1 HW1 optional exercises

1.1 HW1, Problem 7

1. $T(n) = n^2 + n$ (prove by induction)
2. $T(n) = 2^n - 1$ (prove by induction)

1.2 HW1, Problem 8

First, prove $\sum_{i=1}^{n} \frac{1}{i} = O(\log n)$. For simplicity, assume $n = 2^k$ (easy to get rid of the assumption).

$$\begin{aligned}
& 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \ldots + \frac{1}{n} \\
\leq\ & 1 + \left(\frac{1}{2} + \frac{1}{2}\right) + \left(\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}\right) + \ldots + \underbrace{\left(\frac{1}{2^{\log n}} + \ldots + \frac{1}{2^{\log n}}\right)}_{n} \\
\leq\ & \log n
\end{aligned}$$

Similarly, prove $\sum_{i=1}^{n} \frac{1}{i} = \Omega(\log n)$.

$$\begin{aligned}
& 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \ldots \\
\geq\ & \frac{1}{2} + \frac{1}{2} + \left(\frac{1}{4} + \frac{1}{4}\right) + \left(\frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8}\right) + \ldots + \underbrace{\left(\frac{1}{2^{\log n}} + \ldots + \frac{1}{2^{\log n}}\right)}_{n/2} \\
\geq\ & \frac{1}{2} \log n
\end{aligned}$$

2 Optional exercises from HW2

2.1 HW3, Problem 7

1. For this part, greedy algorithm is enough. Take as much as quarters as possible, then dimes, nickels, and rest pennies. Actually this could take only $O(1)$ by using a division:

   $Q = n/25$
   $D = (n - 25 \times Q)/10$
   $N = (n - 25 \times Q - 10 \times D)/5$
   $P = n - 25 \times Q - 10 \times D - 5 \times N$

   To Prove that this algorithm is correct, consider n in 4 cases:

   (a) $n \geq 25$: We are taking as much quarters as we can. If the optimal solution takes less quarters, then it must contain some other coins with total value at least equal to 25. If there are 3 dimes, we can replace them with one quarter and one nickel; if there are at least 2 dimes and 1 nickel, we can replace them with one quarter; if there are some nickels and pennies, we can replace some of them with one quarter. In any case the replacement will bring less coins.

   Please be careful enough to notice that when n is greater than 25, there are still so many cases. You can't say that there should definitely be 2 dimes and 1 nickel.

   (b) n in $[10, 25)$: We are now taking as much dimes as we can. If the optimal takes less dimes, then it must contain some other coins with total value at least equal to 10. Replacing that combination will always bring less coins.

   (c) n in $[5, 10)$: We are now taking as much nickels as we can. If the optimal takes less nickels, then it must contain some other coins with total value at least equal to 5. Replacing that combination will always bring less coins.

   (d) n in $[0, 4)$: The only solution is using pennies.

2. Similar to (a), take as many c^k coins as possible, and then $c^k - 1$, and so on. To prove that this greedy algorithm is right, simply prove that except c^k coins, there can't be more than $(c^k - 1)$ coins of a kind (otherwise they would be a higher coin). Then prove that $(c-1)c^{k-1} + (c-1)c^{k-2} + ... + (c-1) < c^k$, which means there should be as many c^k coins as possible.

3. Any reasonable counterexample is acceptable. For example, with the coins set $\{1, 3, 4\}$, and the target value 6, greedy comes up with $\{4, 1, 1\}$, but $\{3, 3\}$ is a better choice.

4. Use dynamic programming.

```
Let $Cnt$ be a new array $[0...n]$
Let $Taken$ be a new array $[0...n]$
$Cnt[0] = 0$
    For $i = 1$ To $n$ Do
        $Cnt[i] = MAX$
        For $j = 1$ To $k$ Do
            If $Cnt[i] \geq Cnt[i - Coin[j]] + 1 Do$
            If $Cnt[i] = Cnt[i - Coin[j]]$
            $Taken[i] = j$
Return $Cnt[n]$
```

Method takes $O(nk)$ time. Use Taken to backtrack the coin we need.

2.2 HW3, Problem 8

1. Still use dynamic programming. Use array M to record the length of the sequence and S for backtracking. For every i from range 1 to n, find j from from range 1 to $i - 1$ such that $A[i] >= A[j]$ with maximum $M[j]$. Let $M[i] = M[j] + 1$ and $S[i] = j$. Find the maximum $M[i]$ and backtrack the sequence. As there are n elements in A and each take $O(n)$ time to compute, it totally costs $O(n^2)$.

2. For any subsequence, there are only two things that really matter in this problem: the ending element and the length. Use a list $L[1]$ which is initially empty to store the smallest ending element of any subsequence with length 1 and two arrays S and T to backtrack, where $S[i]$ means the last element of longest subsequence ending with $A[i]$ and $T[i]$ means the last element of subsequence of length i. During an iteration for $A[i]$, if $A[i]$ is greater than the greatest element in L (or if it's empty), add one more element $L[k + 1] = A[i]$ where k is the length of L before. And if $A[i]$ is less than or equal to the least element in L, update $L[1]$ with $A[i]$. Otherwise binary search for a j such that $L[j] < A[i] <= L[j + 1]$. Update $L[j + 1]$ with $A[i]$ and let $T[j + 1] = i, S[i] = T[j]$. Finally, use S and T to backtrack the subsequence. The time cost is $O(n \log n)$.

3 Optional exercises from HW4

3.1 HW4, Problem 6

1. Forward direction:
   If the root r has only one child, after deleting r, the rest of the graph is connected, so r is not an articulation point. So if the root r is an articulation point then r has at least two children.

   Backward direction:
   If the root r has at least two children, then there is no edge to connect these sub-trees, i.e. without r, the graph is disconnected. Hence r is an articulation point.

2. Forward direction:
   If v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v, it means that if we remove v, then node s and its descendants cannot reach any other node of the graph. Thus, the graph would be not connected and v is an articulation point.

   Backward direction:
   If v is a non-root articulation point of G, then after removing v from G, the graph is not connected. Thus, the proper ancestors of v are not connected to some descendants of v. Then there must be a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v.

3. Algorithm:
   We can use the following equation to compute v.low by starting at the leaves of the G_π.

   $$v.low = min(v.d, min(y.low), min(w.d))$$

   Where y is child of v and (v, w) is a backedge.

   Proof of Correctness:
   It's a dynamic programming problem. We can use optimal substructure to prove the correctness.

   Running Time Analysis:
   It has the same running time as DFS. Since it is a connected graph, it is $O(E)$.

4. Algorithm:
   We first run DFS and then the algorithm in part c). By part a), we can test if the root is articulation point. By part b), if v has a child s in G_π such that $s.low \leq v.d$, v is not an articulation point. Otherwise, it is an articulation point.

   Proof of Correctness:
   If v has a child s in G_π such that $s.low \leq v.d$, then s has a back edge to a proper ancestor of v and thus v cannot be an articulation point.

   Running Time Analysis:
   Both DFS and algorithm in part c) run in $O(E)$, therefore the algorithm run in $O(E)$.

5. Forward direction:
   If (u, v) lies on a simple cycle, then there is a cycle $u \to v \to x_1 \to \ldots \to x_n \to u$, such that all of u, v, xi are distinct. Any path that included the edge (u, v) can be modified to include the path $u \to x_n \to \ldots \to x_1 \to v$. Thus, (u, v) is not a bridge. So, if (u, v) is a bridge, then it is not on a simple cycle.

   Backward direction:
   Assume (u, v) is not on a simply cycle and it is not a bridge. If we remove the edge (u, v), there is still a path connecting u and v, $u \to x_n \to \ldots \to x_1 \to v$. Then, the edge (u, v) forms a simple cycle, which contradicts to the assumption. So (u, v) must be a bridge.

6. Algorithm:
   Apply the algorithm in part c). And then iterate all the edges in G_π, if an edge (u, v) satisfies $v.low = v.d$, then the edge (u, v) is a bridge.

   Proof of Correctness:
   Any bridge in the graph G must exist in the graph G_π. Otherwise, assume that (u, v) is a bridge and that we explore u first. Since removing (u, v) disconnects G, the only way to explore v is through the edge (u, v). So, we only need to consider the edges in G_π as bridges. If there are no simple cycles in the graph that contain the edge (u, v) and we explore u first, then we know that there are no back edges between v and anything else. Also, we know that anything in the subtree of v can only have back edges to other nodes in the subtree of v. Therefore, we will have $v.low = v.d$ since v is the first node visited in the subtree rooted at v.

Running Time Analysis:
Computing $v.low$ for all vertex v takes time $O(|E|)$ as we showed in part c). Looping over all the edges takes time $O(|E|)$. Thus the total time to compute the bridges in G is $O(|E|)$.

4 Optional problems from HW5

4.1 HW5, Problem 7

Value of max-flow= 11, min-cut=({s, a, b,c}, {d, t}).

4.2 HW5, Problem 8

1. Given a flow network G where S is the set of sources and T is the set of sinks, we construct a flow network G' as follows.

   (a) Add two new nodes, a super source s^* and a super sink t^*.

   (b) Add an edge (s^*, s) for every source $s \in S$ in G with capacity ∞.

   (c) Add an edge (t, t^*) for every sink $t \in T$ in G with capacity ∞.

   Clearly the transformation is efficient. We now claim that the value $|f'|$ of the max flow in G' is v if and only the value $|f|$ of the max flow in G is v. Note that $|f| = \sum_{s\in S} \sum_{(s,u)\in G} f(s, u)$ and $|f'| = \sum_{s\in S} f'(s^*, s)$.

   First, for every flow f in G we can obtain a flow f' in G' with value $|f'| = |f|$ by setting

   (a) $f(s^*, s) = \sum_{(s,u)\in G} f(s, u)$ for every $s \in S$;

   (b) $f(t, t^*) = \sum_{(u,t)\in G} f(u, t)$ for every $t \in T$; and

   (c) $f'(e) = f(e)$ for every other edge in G'.

   Next suppose that the max flow f' in G' has value $|f'| = v$. Let f be the flow in G obtained by setting $f(e) = f'(e)$ for all $e \in G$. By flow conservation in G', $|f| = |f'| = v$. Further f is the max flow in G. For suppose f_1 is the max flow in G. We can construct its corresponding flow f'_1 in G' as above without violating capacity constraints since $c(s^*, s) = \infty$. Then $|f'_1| = |f_1| > |f| = |f'|$, contradicting maximality of f'.

2. Given a flow network G where both the edges *and the vertices* have capacities, we construct a flow network G' as follows.

   (a) Replace every vertex $u \in V - \{s, t\}$ by two vertices u_1, u_2.

   (b) All edges in G coming into u now enter u_1 in G'.

   (c) All edges in G going out of u now leave u_2 in G'.

   (d) Add an edge (u_1, u_2) in G' with capacity equal to the capacity $c(u)$ of vertex u.

The transformation is efficient: G' consists of $2|V|-2$ vertices and $|E|+|V|-2$ edges. We now claim that the max flow in G' has value v if and only the max flow in G has value v. First, given a flow f in G, we can construct a flow f' in G' such that $|f'| = |f|$ as follows:

(a) for all $(s, u_1) \in G'$ and $(x_2, t) \in G'$, we assign $f'(s, u_1) = f(s, u)$ and $f'(x_2, t) = f(x, t)$;

(b) for every $(x_2, u_1) \in G'$, we assign $f'(x_2, u_1) = f(x, u)$;

(c) for every $(u_1, u_2) \in G'$, we assign $f'(u_1, u_2) = \sum_{(x_2,u_1)\in G'} f'(x_2, u_1)$

Next suppose that the max flow f' in G' has value $|f'| = v$. Let f be the flow in G obtained by setting $f(s, u) = f'(s, u_1)$, $f(u, t) = f'(u_2, t)$ and $f(x, u) = f'(x_2, u_1)$; then f satisfies edge and node capacity constraints in G, and $|f| = |f'| = v$. Further f is the max flow in G. For suppose f_1 is the max flow in G. We can construct its corresponding flow f_1' in G' as above. Then f_1' satisfies $|f_1'| = |f_1| > |f| = |f'|$, contradicting maximality of f'.

5 Optional problem from HW6

5.1 HW6, Problem 7

For a subset S of the vertices including 1 and at least one other city, let $OPT(S, j)$ be the shortest path that starts at 1, visits all other vertices in S, and ends at j. The key observation is that the shortest path from 1 to j through all the vertices in S consists of some shortest path from 1 to a vertex i in $S - \{j\}$, and the additional edge from i to j. So we can generate the cost for every subproblem $OPT(S, j)$ as follows:

$$OPT(S, j) = \min_{i \in S, i \neq j} \{OPT(S - \{j\}, i) + d_{ij}\}.$$

For all j, we initialize $OPT(\{1, j\}, j) = d_{1j}$.

The cost of the optimal tour is given by

$$\min_{j \in V, j \neq 1} \{OPT(V, j) + d_{j1}].$$

Intuitively, we build up paths one node at a time, not caring (at least for now) where they will end up. Once we have paths that go through all the vertices, it is easy to check the tours: they consist of a shortest path through all the vertices followed by an additional edge.

The algorithm requires $O(n^2 2^n)$ time as there are $O(n 2^n)$ entries in the dynamic programming table and each requires $O(n)$ time to fill in.