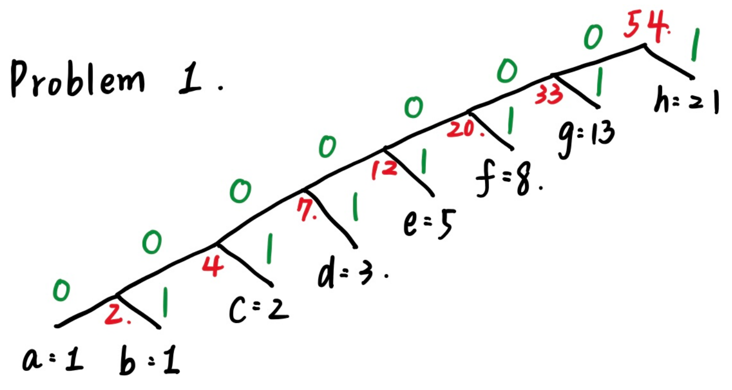
Algorithm HW3

B06702064 會計五 林聖硯

No collaborator

Problem 1

參考資料: https://sites.math.rutgers.edu/~ajl213/CLRS/Ch16.pdf (https://sites.math.rutgers.edu/~ajl213/CLRS/Ch16.pdf)



a = 0000000
b = 0000001
c = 000001
d = 00001
e = 0001
f = 001
g = 01
h = 1

The result can be generalized to the following fact. Assume we have n Fibonacci numbers (from the least frequent to the most frequent), when $k < n$, the huffman code of the k-th most frequent letter is $0^{k-1}1$ (duplicate 0 for $k-1$ times and append a 1 behind it). When $k = n$, which is the least frequent letter (the n-th most frequent letter), its huffman code will be 0^{n-1}

After finding the optimal huffman code for the first 8 fibonacci sequences, I found that the $n+1$ th Fibonacci number is always greater than the sum of the i-th Fibonacci numbers, where $i = 0, 1, \ldots, n-1$. Also, their relations can be formulated as follow

$$\sum_{i=0}^{n-1} F(i) = F(n+1) - 1$$

where $F(i)$ is the i-th Fibonacci number, with $F(0) = 1$ and $F(1) = 1$.

Now, I'm going to prove this formula by induction.

Base case: when $n = 1$, $F(0) = 1 = F(2) - 1$

Induction: Suppose that when $n = k$, the formula holds. i.e. $\sum_{i=0}^{k-1} F(i) = F(k+1) - 1$

When $n = k+1$, $\sum_{i=0}^{k} F(i) = \sum_{i=0}^{k-1} F(i) + F(k) = [F(k+1) - 1] + F(k) = F(k+2) - 1$.

QED

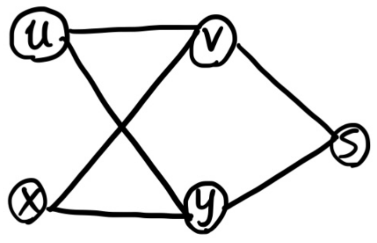
Now I need to prove that, at each stage, we can maintain one subtree which contains all of the least frequent letters as we greedily combining the nodes. Intially, the subtree will contain the least frequent letter. Then, at stage k, we assum that the subtree contains the k least frequent letters. The subtree has weight $\sum_{i=0}^{k-1} = F(k-1) - 1$ and all other nodes have weights $\{F(k), F(k+1), \ldots, F(n-1)\}$. The greedy algorithm tells us that we need to combine the subtree and $F(k)$ node since they are the two lowest weight subtrees. By combining them, the above argument tells us that the subtree will maintains its property again, thus completing the induction.
The above algorithm is similar to the HUFFMAN algorithm in the textbook, thus the correctness can also be proved by Lemma 16.2 and Lemma 16.3 similarly.

Problem 2

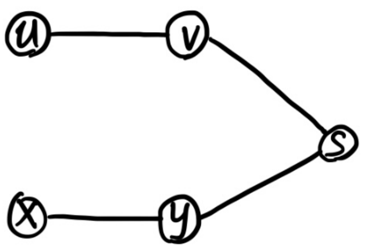
参考資料: https://walkccc.me/CLRS/Chap22/22.2/ (https://walkccc.me/CLRS/Chap22/22.2/)

Problem 2.

Let $G = (V, E)$ be.

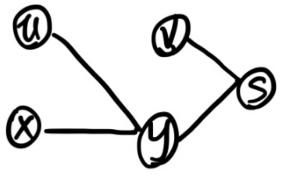


Let $G_\pi = (V, E_\pi)$ be.

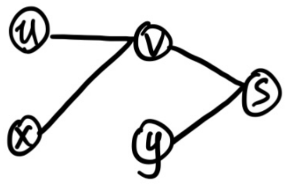


Run BFS on G from vertex s.

If y precedes v in Adj[s], we get



If v precedes y in Adj[s], we get



Problem 3

参考資料: https://walkccc.me/CLRS/Chap22/22.3/ (https://walkccc.me/CLRS/Chap22/22.3/)

```
DFS-VISIT-PRINT(G, u)
    time = time + 1
    u.d = time
    u.color = GRAY
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            print "(u, v) is a tree edge."
            v.π = u
            DFS-VISIT-PRINT(G, v)
        else if v.color == GRAY
            print "(u, v) is a back edge."
        else //v.color == BLACK
            if v.d > u.d
                print "(u, v) is a forward edge."
            else
                print "(u, v) is a cross edge."
    u.color = BLACK
    time = time + 1
    u.f = time
```

Problem 4

參考資料: https://sites.math.rutgers.edu/~ajl213/CLRS/Ch22.pdf (https://sites.math.rutgers.edu/~ajl213/CLRS/Ch22.pdf)

```
FIND-PATH(u, v)
    if u == v
        return 1
    else if u.paths != NIL
        return u.paths
    else
        for w ∈ Adj[u]
            u.paths = u.paths + FIND-PATH(w, v)
        return u.paths
```

The attribute u.paths of node u records the number of simple paths from u to v, assuming that v is fixed throughout the entire process. We can the sum the number of paths which leave from each of u's neighbors to find the paths from u to v. Since we have no cycles, we will never risk adding a partially completed number of paths and we do not have to consider the same edge twice in the recursive calls. The time complexity of the algorithm is \(O(|V|+|E|)\) since we store the number of paths in attribute u.paths and we reach each vertex and node once.

Problem 5

參考資料:

1. https://www.chegg.com/homework-help/euler-tour-euler-tourof-connected-directed-graph-g-v-e-cycle-chapter-22.p-problem-3p-solution-9780070131514-exc (https://www.chegg.com/homework-help/euler-tour-euler-tourof-connected-directed-graph-g-v-e-cycle-chapter-22.p-problem-3p-solution-9780070131514-exc)
2. https://walkccc.me/CLRS/Chap22/Problems/22-3/ (https://walkccc.me/CLRS/Chap22/Problems/22-3/)

Prove (=>), i.e. if G has an Euler tour, then in-degree(v) = out-degree(v) for each vertex v \(\in\) V.

An Euler tour can be decomposed into a set of edge-disjoint simple cycles. If G has an Euler tour, each vertex **in a simple cycle** must have one entering edge and one leaving edge. Therefore, in each simple cycle, the degrees are either 1 (if v is on the simple cycle) or 0 (if v is not on the simple cycle). Adding all in-degrees and out-degrees over all vertex, we prove that in-degree(v) = out-degree(v) for all vertex in v.

Prove (<=), i.e. if in-degree(v) = out-degree(v) for each vertex v \(\in\) V, then G has an Euler tour

Prove by contradiction. Suppose that there is one vertex v whose in-degree and out-degree is not equal, assuming that in-degree(v) \(>\) out-degree(v). If \(v\) is the start of the cycle, in whatever cycle we take, after getting in v for "out-degree" times, there are still unused edges going in that we need to use. Thus, this graph can't be an Euler tour, which means that our assumption is False. The reverse case (out-degree(v) \(>\) in-degree(v)) can be proved with a similar argument on its transpose graph. Thus, if i.e. in-degree(v) = out-degree(v) for each vertex v \(\in\) V, then G has an Euler tour

Problem6

If an Euler tour really exists, we can just arbitrarily walk on any edge as long as we didn't walk on the same edge. In the end, we must end up with a valid Euler tour.

```
FIND-EULER-TOUR(G)
    color all edges WHITE
    answer = [] //record the visiting vertex
    randomly pick a vertex v
    answer.append(v)
    while there are some WHITE edges (v, w) coming out of v
        color the edge (v, w) BLACK
        v = w
        append v to L
    return answer
```

Problem7

(a)
The worst case happens when we partition the problem into two subproblems, where there problem size is \(n-1\) and \(1\) respectively. (Notice that it's impossible to partition the problem into \(1\):\(n-1\), since the least frequent character couldn't have 50% of the total frequency. Otherwise, it will lead to contradiction.) Thus, the recurrence can be formulated as

$$T(n) = \begin{cases} \Theta(1), \text{ if } n = 1 \\ T(n-1) + n, \text{ if } n > 1 \end{cases}$$

where n in the second equation is the assigned cost in each subproblem.
Sorting time complexity: $O(n \lg n)$
Recurrence time complexity: $T(n) = 1 + 2 + \cdots + n = O(n^2)$
Thus, the worst-case time complexity is $O(n^2)$

(b)

No, this algorithm doesn't produce optimal prefix code.
If A = 3, B = 5, C = 8, D = 10, the cost of this algorithm is 52. However, we can obtain the optimal cost 50 by Huffman algorithm.

Problem 8

(a)

參考資料: https://walkccc.me/CLRS/Chap16/Problems/16-1/ (https://walkccc.me/CLRS/Chap16/Problems/16-1/)

Greedy algorithm: We can always give the largest denomination, which is smaller than the remaining change and repeat this procedure until there's no more remaining change. Since we the denominations include a one dollar, there must be a solution for every value of n.

Proof of greedy-choice property:

Greedy-choice property: Always give the largest denomination, which is smaller than the remaining change.
Given an optimal solution $(x_0, x_1, \ldots, x_k)$ where x_i indicates the number of coins of demonimination c_i. Suppose that we have some $x_i \geq c$, we can decrease x_i by c and increase x_{i+1} by 1. In this way, the total value of coins will remain the same and the number of coins will decrease by $c - 1$. Thus, the original solution must be non-optimal.

(b)
Let the denominations be {7, 5, 1} and we want to make change for 10. The solution of greedy algorithm is {7, 1, 1, 1}. However, the best solution is {5, 5}.

(c)

```
Make-Changes(n, d, k) //d is an array contining k denominations
    Let c[0, ..., n] and p[1, ..., n] be new arrays
    //c[j] contains the minimum number of coins we need to make changes j dollars
    //p[j] records the size of change

    c[0] = 0
    for i = 1 to n
        c[i] = infinity

    for i = 1 to k
        for j = d[i] to n
            if c[j] < c[j - d[i]]
                c[j] = c[j - d[i]] + 1
                p[j] = j - d[i]

    //output solution
    while n > 0
        give a coin of size (n - p[n]) as change
        n = p[n]
```

The time complexity is $T(n) = O(n) + O(nk) + O(n) = O(nk)$ (The last while loop runs at most n times)