Task 4

Using a heap-like structure

Using a heap-like structure `hp3` to represent unmarked nodes based on `m` values allows getting the m-least unmarked node without linear searching.
Previously in each iteration we need O(1) time to remove a node from the unmarked list, and O(N) time to find the m-least one. As the loop terminates in at most N iterations, the overall time complexity of the algorithm is O(N^2).
Now with a min-heap, in each iteration the removal takes O(log(N)) and getting the minimum takes O(1). In addition, the heap invariant needs to be maintained on update of `m`, each taking O(log(N)) time, but there can be at most E updates, where E is the number of arcs in the graph. So in total the complexity is O(max(NlogN, ElogN)).
Therefore, it makes the algorithm much more efficient when `E << N^2` (i.e. the graph is sparse).

Other variables to be added

Other than the heap `hp3`, which is a list of (m value, node id) pairs, we also need to keep track of the index of each node's corresponding pair in `hp3`, so a list `idx3` needs to be added. The coupling invariants are:

- `(m[i], i) in hp3` iff `un1[i] == 1`, and `len(hp3) == nun1`
- `idx3[i]` is the `hp3`-index of pair (m[i], i) if `un1[i] == 1`, otherwise `idx3[i] == N`

With these variables, `un1` and `nun1` are not necessary any more and can be removed.

Statements involving `m` that are affected

Some statements not involving `m` are also affected, but not listed here because they are not asked. Instead, please see the refinements in the code below directly.

1. After initialisation of m (`m= [INFTY]*N; m[0]= 0`)

The heap and indices needs to be initialized, with the pairs (m[i], i) for all unmarked nodes:

```
hp3 = [(m[i], i) for i in range(N)]
idx3 = list(range(N))
```

Because all `m[i]`s are ∞ except that `m[0] == 0`, after the above initialization `hp3` is already in heap structure, so no need to explicitly heapify it.

2. After updating m (`m[nTo]= newD`)

The heap also needs to be updated with the new m value:

```
heapupdate(hp3, idx3, (newD, nTo))
```

This decreases `hp3[idx3[nTo]]` to `newD`, then re-establish the heap invariant by repeatedly moving the updated element upwards (until its m-value >= its parent's), and updating `idx3` meanwhile.

3. Getting the m-least unmarked node (statements `for n in range(N): if un1[n]: if m[n] <=minD: tn= n; minD= m[n]`)

Now we can replace these statements by reading the m-least unmarked node directly from the heap top:

```
if hp3:  # if there are remaining unmarked nodes
    minD, tn = hp3[0]
```

This is essentially a `peek`, but not `pop`. We do not pop it until the beginning of next iteration, when we mark the node `tn`, to maintain the coupling invariant.

Code

```
import sys; INFTY= 1000

def heappop(hp, idx):
    '''
    Pop the minimum element and re-establish the heap invariant, in
O(logN) time.
    Restore the heap invariant by first swapping the last element to the
first and deleting the last element, then repeatedly moving the first
element down until its m-value <= its children's. Meanwhile, `idx` is
updated.
    '''
    # update top and remove redundancy
    hp[0] = hp[-1]; del hp[-1]
    if not hp:  # hp becomes empty
        return
    idx[hp[0][1]] = 0
    # re-heapify
    i, n = 0, len(hp)
    while 2*i+1 < n:  # while node i has at least the left child
        smallest = i
        if hp[2*i+1] < hp[i]:
            smallest = 2*i+1
        elif 2*i+2 < n and hp[2*i+2] < hp[i]:
            smallest = 2*i+2
        if smallest == i:  # heap invariant restored, adjusting finished
            break
        # otherwise do the swap and continue testing with new children
        idx[hp[smallest][1]] = i
        idx[hp[i][1]] = smallest
        hp[smallest], hp[i] = hp[i], hp[smallest]
        i = smallest


def heapupdate(hp, idx, (newD, nTo)):
```

```
    '''
    Decrease the m-value of hp[idx[nTo]] to newD and re-establish the heap
invariant, in O(logN) time.
    Restore the heap invariant by repeatedly moving the updated element
upwards (until its m-value >= its parent's), and update `idx` meanwhile.
    '''
    hp[idx[nTo]] = (newD, nTo)
    # Since it's decreased, it could only be moved upwards but not
downwards
    i = idx[nTo]
    parent = (i-1)//2
    while i > 0 and hp[i] < hp[parent]:
        idx[hp[i][1]] = parent
        idx[hp[parent][1]] = i
        hp[i], hp[parent] = hp[parent], hp[i]
        i = parent
        parent = (i-1)//2  # becomes negative when i == 0 but doesn't
matter because it will never be accessed (we have loop guard i>0)

print "How many nodes? ",
N= int(sys.stdin.readline()) # N= number of nodes.
print "Nodes are 0..%d, with initial node %d." % (N-1, 0)

# Three white-space separated integers representing (from,dist,to) per
line.
print "Now enter the edges:"
G2 = [[] for i in range(N)]
for line in sys.stdin.readlines():
    nFrom, dist, nTo = map(int, line.split())
    G2[nFrom].append((dist, nTo))  # put every arc in the respective list
m= [INFTY]*N; m[0]= 0   # m:=  initially INFTY except for initial node
#>> un1 = [1]*N; nun1 = N
hp3 = [(m[i], i) for i in range(N)]
idx3 = list(range(N))
#<<
tn= 0                   # tn:= initial node

# Initialisation has established the invariants:
#   I1--- For all marked nodes, m has the least distance from 0. // A.
#   I2--- For all unmarked nodes, m has least all-marked distance from 0.
// B.
#   I3--- Node tn is unmarked, and is m-least among the unmarked nodes. //
C.

while 1:
#>> un1[tn] = 0; nun1 -= 1
#>> if nun1 == 0: break
    heappop(hp3, idx3)  # mark tn, which is at the heap top
    if len(hp3) == 0: break
#<<

    # Re-establish I2. // G.
    for (dist, nTo) in G2[tn]:
#>>     if un1[nTo]:
```

```
        if idx3[nTo] != N:
#<<
            newD= m[tn]+dist
            if newD<m[nTo]:
                m[nTo]= newD
#>>
                heapupdate(hp3, idx3, (newD, nTo))
#<<

    # Re-establish I3. // H.
    minD= INFTY
#>> for n in range(N):
#>>   if un1[n]:
#>>     if m[n]<=minD: tn= n; minD= m[n]
    if hp3:  # if there are remaining unmarked nodes
        minD, tn = hp3[0]
#<<
    if minD==INFTY: break # All remaining nodes unreachable. // I.
###
#   I1,I2 and m is INFTY for all nodes in un1. // J,K,L,M.

print "Least distances from Node 0 are:"
for n in range(N):
    if m[n]!=INFTY: print "Distance to Node %d is %d." % (n,m[n])
    else:           print "Node %d is unreachable." % n
```