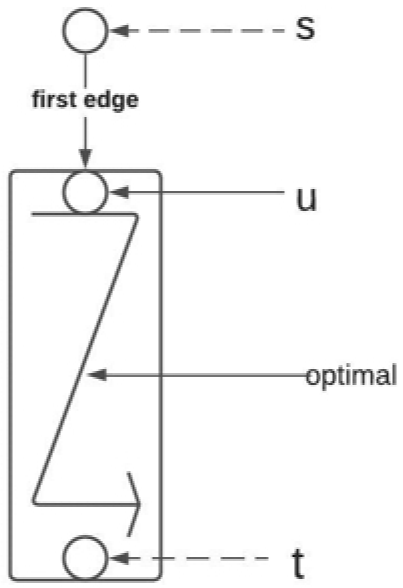
Assignment 5

Email	neis@umn.edu
Name	Kelsey Neis
Section	001

15-1 DAG Longest path

Optimal substructure: the optimal substructure will be the first edge versus the rest of the path, which is optimal.



Prove that u → t is optimal:

1. Assume we have a path from s → t which is optimal, where u → t is **not** optimal. Say the path from s → u is X and the path from u → t is Y, so the full path is then Z = X + Y
2. Then, it is possible to modify u → t to make it optimal. This would increase the length of Y by some amount, q.

3. This would result in s → t increasing to Y + q, increasing the total length: Z = X + Y + q.
4. Since the original full path was optimal, but modifying the path s → t increases the length of the full path, this leads to a contradiction. Therefore, the path from s → t must be optimal to begin with.

Find a top-down recursive algorithm for finding the length of the longest path

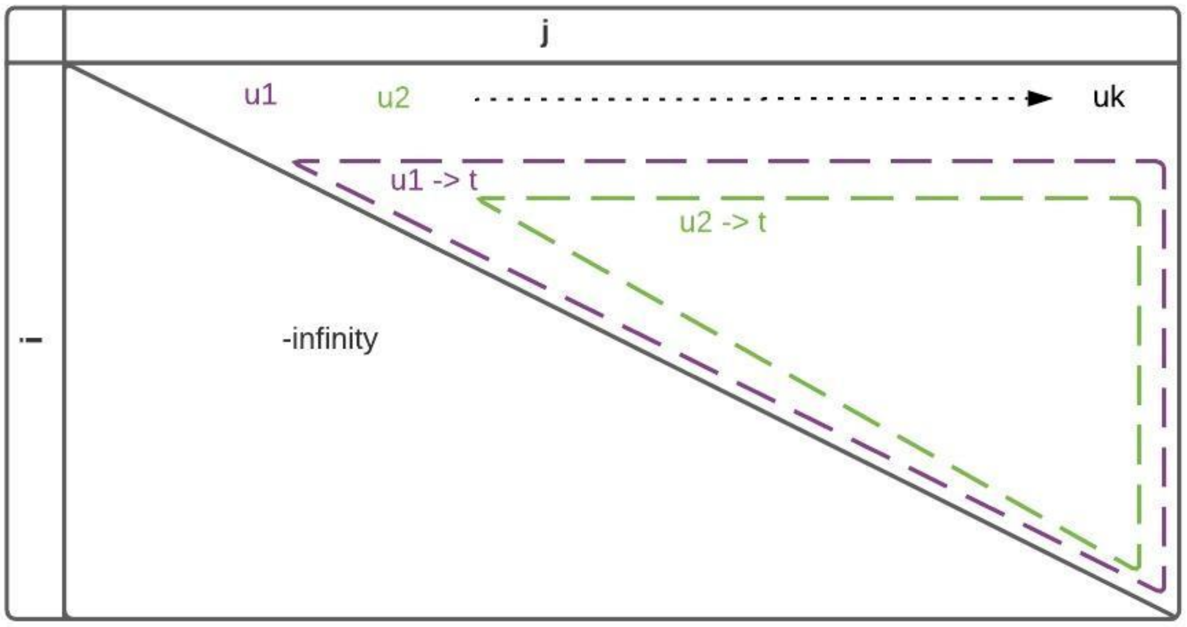
```
LongestPath(s, t, w):
    if s == t-1:
        return 0
    m = -∞
    for i = s+1 to t:
        m = max(m, (w[s][i] + LongestPath(i, t, w)) )
    return m
```

This algorithm takes theta(n ^ 2) runtime, to cover all possible combinations of i and j.

Record the optimal lengths

We need to find the combination of edges that yields the longest path. The recursive relationship of the first edge and the rest of the path will get us this answer. Below is an illustration of the optimal substructures. If u1 is chosen for the first edge, u1 → t is the rest of the path. If u2 is chosen, u2 → t is the rest of the path. L[u, k] will be an array holding the index of u and the total length of the path from s → u + the remainder of the path u → t.

Table of Weights w



To fill in the array L[u, k], we recursively call LongestPath, each time adding an entry to L with the result from the calculation of the max from recursing down to u = t.

```
LongestPath(s, t, w, L):
    if s == t - 1:
       return {u: s, v: t, length: w[s][t]}, L // base case: u and t are adjacent
    max = -∞
    for i = s+1 to t:
        l = LongestPath(i, t, w, L)
        // here MAX preserves the object structure, rather than simply integer
        max = MAX(max, l)
        if max == l
          // add the result of max to the L array
          new_max = { u: s, v: i, length: l.length + w[s][i] }
          L.add( new_max )
          return new_max
    return max, L
```

The recursion depth is n^2, because you have to look at all of the possible combinations between i and j in order to find the combination that gets the longest path.

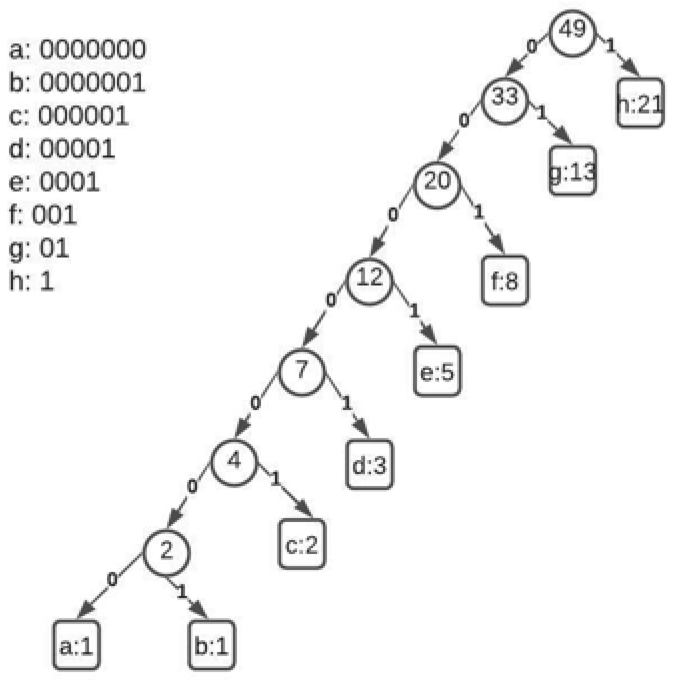
What remains is to use the L array, traversing it using the values of u and v in order to skip over the vertices that should be eliminated for not being greater than a value already in the optimal solution, and the final value of max to reconstruct the longest path.

```
ReconstructPath(max, L):
  A = []
  v = max.v
  for i = max.u to L.length:
    A.add(i)
    i = v
    v = L[v].v
  return A
```

The runtime of ReconstructPath is linear. It is at most n, so theta(n)

16.3

3 - Huffman code for Fibonacci sequence



To add more Fibonacci numbers, for each one, add a "0" to the beginning of all of the other lower frequency letters, and the highest frequency letter will have the code "1". This is because adding up the sum of the previous two numbers plus the next Fibonacci number is always between the next two, so the next level will always involve one node and one leaf.

i = 1 : C = "0"*(n-i)

otherwise: C = "0"*(n-i) + "1"

(note that when i = n, it is the highest frequency, so no "0", but one "1")

6 - transmit codes

```
treeStructure(g, i, d):
    code_map = []
    for i= 0 to g.length:
        if g[i]['l'] == null:
            code = g[i]['dir']
            j = i
            while j > 0:
                j = g[j]['parent']
                if j > 0:
                    code = g[j]['dir'] + code
            code_map.add({ 'code': code, 'num': i})
        else:
            g[g[i]['l']]['parent'] = i # set parent
            g[g[i]['l']]['dir'] = 'l' # set l
            g[g[i]['r']]['parent'] = i # set parent
            g[g[i]['r']]['dir'] = 'r' # set r
    return code_map
```

The outer for loop will execute 2n - 1 times, as there are that many nodes. The while loop will execute nlgn times, since the code traverses up the tree via the parents, which corresponds to the height, lgn. This is done for each letter to get the corresponding code.

7 - Huffman ternary

```
HuffmanTernary(C):
  n = |C|
  Q = C
  if (2n - ceil(n/2) - 1) % 3 > 0:
    allocate new node m
```

```
5    m.freq = 0
6    Insert(Q, m)
7    n++
8  for i = 1 to n - 1:
9    allocate a new node z
10   z.left = x = EXTRACT-MIN(Q)
11   z.right = y = EXTRACT-MIN(Q)
12   z.middle = m = EXTRACT-MIN(Q)
13   z.freq = x.freq + y.freq + m.freq
14   INSERT(Q)
15 return EXTRACT-MIN(Q)
```

In order to make the code optimal, the solution must create a perfect tree. To make a perfect ternary tree, with each node having 3 children, the number of nodes below the root must be divisible by 3. The total number of nodes is floor(3n/2), since for every leaf, we're adding a node, but this is shared with two other leaves (or nodes), so subtracting the root, you get floor(3n/2) - 1. So, in lines 3-7, if the result is not divisible by 3, a placeholder node must be created with a frequency of 0. This ensures that the placeholder will be at the bottom of the tree, and will not break the optimal structure.

Proof:

1. The proof is similar to the proof for the binary variant:
2. Say a tree T is optimal for C
3. T' = C' = C \ {x, y, m}U{z}
4. f(z) = f(x) + f(y) + f(m)
5. B(T) = B'(T) + f(z)
6. Assume T' is not optimal
7. Change T' to make optimal for C' and call it ~T'
8. B(~ T') = B(~T') + f(z)
9. B(T') > B(~T')
10. B(T) > B(~T)
11. This leads to a contradiction, because the original assumption was that B(T) was optimal. So the assumption that the substructure is not optimal is false.

9 - Random files

In order to compress a file to a smaller size, you must be able to describe the file in fewer bits than the original file. If an n-bit file is random, there are n! possible permutations of the file. Without any pattern to use for describing the file, the number of possible compressed files will also be n!.