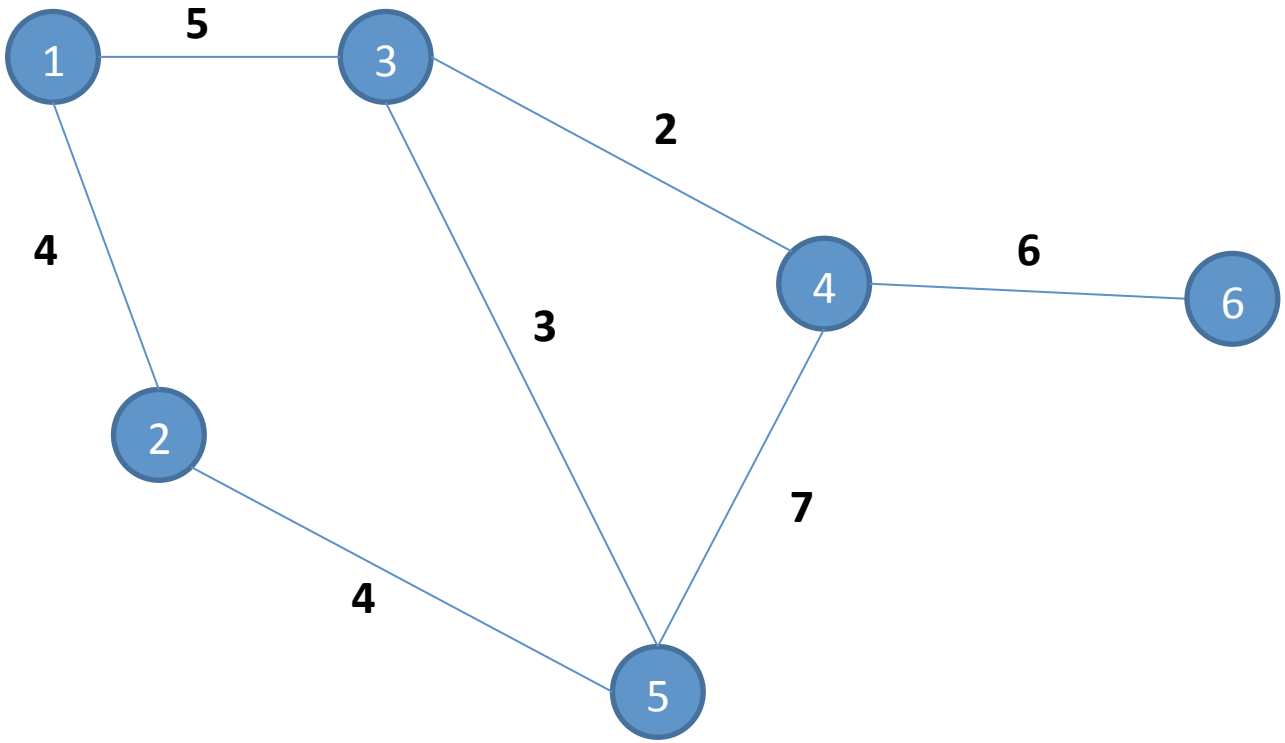
A bit more naive approximation gives $\mathbf{O(n^2 \cdot m^2)}$ by searching for all rectangles and using the additional array $\mathbf{beta}$. This suboptimal algorithm, however, fails to be fast enough on larger orchards.

Task 2: SIGHTSEEING

Nathan Azaria, Harta Wijaya, Ken Sung Wing Kin
Presented by Harta Wijaya

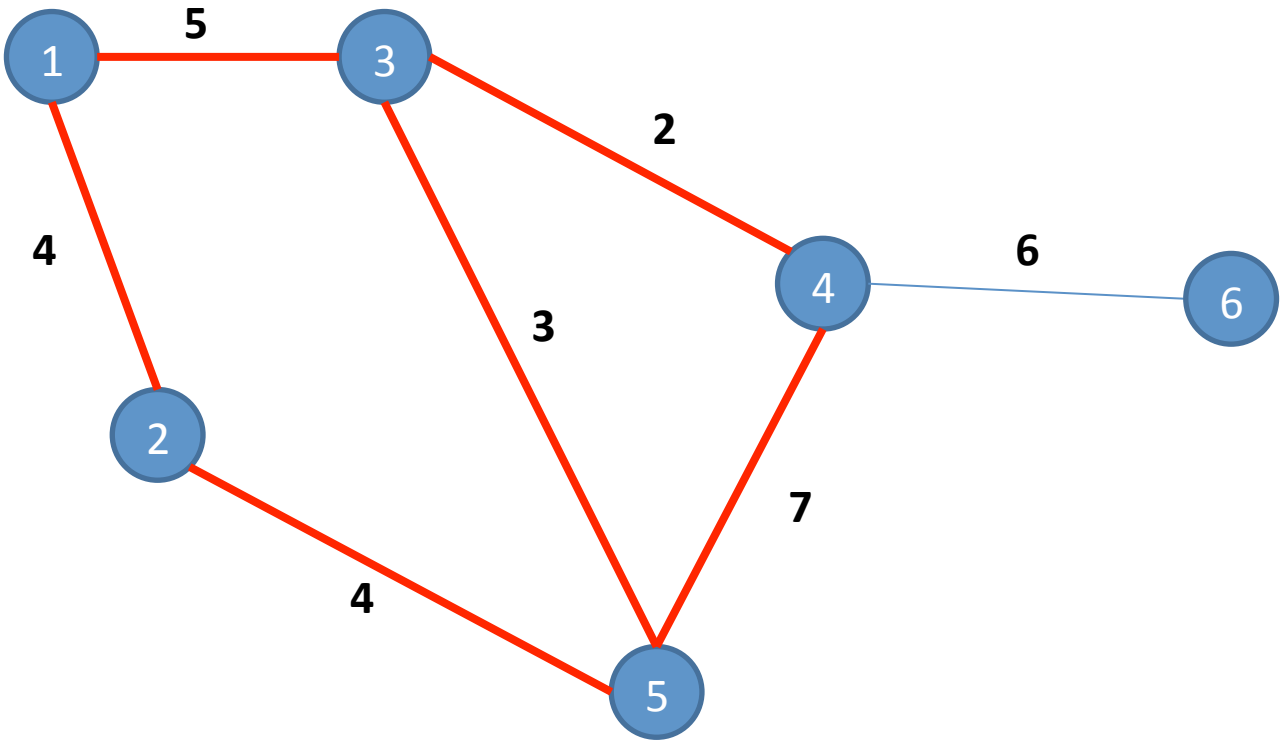
Problem

- Find the best quality from node 1 to node X.
- Example:



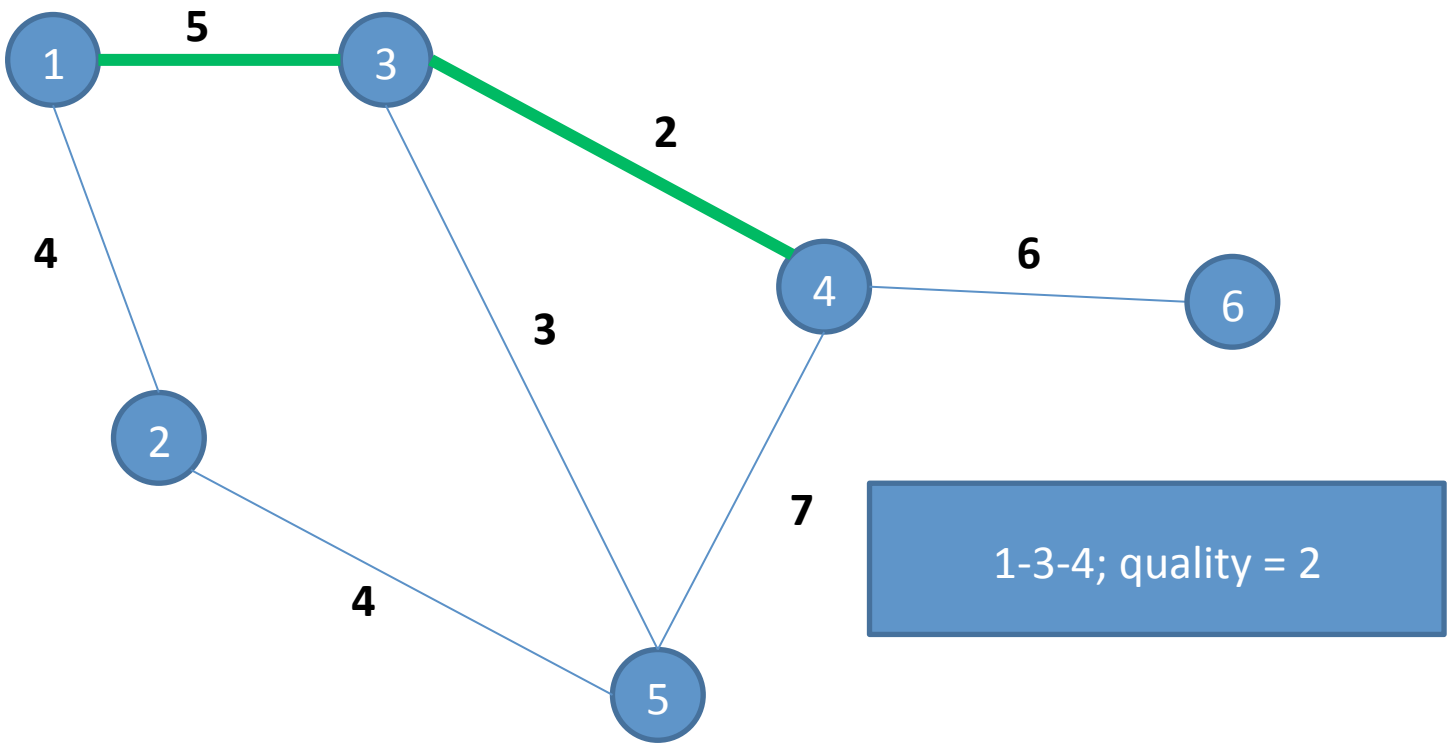
Problem

- The best quality from node 1 to node 4 is 4



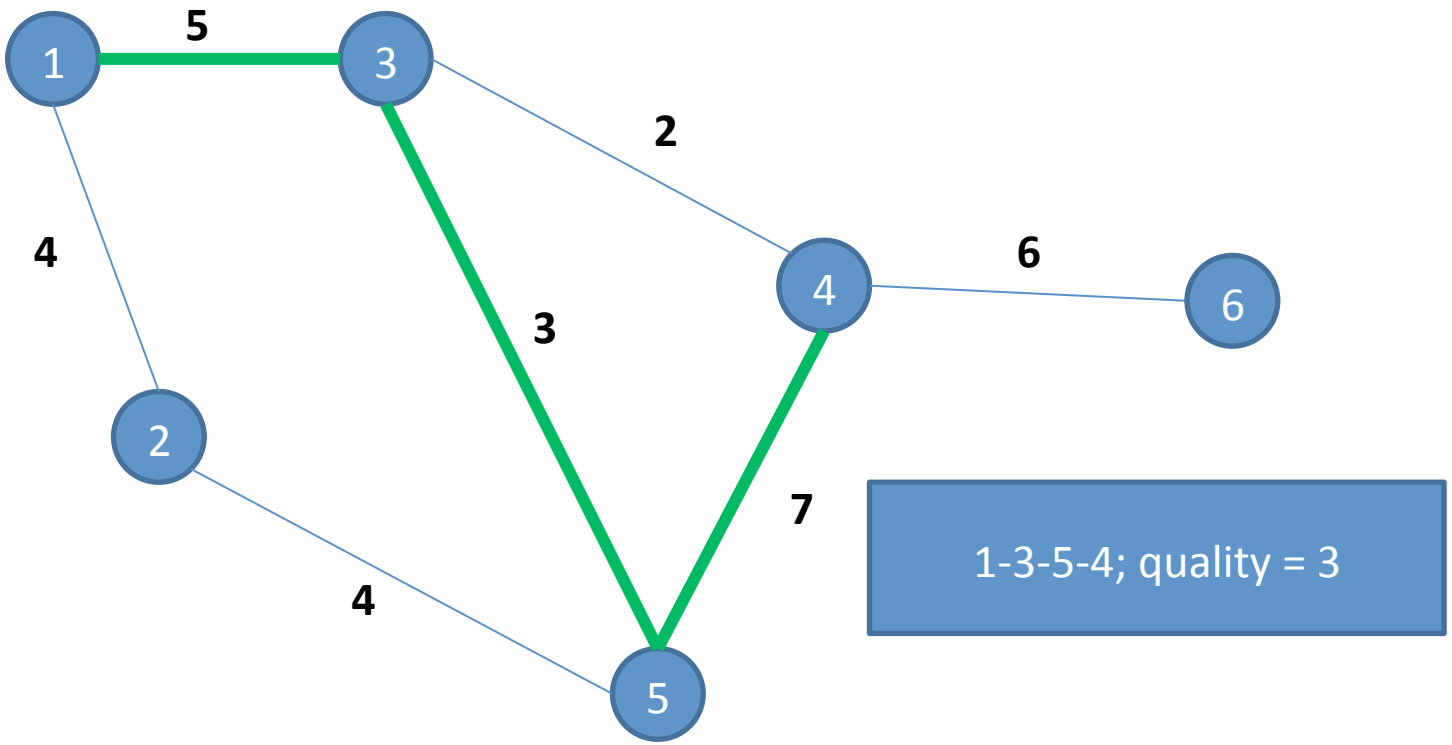
Problem

- The best quality from node 1 to node 4 is 4



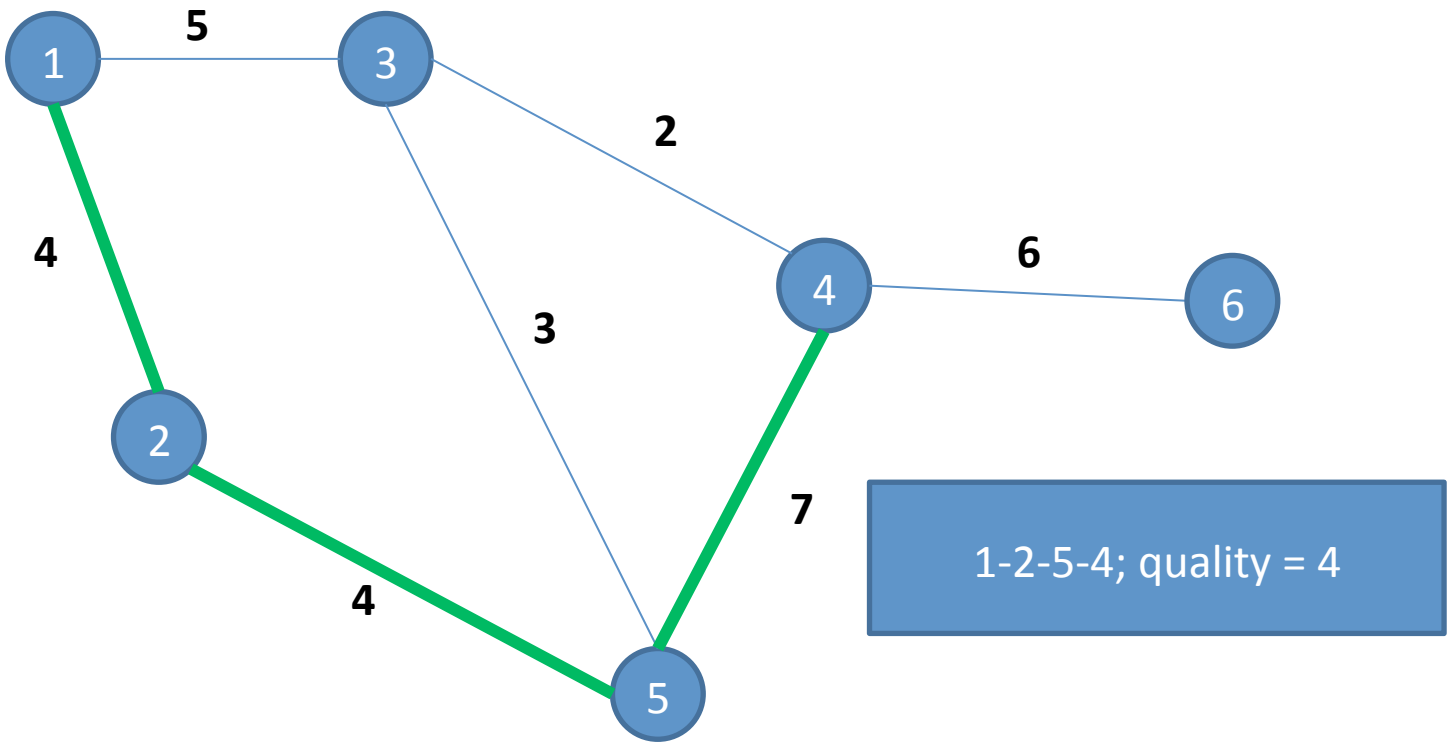
Problem

- The best quality from node 1 to node 4 is 4



Problem

- The best quality from node 1 to node 4 is 4



Subtask 1

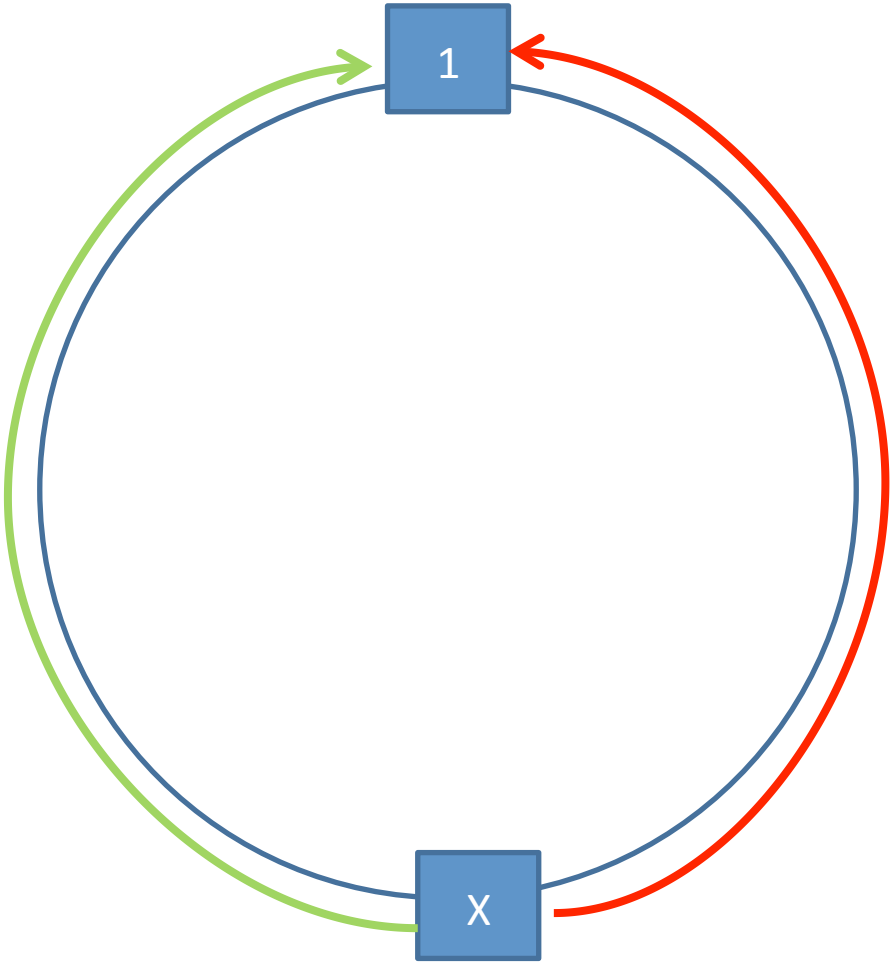
- A simple cycle V <= 100; E <= 100; Q <= 50
- Solution:

  a. There are only 2 possible routes from node 1 to node X

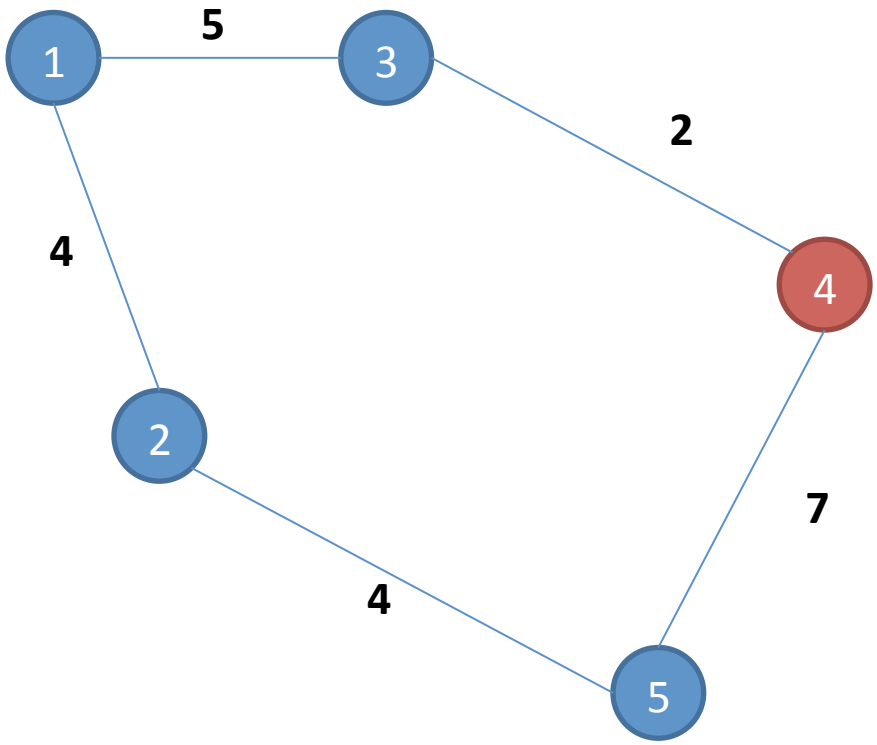
  b. For each query, find the maximum quality between those 2 routes.

- Complexity: O(Q * V)

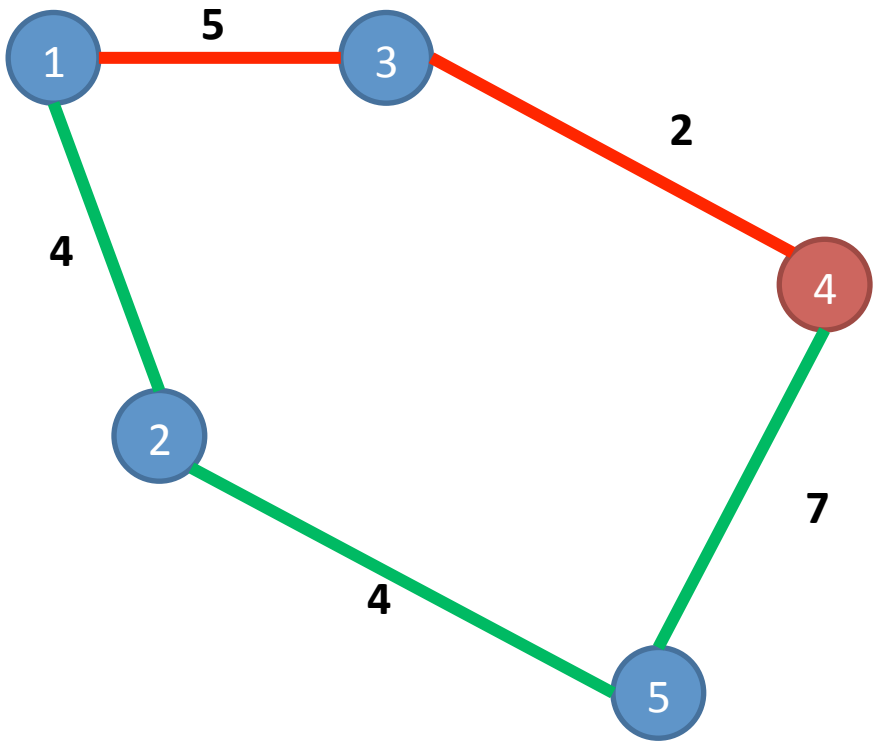
Subtask 1



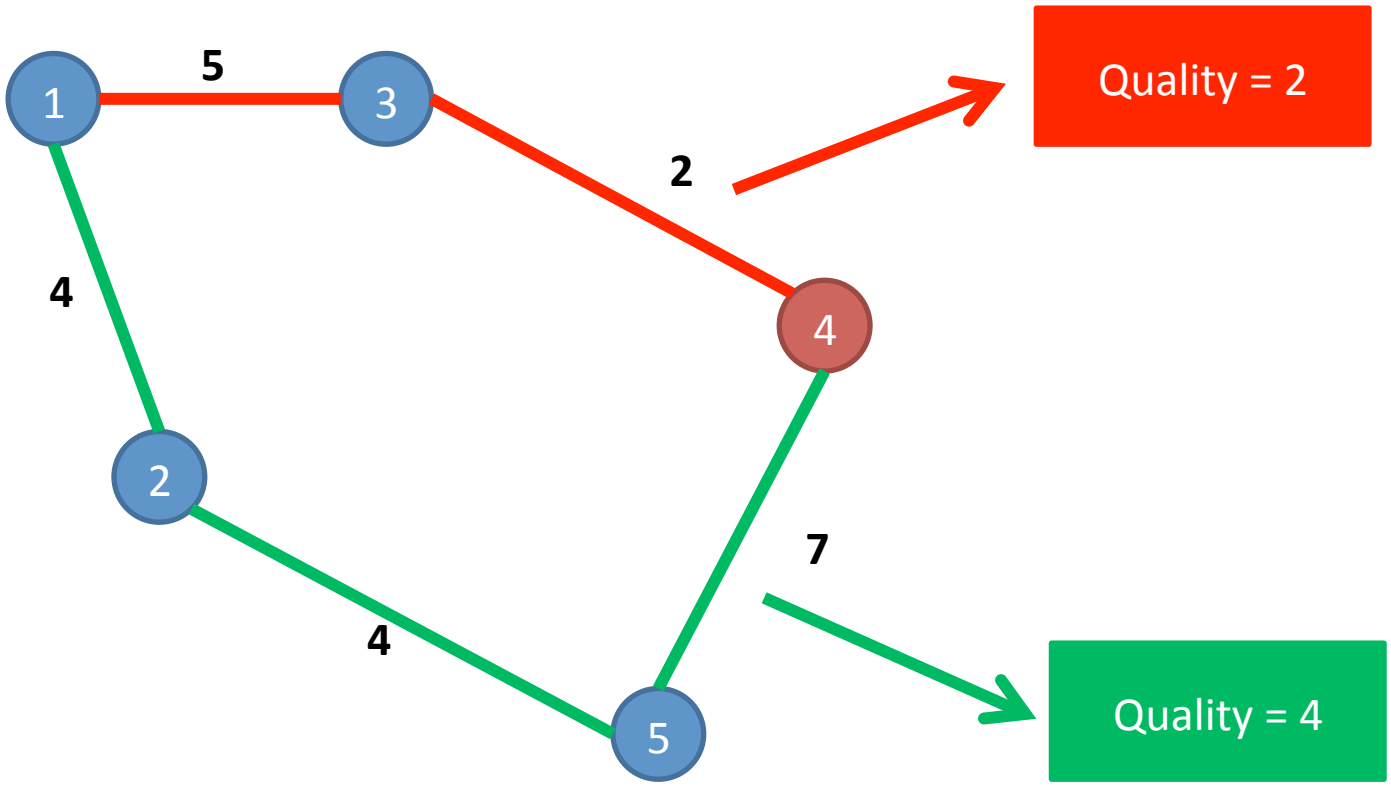
Subtask 1



Subtask 1



Subtask 1



Subtask 2

- General graph, V <= 500; E <= 4,000; Q <= 100
- Solution:

  a. Pre-compute the solution for every tourist sites. This allows us to answer the query in O(1).

  b. Use Floyd Warshall algorithm
- Complexity: O(V^3)

Subtask 2

```
for k = 1; k <= V; k++
    for i = 1; i <= V; i++
        for j = 1; j <= V; j++
            ans[i][j] = max(ans[i][j],
                            min(ans[i][k], ans[k][j]))
```

Subtask 3

- General graph, V <= 50,000; E <= 1,000,000; Q <= 10,000
- Solution:

  a. Pre-compute the solution for every tourist sites. This allows us to answer the query in O(1).

  b. Use Dijkstra algorithm
- Complexity: O(ElogV)

Subtask 4

- General graph, V <= 500,000; E <= 5,000,000; Q <= V - 1
- Solution:

  a. Pre-compute the solution for every tourist sites. This allows us to answer the query in O(1).

  b. Use disjoint set data structure (union by rank and path compression)
- Complexity: O(E α), where α is the inverse of the fast-growing Ackermann function

Subtask 4

- Sort the edges based on quality in descending order (counting sort).
- For each edge e = (from, to, quality),

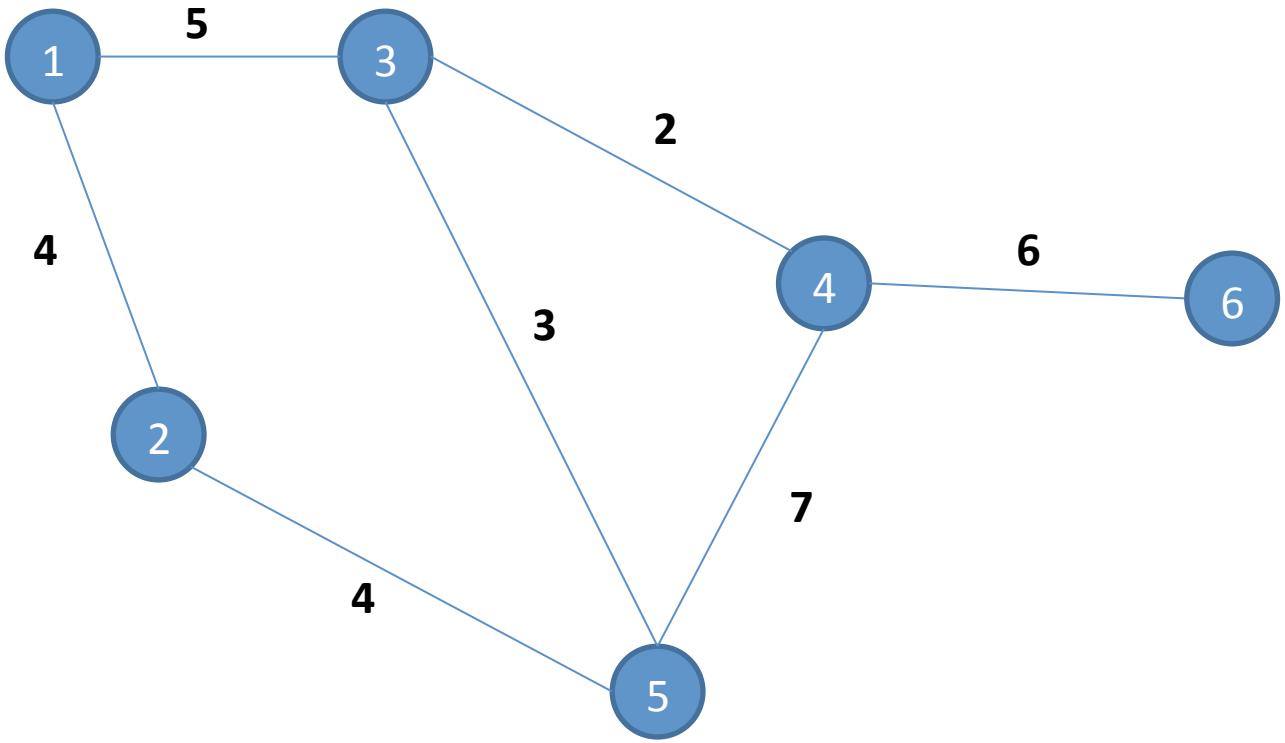
  join(e.from, e.to, e.quality)
- We need to keep track of the nodes connected to the hotel in our disjoint forests to allow us determine the best quality to the hotel.

Subtask 4

- During a join operation, when a tree Tn (not containing node 1) is connected to another tree T (containing node 1), we can determine the quality of all nodes in Tn, i.e. the quality is equal to the edge connecting Tn and T.

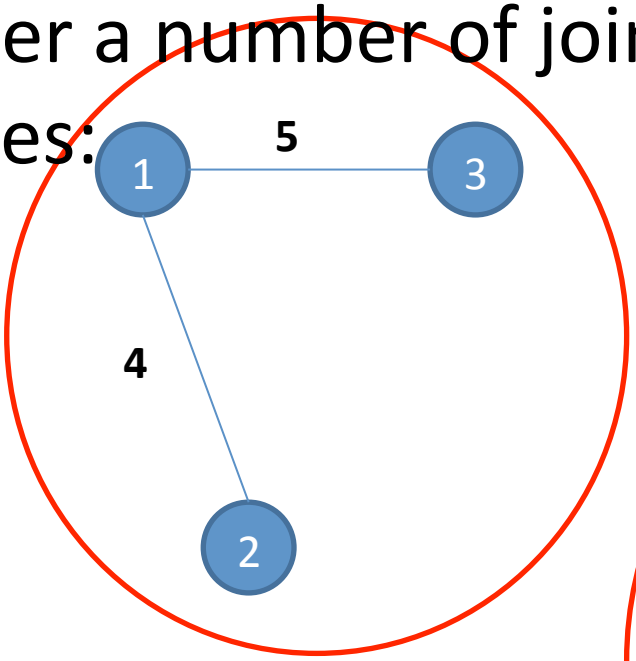
Subtask 4

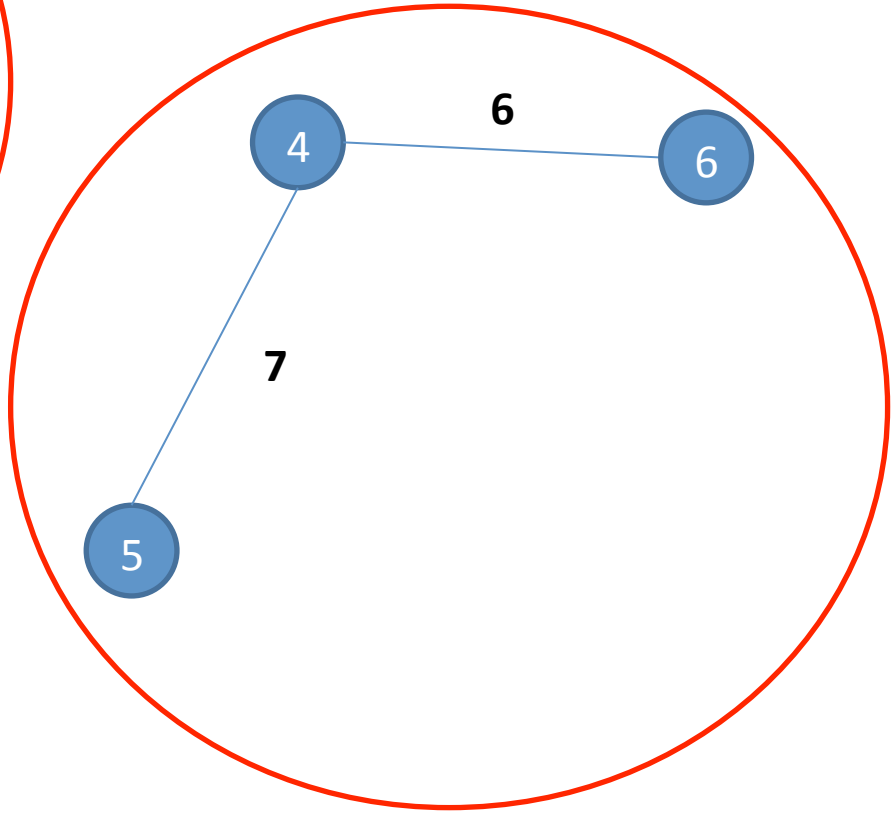
- The initial graph:



Subtask 4

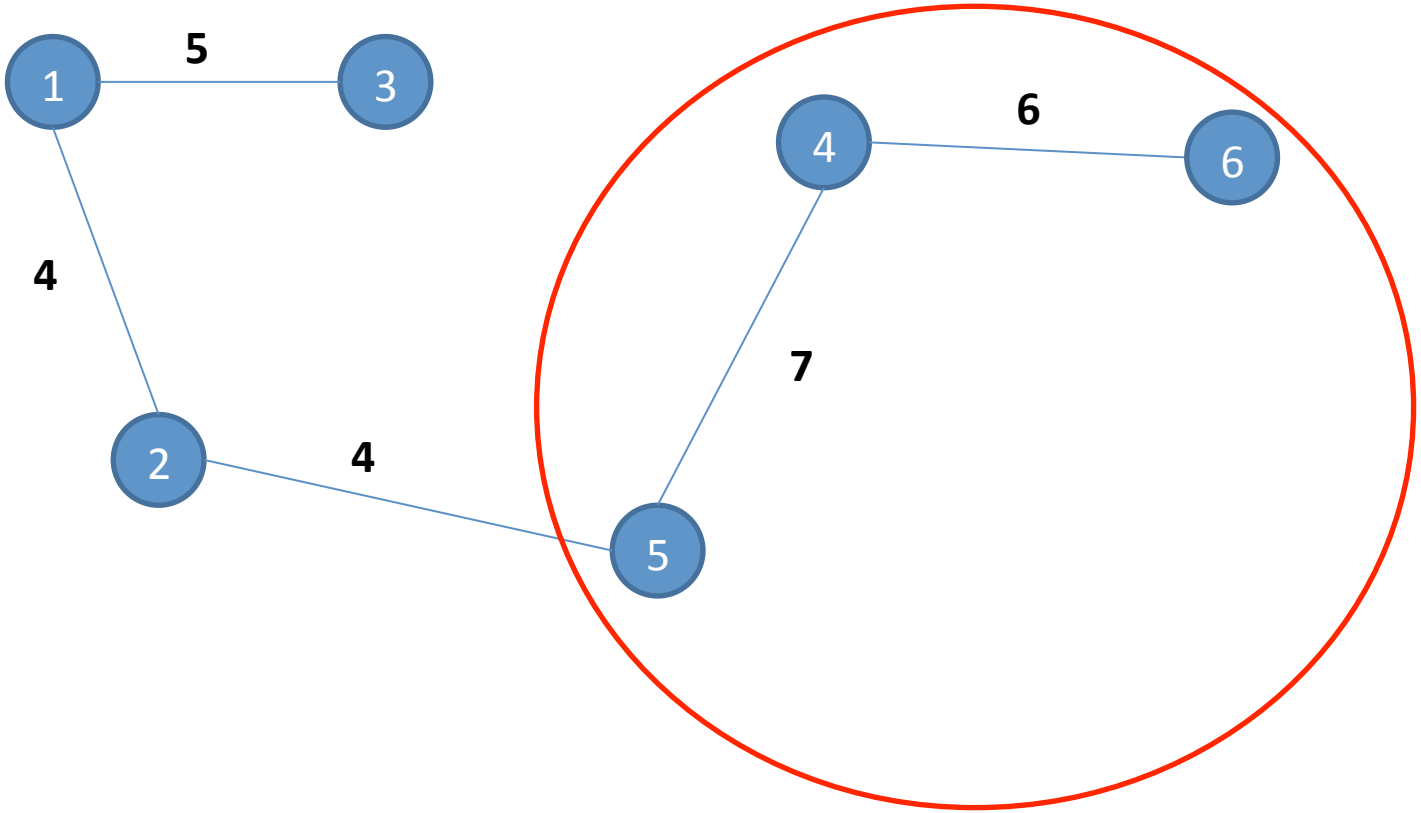
- After a number of join operations, we have 2 trees:





Subtask 4

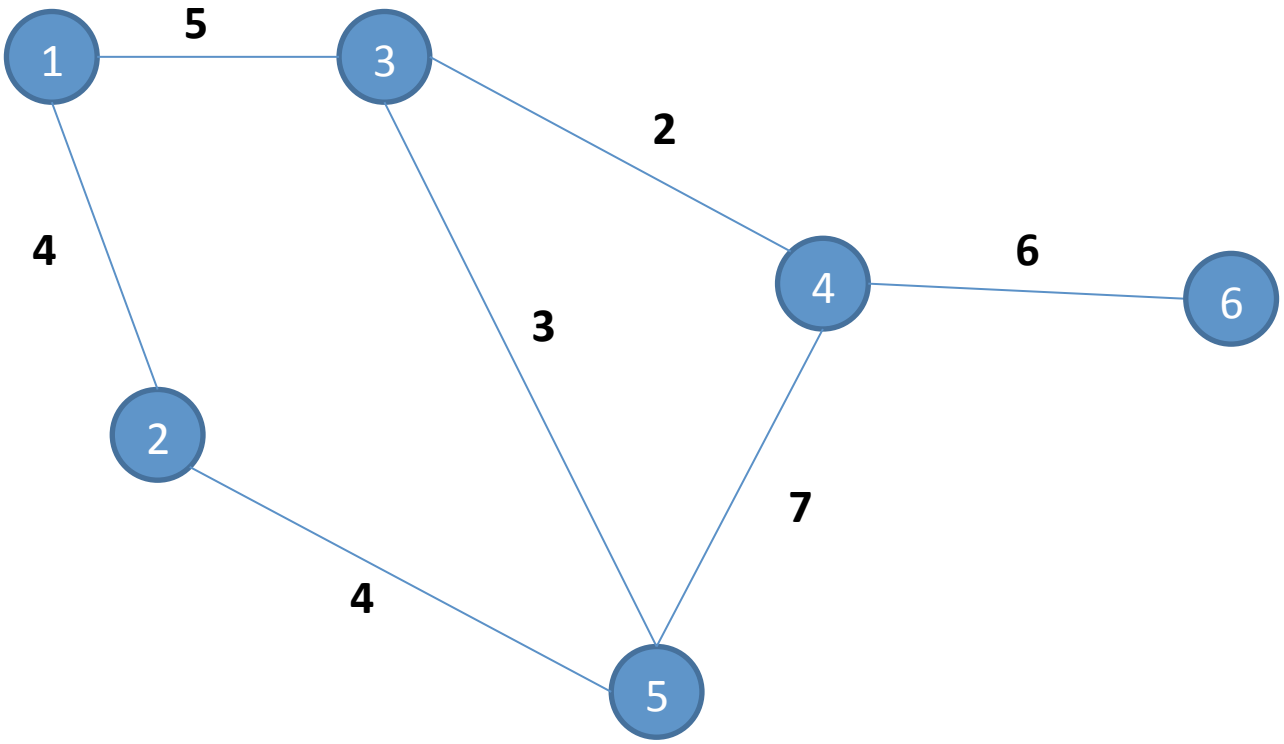
- The next operation is to join (2, 5) with edge.quality = 4.



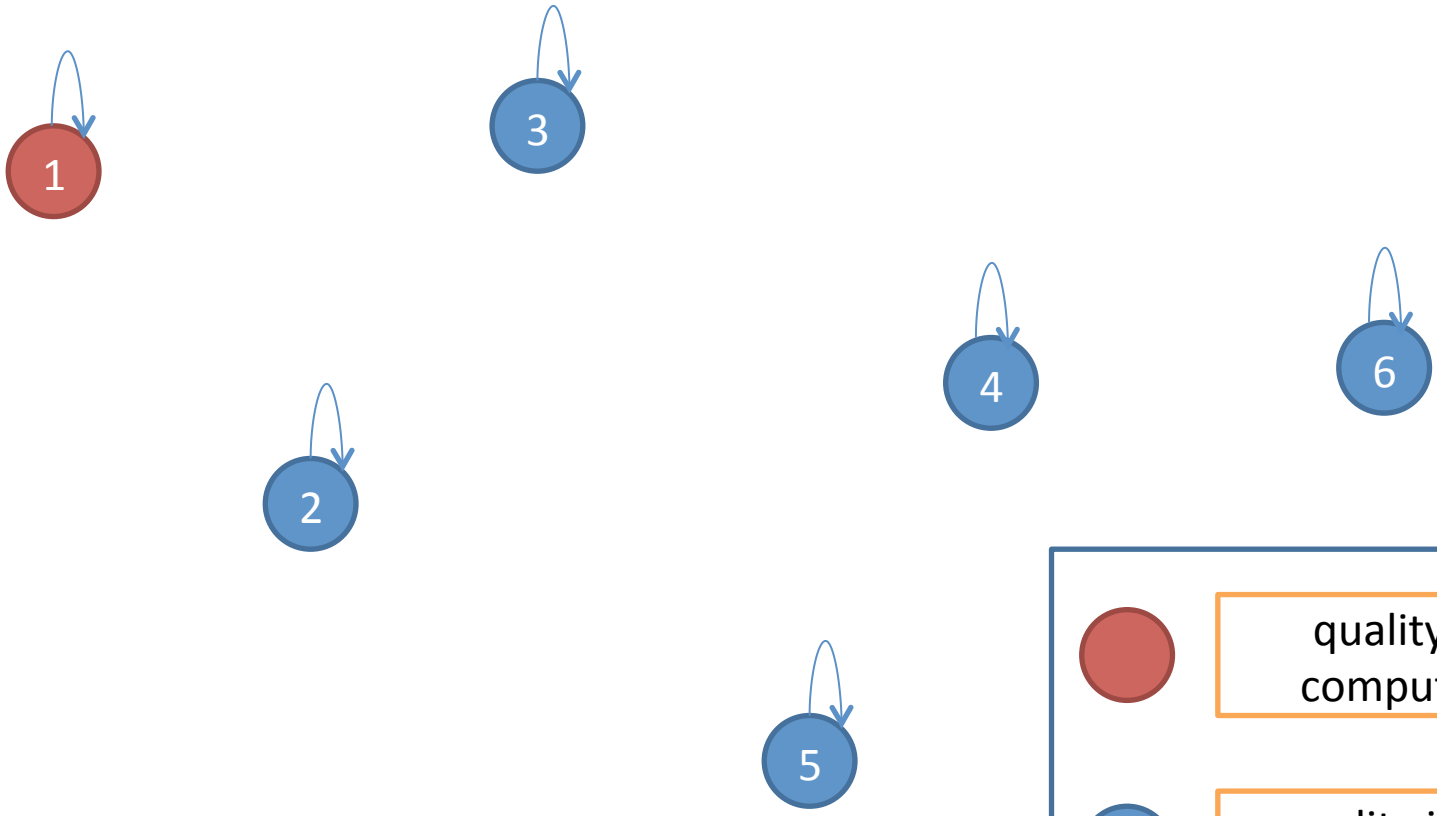
Subtask 4

- After joining that edge (2, 5), the node 4, 5 and 6 will be connected to 1 (able to reach 1) and since we process the edges one-by one from the heaviest edge to the lightest edge, we know that the best quality to reach 4, 5 and 6 is the quality of edge (2, 5) = 4.

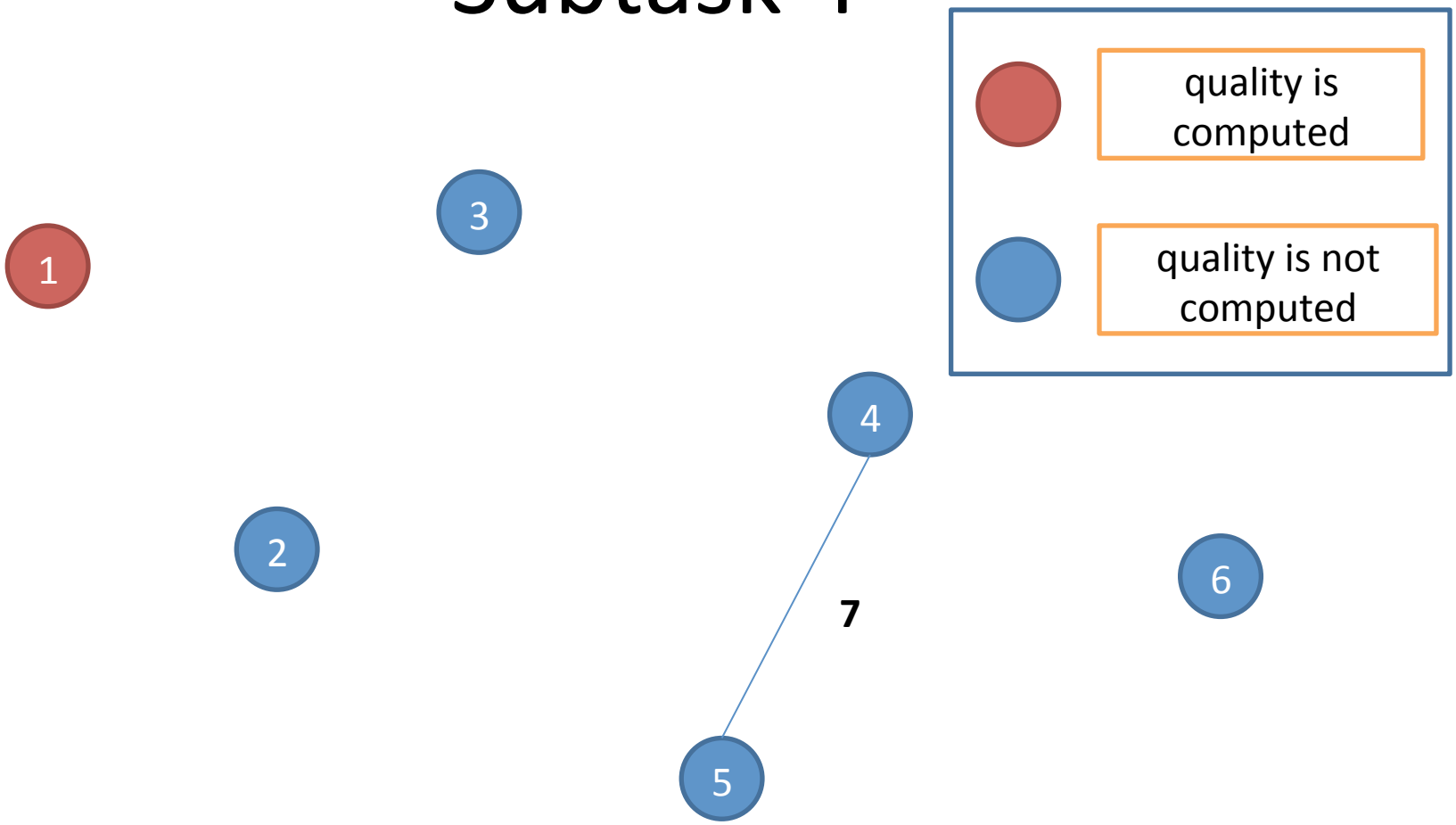
Subtask 4



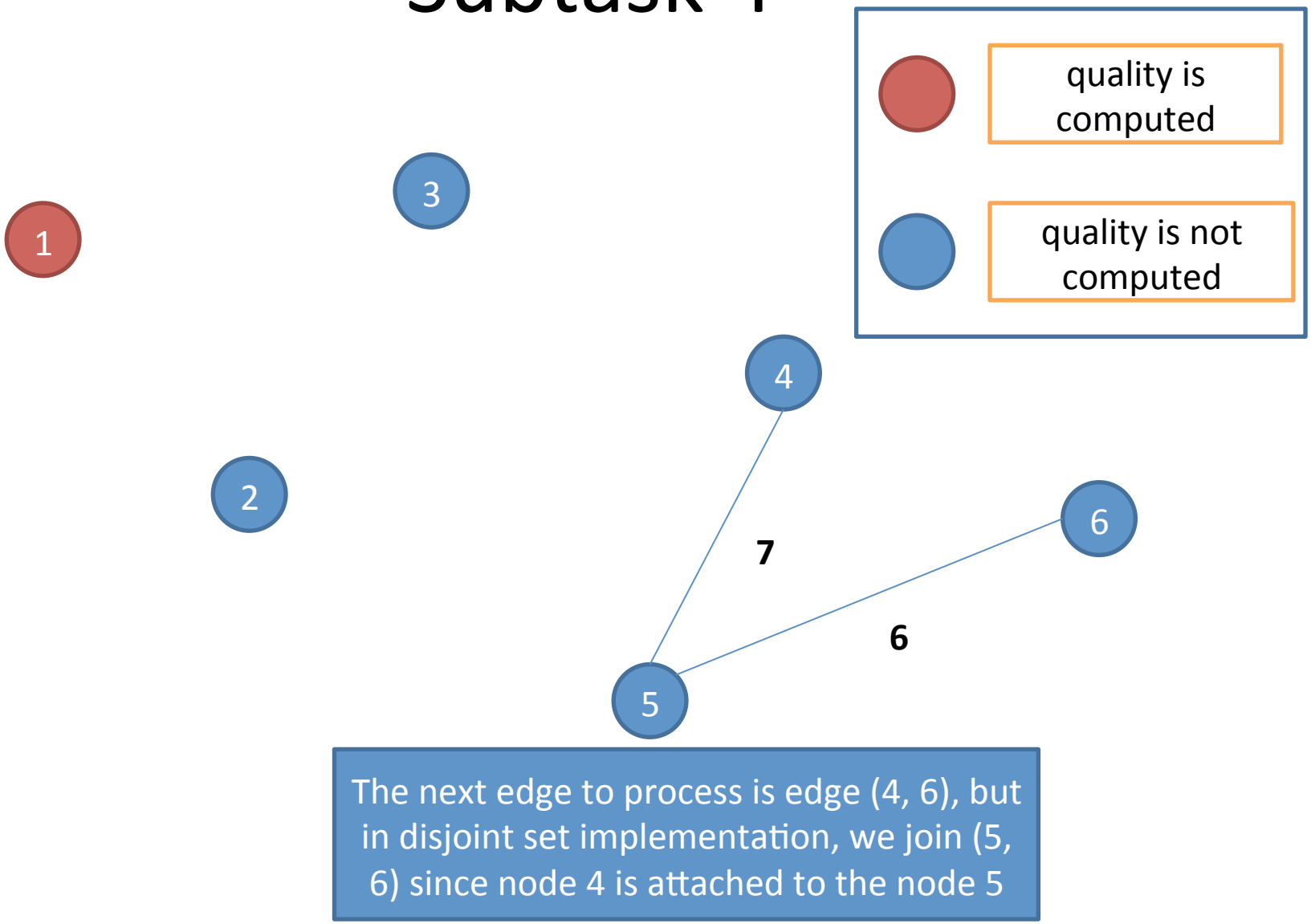
Subtask 4



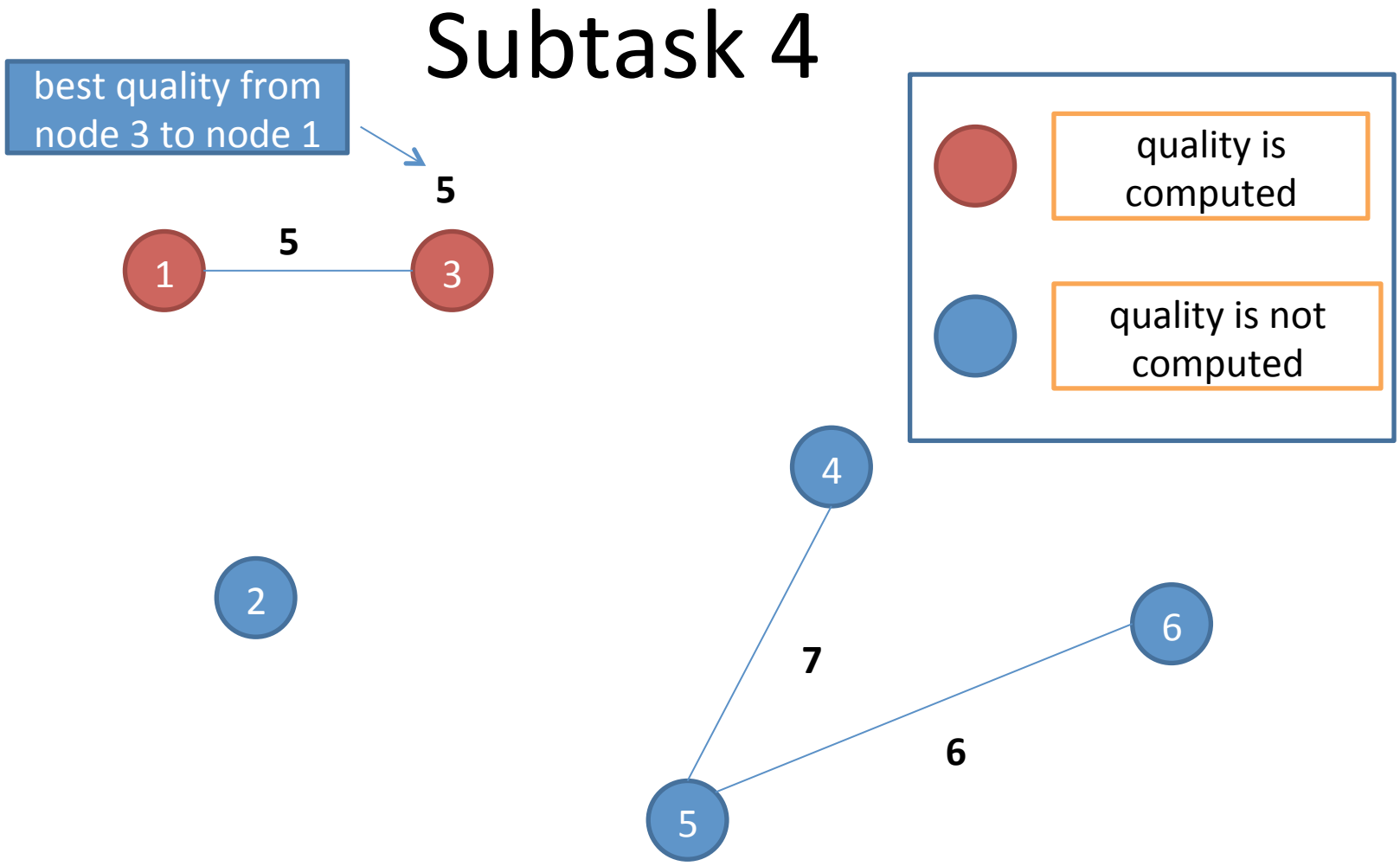
Subtask 4



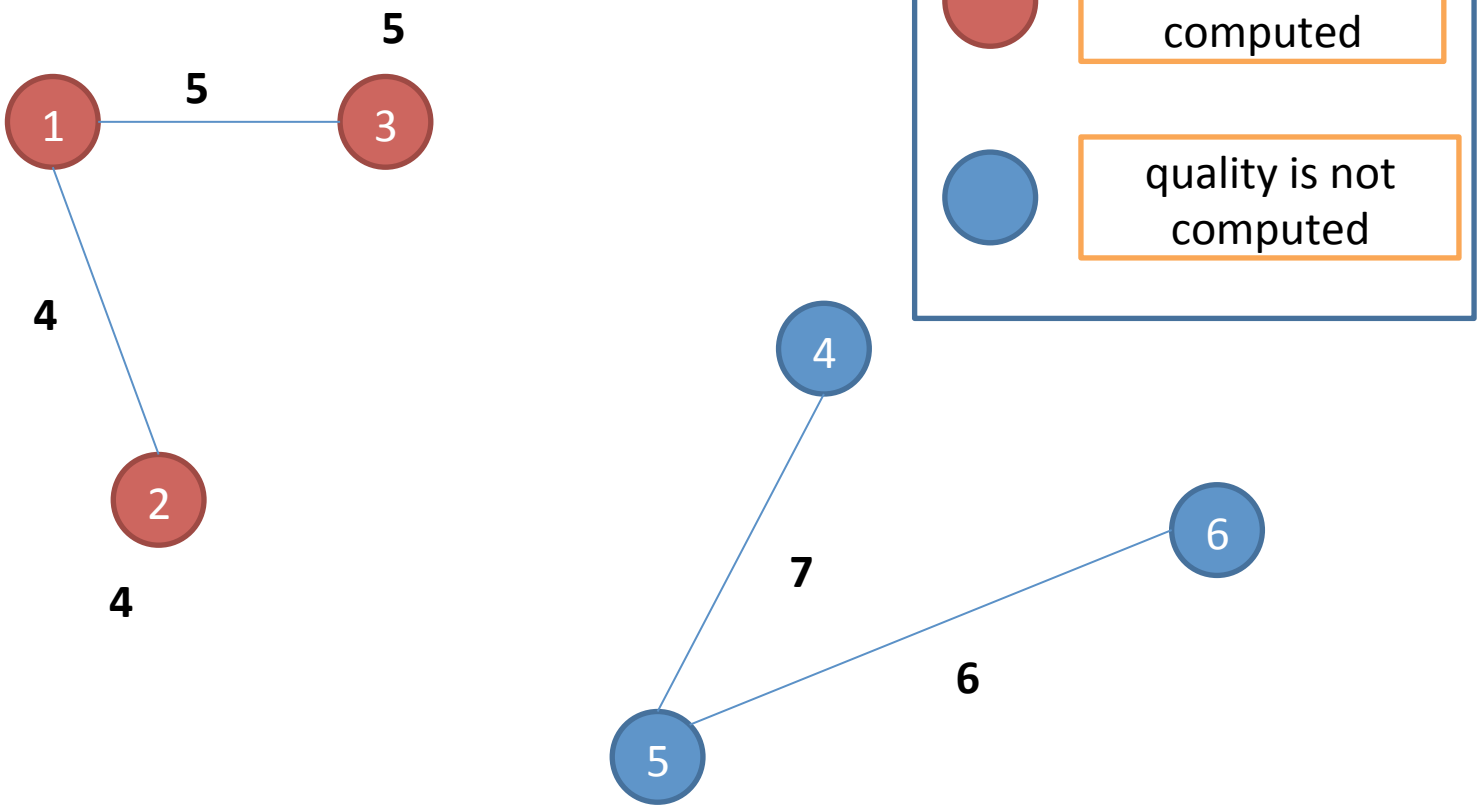
Subtask 4


quality is computed
quality is not computed
1
2
3
4
5
6
7
6
The next edge to process is edge (4, 6), but in disjoint set implementation, we join (5, 6) since node 4 is attached to the node 5

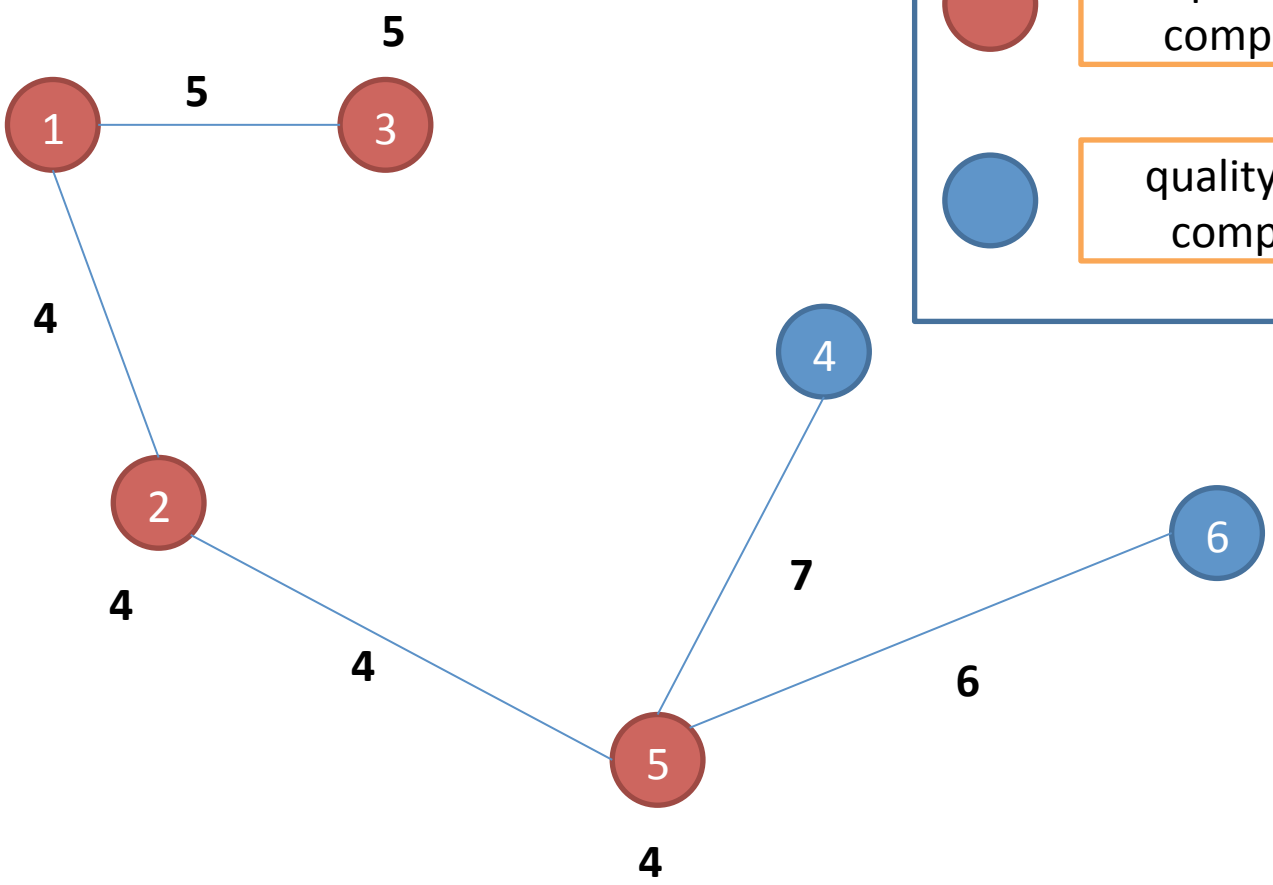
Subtask 4


best quality from node 3 to node 1
5
5
1
3
quality is computed
quality is not computed
4
2
7
6
6
5

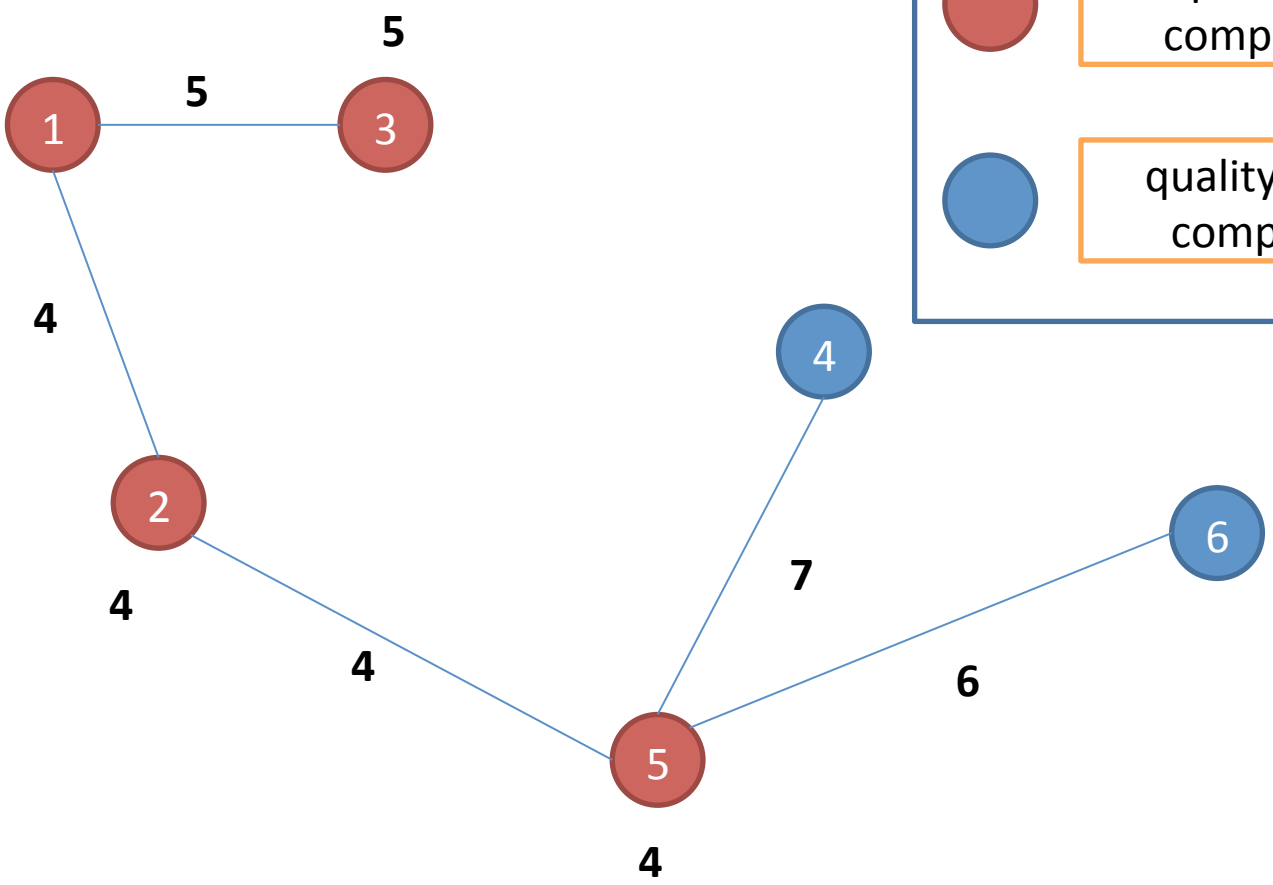
Subtask 4



Subtask 4


quality is computed
quality is not computed
1
2
3
4
5
6
5
5
4
4
4
7
6
4

Subtask 4



During the query, we can get the quality of node 4 and 6 by looking at the quality of the parent in which it is joined to, in this case the parent is node 5, so the quality of node 4 and node 6 is 4.

Subtask 4

- Note that once a group of nodes in a tree Tn is connected to node 1, you SHOULD NOT update the quality for ALL nodes in that group. You only need to update the quality for the parent (update the quality for 1 node only – for efficiency purpose).
- For each join operation between node A and B, you have to keep 1 additional information for node A and B, i.e. the parent of the tree that they are in.

Task 3: CATS

Gan Wei Liang, Steven Halim

Presented by Gan Wei Liang

Problem Statement

- Provided with the pseudo-code of a program
- Problem: Deduce the output of the program when provided with a given input
- Program solves a simple problem using a complicated and slow algorithm

Problem Statement

```
COUNTER = X
WHILE COUNTER > 0
        S2 PUSH T1
        S1 POP
        FLIP LAST BINARY BIT OF ALL NUMBERS IN S1
        IF T2 > L
                COUNTER = COUNTER - 1
                IF COUNTER == 0 PRINT T2
        ELSE
                S2 PUSH N
                S2 PUSH N
                S2 ADD
                S2 ADD
                S1 PUSH T2
                S1 PUSH T2
                S2 POP
                S2 POP
```

Method of Solving

- Non-conventional problem type
- Optimise slow operations
- Deduce the problem that the code is trying to solve
- Solve it as a separate problem

Subtask 1: Simulation

- Code a program that follows the pseudo-code exactly
- Simulate infinite 0’s by pushing sufficiently large number of 0’s onto each stack initially
- Minimal algorithmic knowledge required

Subtask 2: Optimisation

```
COUNTER = X
WHILE COUNTER > 0
        S2 PUSH T1
        S1 POP
        FLIP LAST BINARY BIT OF ALL NUMBERS IN S1
        IF T2 > L
                COUNTER = COUNTER - 1
                IF COUNTER == 0 PRINT T2
        ELSE
                S2 PUSH N
                S2 PUSH N
                S2 ADD
                S2 ADD
                S1 PUSH T2
                S1 PUSH T2
                S2 POP
                S2 POP
```

Adding 2N to top element of S2

Irrelevant as S2 elements not needed again

Subtask 2: Optimisation

```
COUNTER = X
WHILE COUNTER > 0
        S2 PUSH T1
        S1 POP
        FLIP LAST BINARY BIT OF ALL NUMBERS IN S1
        IF T2 > L
                COUNTER = COUNTER - 1
                IF COUNTER == 0 PRINT T2
        ELSE
                ADD 2N TO T2
                S1 PUSH T2
                S1 PUSH T2
```

For an element K pushed onto S1, the number of times its last binary bit is flipped before it is popped is equal to the number of elements pushed onto S1 before K is popped since when each of these elements is popped, the last binary bit of K is flipped

Subtask 2: Optimisation

```
COUNTER = X
WHILE COUNTER > 0
        S2 PUSH T1
        S1 POP
        FLIP LAST BINARY BIT OF ALL NUMBERS IN S1
        IF T2 > L
                COUNTER = COUNTER - 1
                IF COUNTER == 0 PRINT T2
        ELSE
                ADD 2N TO T2
                S1 PUSH T2
                S1 PUSH T2
```

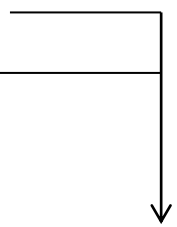
Elements are pushed onto S1 in pairs

Last binary bit is flipped an odd number of times

Last binary bit is flipped an even number of times

Subtask 2: Optimisation

```
COUNTER = X
WHILE COUNTER > 0
        S2 PUSH T1
        S1 POP
        IF T2 > L
                COUNTER = COUNTER - 1
                IF COUNTER == 0 PRINT T2
        ELSE
                ADD 2N TO T2
                S1 PUSH (T2 XOR 1)
                S1 PUSH T2
```



Flipping last binary bit of a number twice leaves it unchanged so value of the element when popped only depends on the parity of the number of times its last binary bit if flipped. Following this pseudo-code is sufficient to solve subtask 2

Subtask 3: Analysis

- With some analysis, the code can be seen as simulating a depth-first search (DFS) according to elements in S1
- Pushing elements onto S1 represents going deeper into the tree
- For clarity, we can convert the code to perform DFS using recursion instead

Subtask 3: Analysis

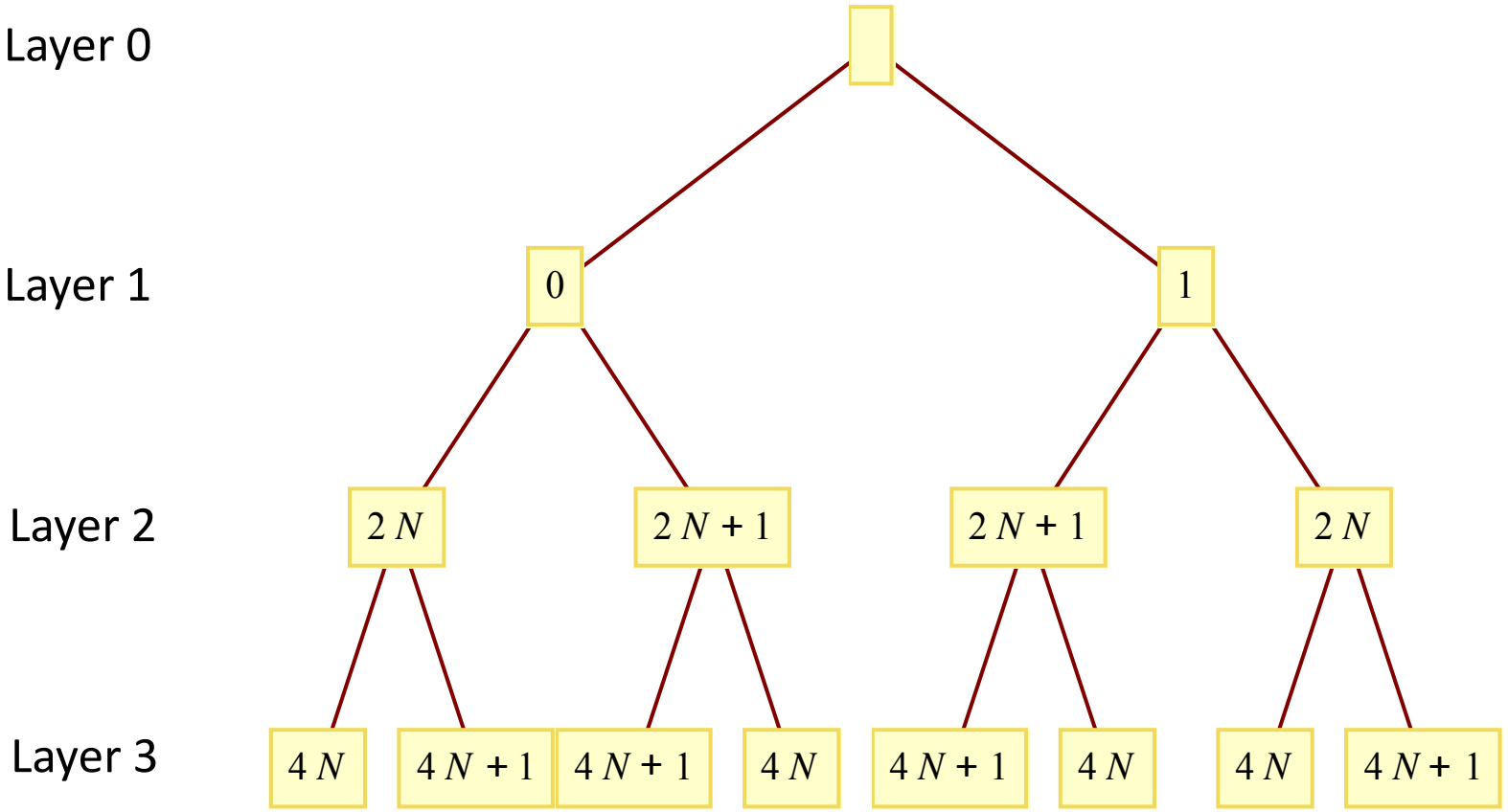
```
COUNTER = X
FUNCTION CAT ( K ):
        IF K > L
                COUNTER = COUNTER – 1
                IF COUNTER == 0
                        PRINT K
                        STOP PROGRAM
                RETURN
        K = K + 2N
        CAT ( K )
        CAT ( K XOR 1 )
WHILE TRUE
        CAT ( 0 )
        CAT ( 1 )
```

Element pushed first onto stack is popped last so order is reversed

Due to the infinite 0's in each stack which alternate between 0 and 1 as last binary bit is flipped

Subtask 3: Analysis

We can construct the recursion tree using the fact that for any integer M, 2M XOR 1 = 2M+1 and (2M+1) XOR 1 = 2M



Subtask 3: Analysis

- DFS only goes deeper if the element is not more than L
- Each layer only contains 2 numbers, 2KN and 2KN+1 for some integer K
- The numbers in 1 layer are either all larger than L or all not larger than L as L is not a multiple of N

Subtask 3: Analysis

- DFS stops at layer $D = \left\lfloor \frac{L}{2N} \right\rfloor + 2$ of the tree which contains the first multiple of 2N that is more than L and visiting each element in this layer decrements the counter

- As there are only 2^D elements in this layer and DFS repeatedly searches the tree, the code prints the $(X\%2^D)$ -th element in the layer

Subtask 3: Analysis

```
COUNTER = X % (2^(floor(L/(2N))+2))
FUNCTION CAT ( K ):
        IF K > L
                COUNTER = COUNTER – 1
                IF COUNTER == 0
                        PRINT K
                        STOP PROGRAM
                RETURN
        K = K + 2N
        CAT ( K )
        CAT ( K XOR 1 )
WHILE TRUE
        CAT ( 0 )
        CAT ( 1 )
```

As L is not a multiple of N, we have $\left\lfloor \frac{L}{2N} \right\rfloor \leq \left\lfloor \frac{79}{2(2)} \right\rfloor = 19$ given the limits of subtask 3 and this cuts the run time enough to solve this subtask

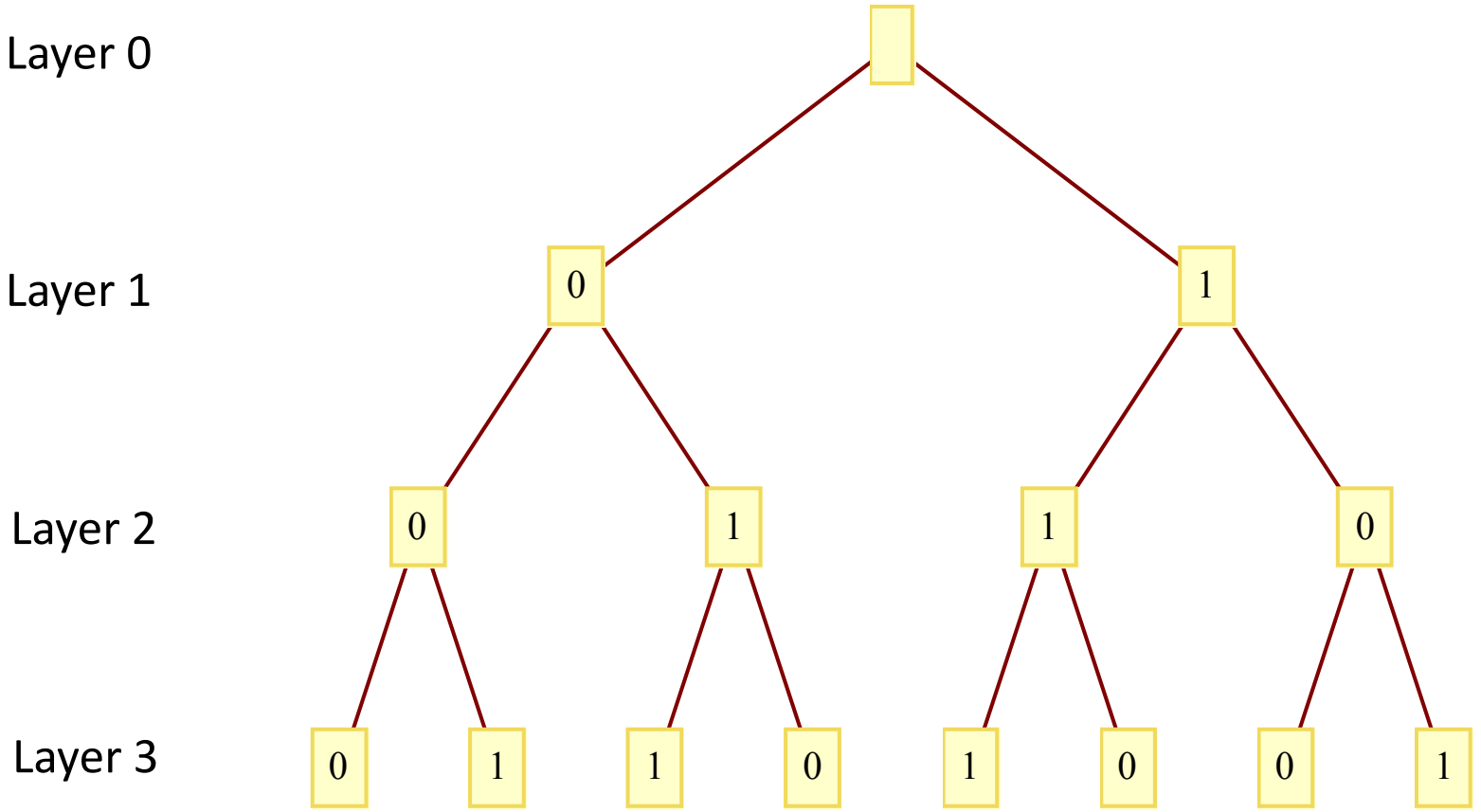
Subtask 4: Pre-computation

- For any layer K in the recursion tree, the elements are $2N(K-1)$ and $2N(K-1)+1$

- Hence, to find the required element we only need to know which layer it is in the tree, which we have shown to be $D = \left\lfloor \frac{L}{2N} \right\rfloor + 2$, and whether it is odd or even

Subtask 4: Pre-computation

We can reconstruct the tree and replace even elements by 0 and odd elements by 1



Subtask 4: Pre-computation

- We can see that to generate the next layer of the tree from a previous layer, we simply replace 0 by 01 and 1 by 10

- Due to this fixed rule, it is possible to show that for any X and any 2 layers with at least X elements, the X-th elements of both layers in this new tree are identical, but we shall omit the technical proof here

Subtask 4: Pre-computation

- Therefore, if we start with a single 0 and repeatedly apply the rule to obtain an infinite sequence of 0's and 1's, the elements in layer K of the new tree for any K is equal to the first 2^K elements of this sequence

- If we label this sequence $\{c_0, c_1, \dots\}$ then the X-th element of layer K is $2N(K-1) + c_{X-1}$

Subtask 4: Pre-computation

- For this subtask, we have $X \leq 1000000$ so we can pre-compute $\{c_0, c_1, \ldots c_{999999}\}$ using the rule above

- We can then answer each query in O(1) using $2N(D-1) + c_{Y-1}$ where $D = \left\lfloor \frac{L}{2N} \right\rfloor + 2$ and $Y = X\%2^D$ as described previously

Subtask 5 & 6: Recurrence

- By analysing the rule of replacing 0 by 01 and 1 by 10, we can obtain the relation
  $c_i = c_{\lfloor \frac{i}{2} \rfloor} \; XOR \; (i\%2)$

- Hence, we can calculate c_X and thus answer each query O(log X) time as per subtask 4

- In fact, c_X is 1 if X has an odd number of 1's in its binary representation and is 0 otherwise but this result is not required to solve the problem as the recurrence is sufficient

Finale

$$p(X) = \text{no. of 1s in binary representation of } X$$

$$f(X, L, N) = 2N\left(\left\lfloor \frac{L}{2N} \right\rfloor + 1\right) + (p(X)\%2)$$

Fun fact: A Turing machine can be simulated by a finite-state machine with 2 stacks so CATS is Turing complete

Task 4: OBELISK

Chin Zhan Xiong, Raymond Kang, Ooi Wei Tsang

Presented by Ooi Wei Tsang

Problem

- Find the **least number of rolls** to move a 1 x 1 x M block from a start point to an end point

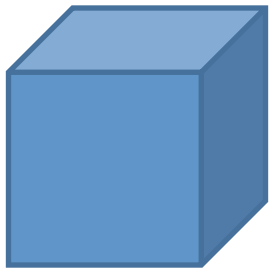
Problem

- Start point and end point can be on **different floors**
- Fall through holes to go lower

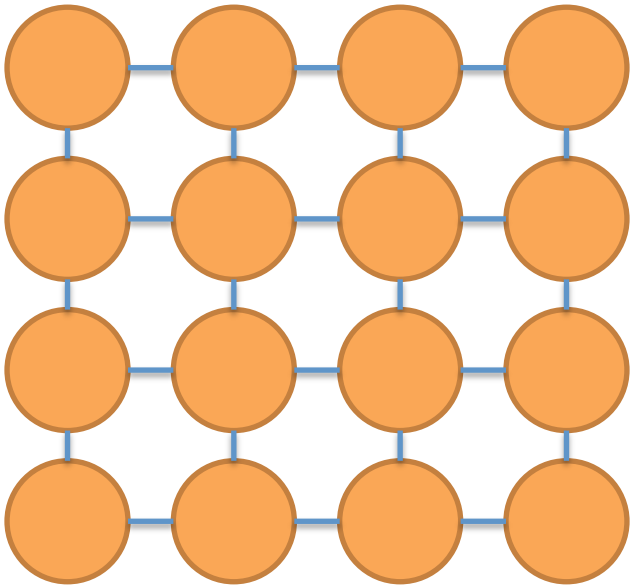
Shortest Path?

Subtask 1

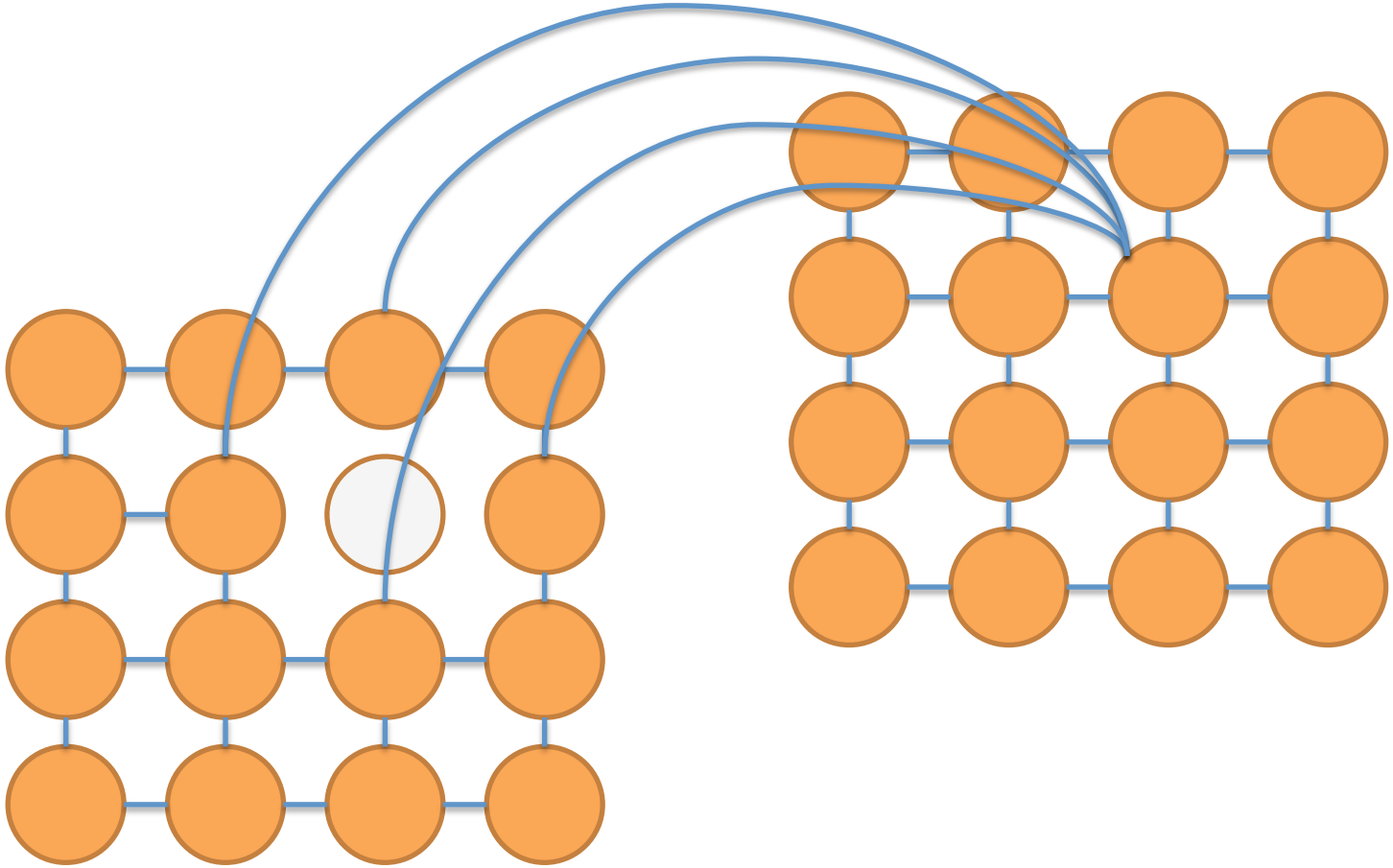
- M = 1
- Coordinates are small (<= 50)



Subtask 1



Subtask 1



BFS State

- state = (floor, row, column)
- $O(|C||F|)$

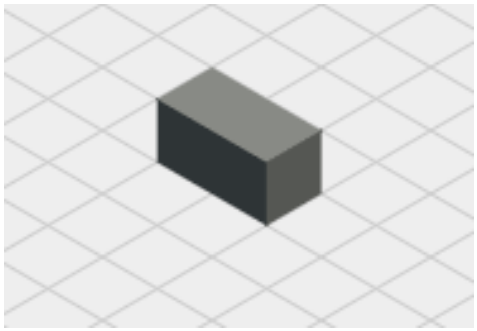
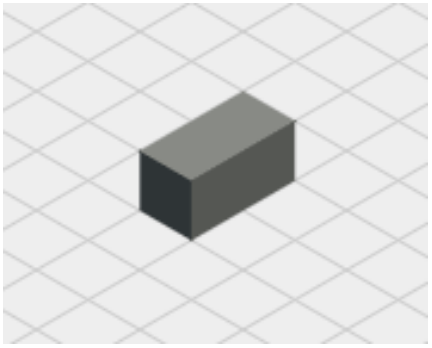
BFS O(|C||F|)

```
solve(source, dest):
Q = new queue()
Q.enq(source)
while Q is not empty
   curr = Q.deq()
   for each adjacent cell c of curr:
       while c is a hole, c.floor ++
       if c.distance is not defined
           c.distance = curr.distance + 1
           Q.enq(c)
return dest.distance
```

Subtask 2

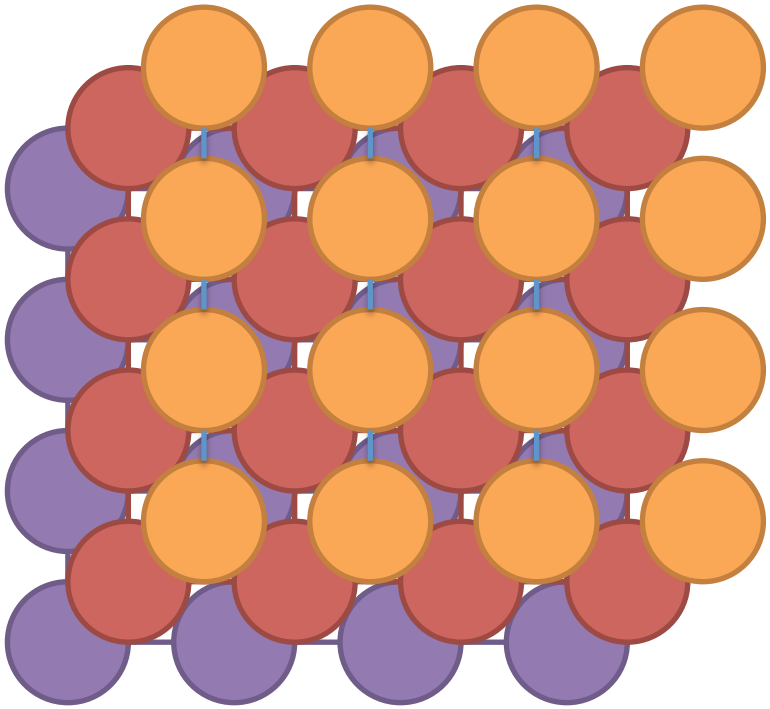
- Same as subtask 1, but M >= 1
- **Orientation** of block matters!
- 3 orientations in total:

Upright Flat vertical Flat horizontal

BFS State

- state = (floor, row, column, **orientation**)
- O(|C||F|)



Subtask 3

- |F| <= 500
- |C| <= 400 x 400
- Number of states = 240 millions
- Memory limit: 256 MB

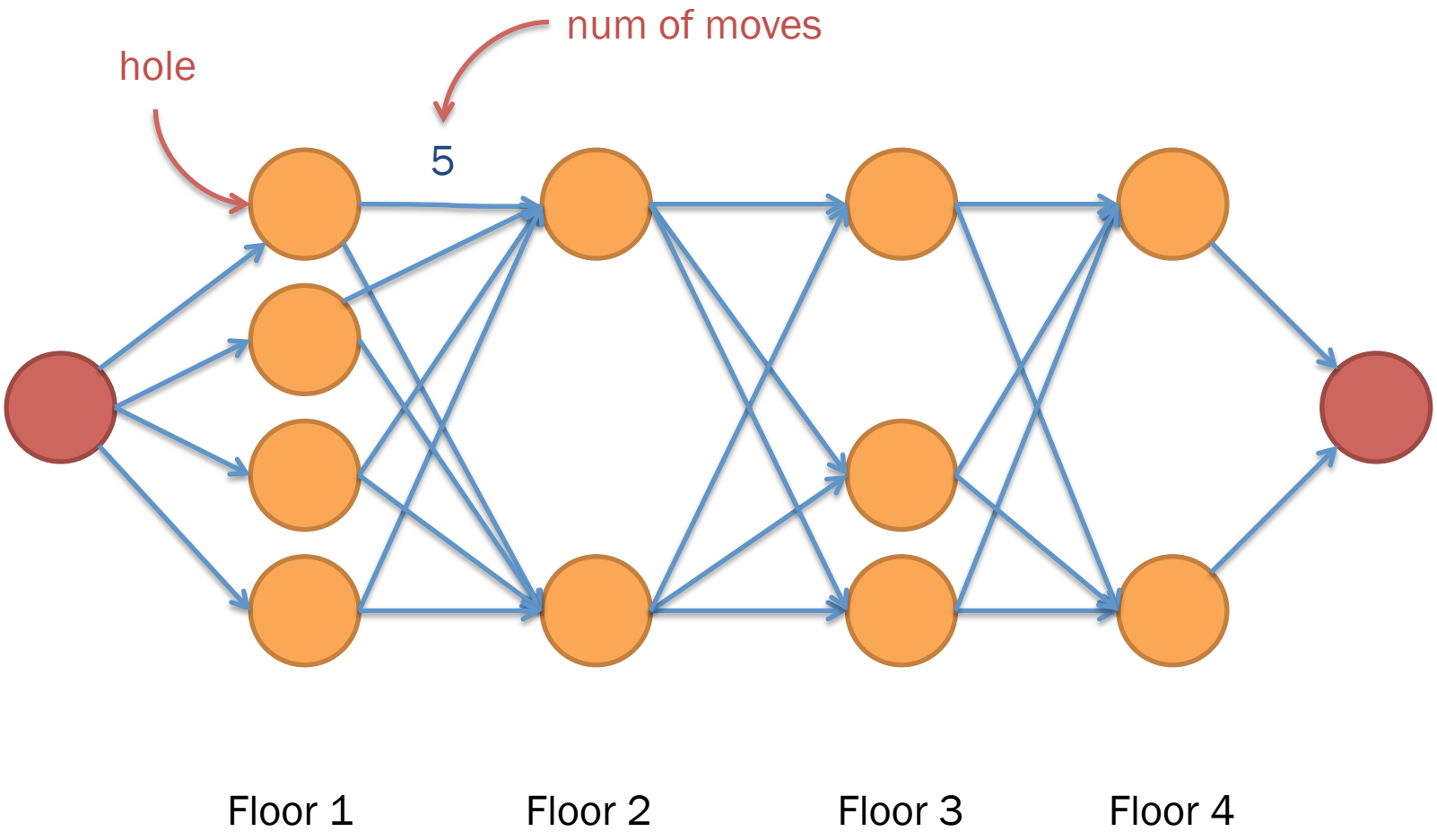
Subtask 3

- Key observation: method to calculate distances on different floors is the same
  - Floor 56: Get from (1, 2) to (3, 4)
  - Floor 64: Get from (1, 2) to (3, 4)
  - Floor 23: Get from (1, 2) to (3, 4)
- If there are 100000 floors, BFS calculates same distance 100000 times!

Idea

- Pre-compute distances in one BFS:
  - From (0, 0) to (x, y) for all x, y <= 400
  - Start and end upright
- **Note:** Limiting coordinates to 400 is insufficient!
  - Add small buffer around 400 x 400 grid

New Graph Model



New graph model

- Movement is always from **hole on a floor** to another **hole on floor below** (except start and end points)
- Let set of vertices be:
  - All holes + Start point + End point
- Weighted edge between two vertices if one is on floor *k* and the other on floor (*k* + 1)
- Find the shortest path in this graph

Weighted shortest path

- Dijkstra's algorithm
  - O(E log E), where E is number of edges
  - $E = O(|H|^2|F|)$
- Or, note that there are no cycles
  - Solved with dynamic programming
  - O(E)

Subtask 4

- Same as subtask 3, but coordinates up to 1,000,000,000
- Impossible to precompute distances!
- So calculate on the spot

Distance from (0,0) to (x,y)

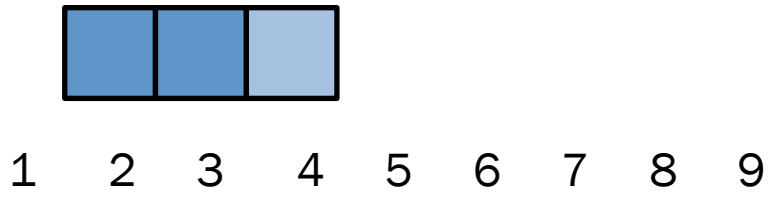
- Find “formula”
- 2D too hard, think about 1D

Distance in 1D

- M = 3
- Get from x = 1 to x = 6

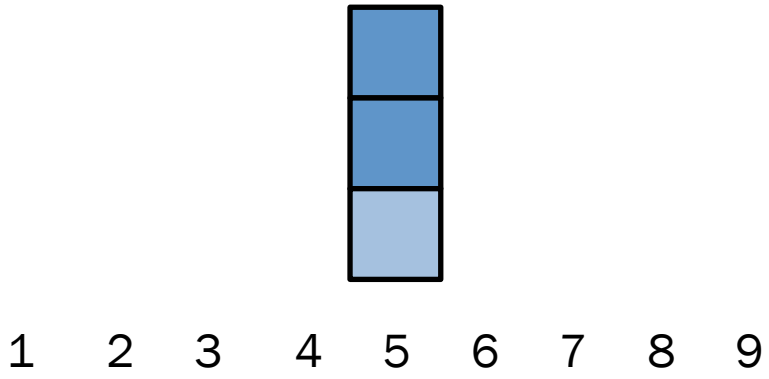
Distance in 1D

- M = 3
- Get from x = 1 to x = 6
- Move 1: Roll →



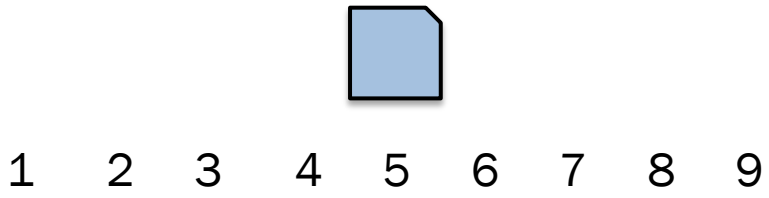
Distance in 1D

- M = 3
- Get from x = 1 to x = 6
- Move 2: Roll →



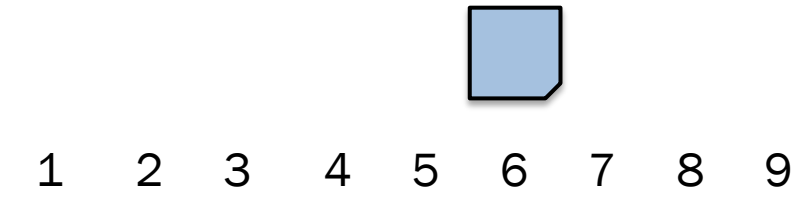
Distance in 1D

- M = 3
- Get from x = 1 to x = 6
- Move 3: Roll into page



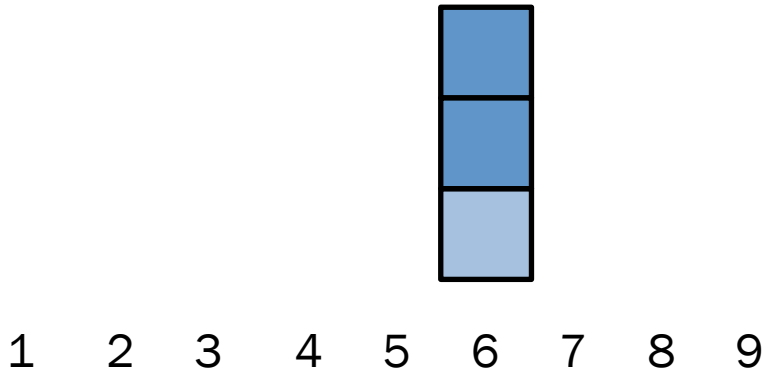
Distance in 1D

- M = 3
- Get from x = 1 to x = 6
- Move 4: Roll →

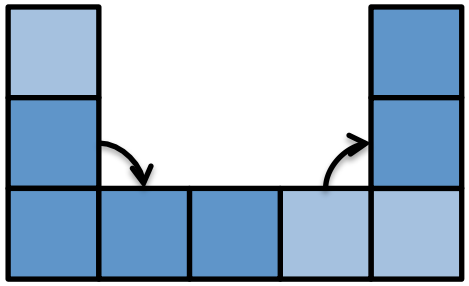


Distance in 1D

- M = 3
- Get from x = 1 to x = 6
- Move 5: Roll out of page

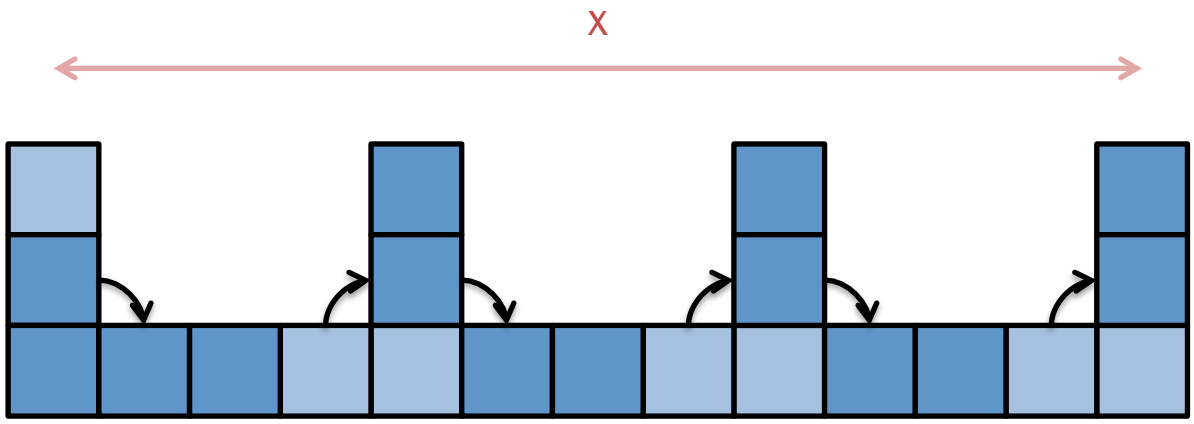


Distance in 1D



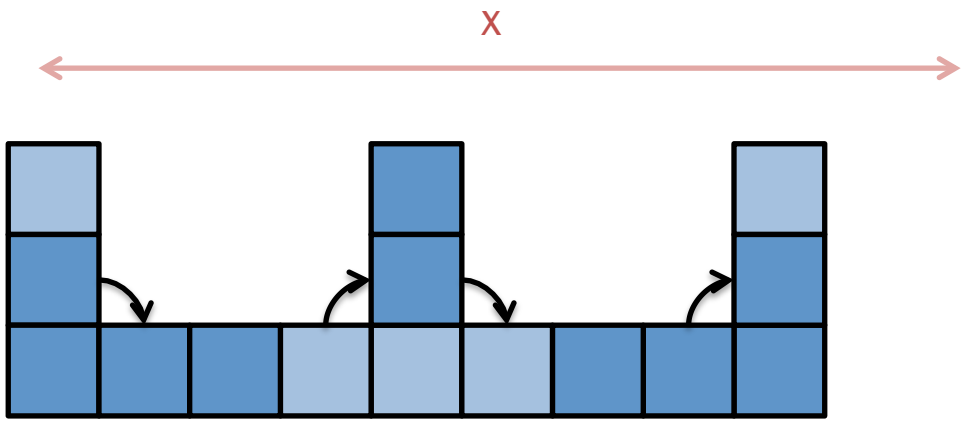
Every two rolls move the obelisk by M+1 cells

Distance in 1D

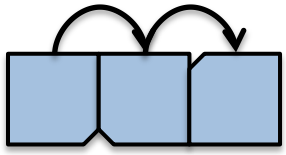


Every two rolls move the obelisk by M+1 cells

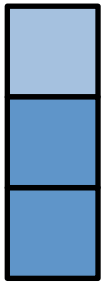
“nice” case: $2\left(\frac{x}{M+1}\right)$



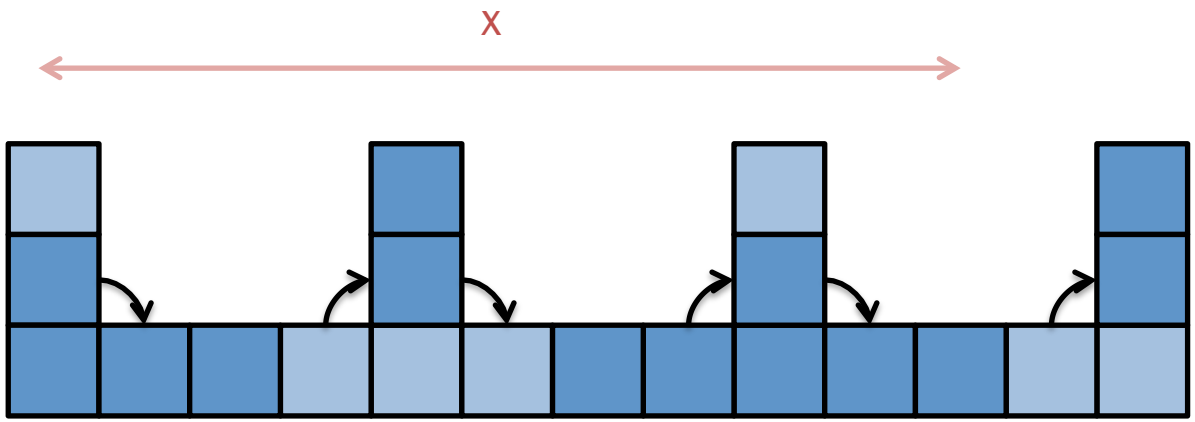
"Cartwheel"



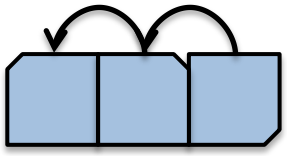
"Roll over"



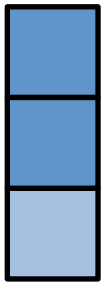
$$2\left\lfloor \frac{x}{M+1} \right\rfloor + 2 + x - (M+1)\left\lfloor \frac{x}{M+1} \right\rfloor$$



"Cartwheel"



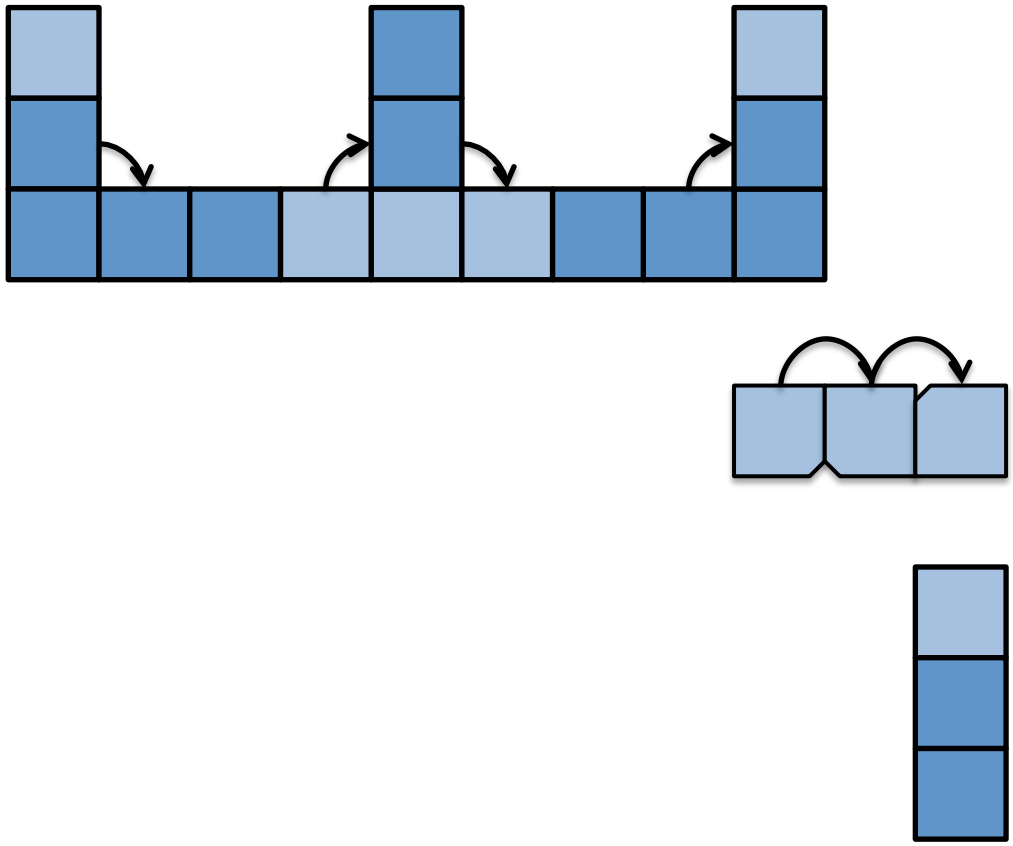
"Roll over"



$$2\left\lceil \frac{x}{M+1} \right\rceil + 2 + (M+1)\left\lceil \frac{x}{M+1} \right\rceil - x$$

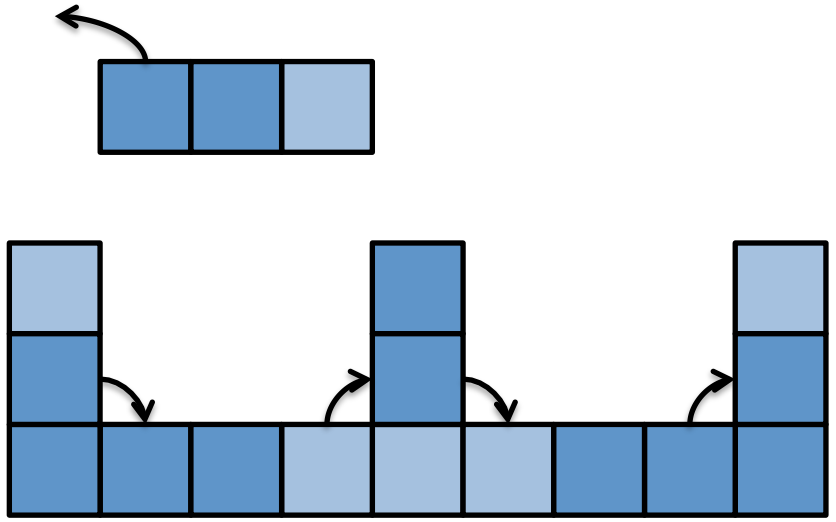
Distance in 2D

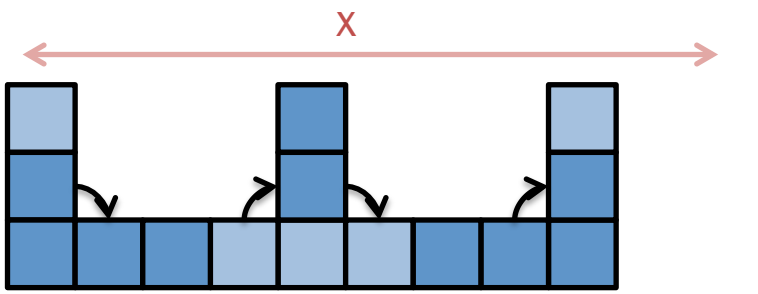
- Sum of 1D distance in x and 1D distance in y

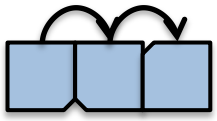


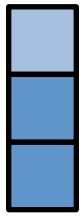
Distance in 2D

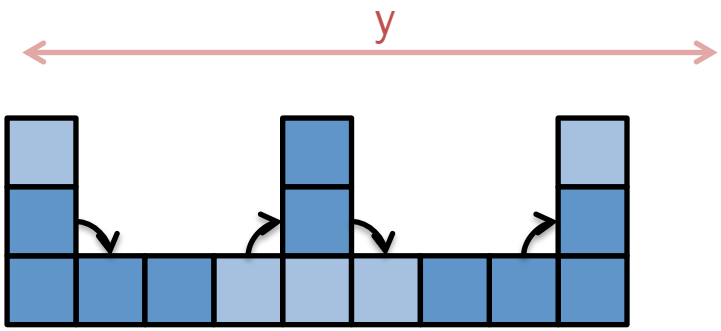
- Sum of 1D distance in x and 1D distance in y

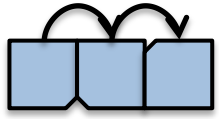


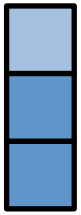

x





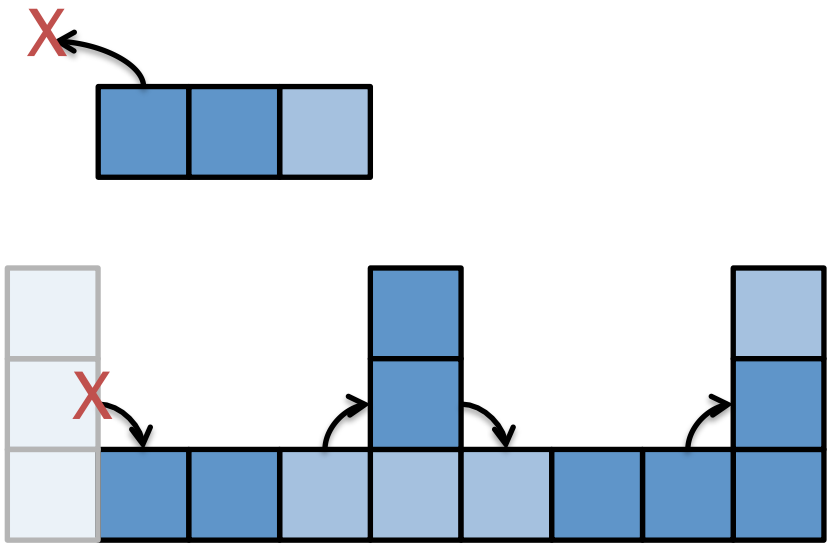

y

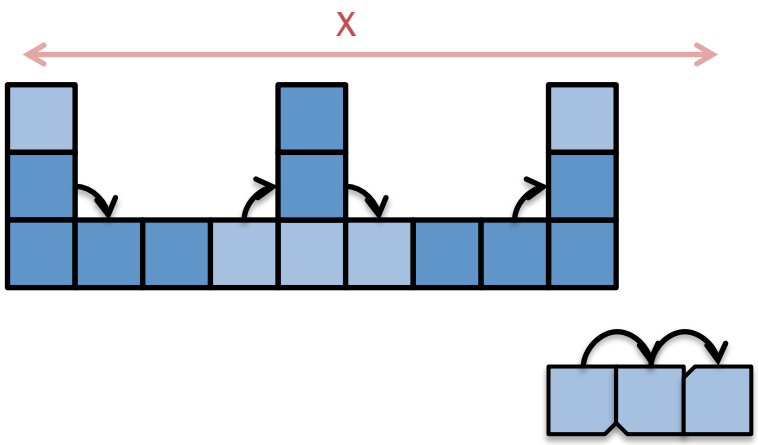


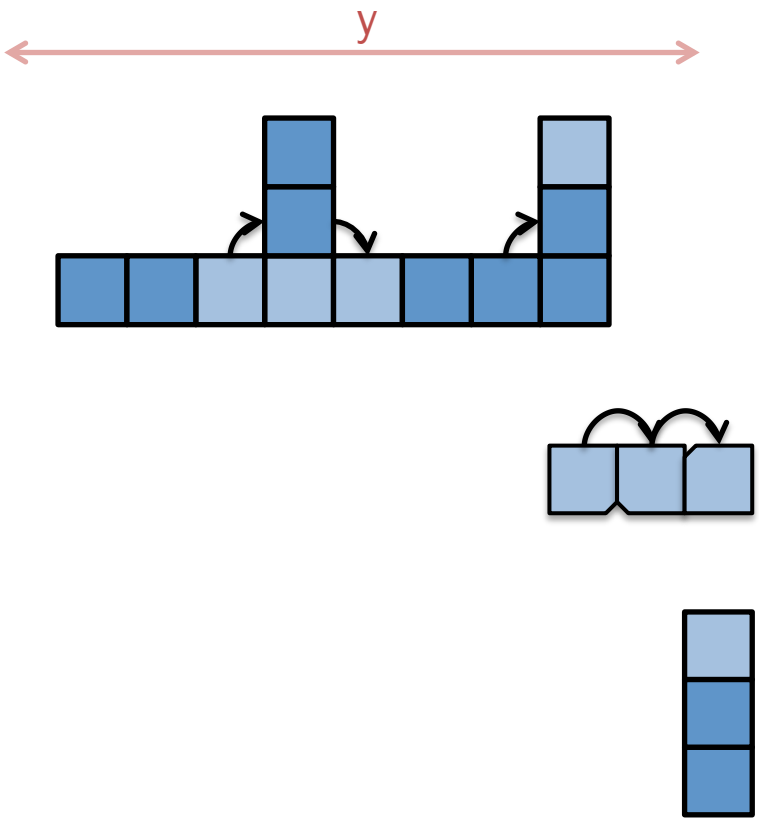


Distance in 2D

- Can save two moves if we ends with “roll over” !




x

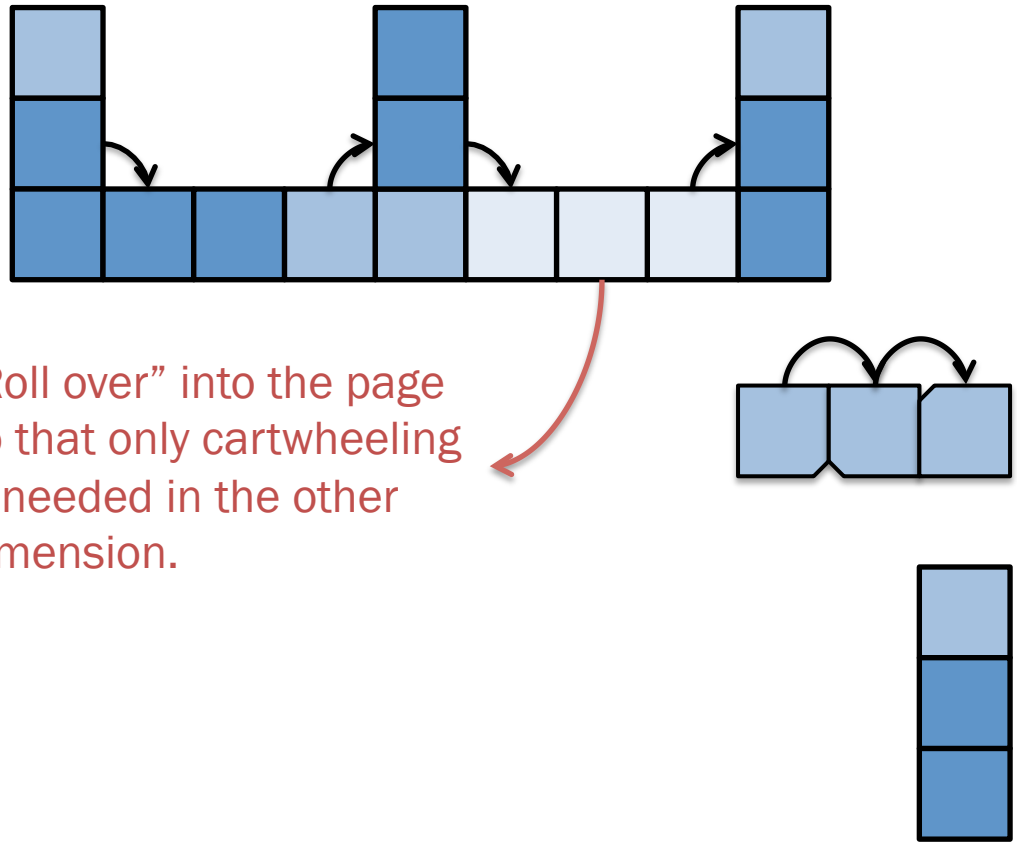

y

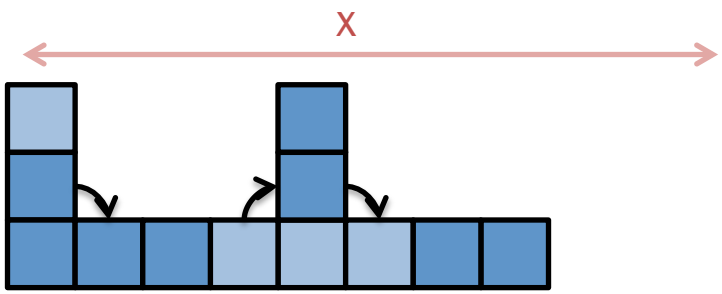
Distance in 2D

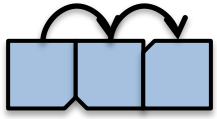
- If both x and y divisible by M+1
  distance2D = distance1D(x) + distance1D(y)
- Else
  distance2D = distance1D(x) + distance1D(y)-2

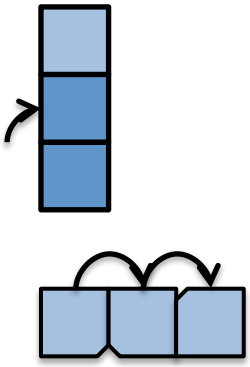
Can we save more?

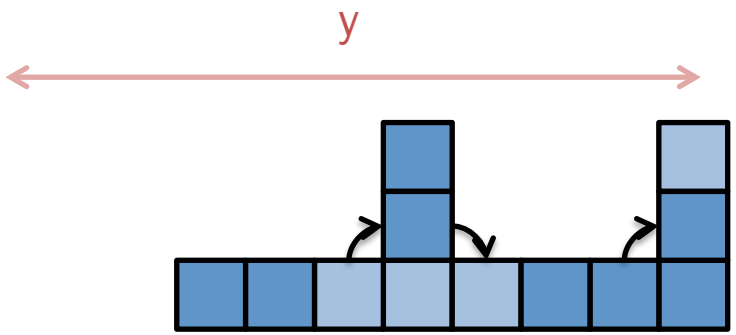
- Yes: can “roll over” while lying down!




x






y

Distance in 2D

- If both x and y divisible by M+1
  distance2D = distance1D(x) + distance1D(y)
- Else if only one of x or y is divisible by M+1
  distance2D = distance1D(x) + distance1D(y) - 2
- Else if both x and y are not divisible by M+1
  distance2D = distance1D(x) + distance1D(y) - 4

Distance in 2D

```
long long dist1D(long long x, bool useHor, bool useVert) {
  if (M == 1) return x;
  if (!x) return 0;
  if (!useHor && !useVert) return INF;
  if (!useHor) return x;
  long long v = x / (M + 1);
  long long sx = v * (M + 1);
  if (sx == x) return 2 * v - 2;
  if (!useVert) return INF;
  long long fromFront = max(0LL, 2 * v - 2) + x - sx;
  long long fromBack = 2 * v + (sx + M + 1) - x;
  return min(fromFront, fromBack);
}
long long getDist(pair<int, int> a, pair<int, int> b) {
  int x = abs(a.first - b.first), y = abs(a.second - b.second);
  long long ans = INF;
  for (int useVert = 0; useVert < 2; ++useVert)
        for (int useHor = 0; useHor < 2; ++useHor)
              ans = min(ans, 2 * (useHor + useVert)
                    + dist1D(x, useHor, useVert)
                    + dist1D(y, useVert, useHor));
  return ans;
}
```