Solutions to Problem Set 1

1. $O(\cdot)$ **Notation**

   (a) Give the best (slowest growing) big-Oh bound for $f(n) = \sum_{k=1}^{n} k^r$, where $r > 0$ is a fixed constant.

   Solution: $O(n^{r+1})$. Since each term in the sum is at most n^r, we have $f(n) \leq n^{r+1}$. This is the best possible bound, because the last $n/2$ terms of the sum are each at least $n^r/2^r$, so we have $f(n) \geq n^{r+1}/2^{r+1}$.

   (b) Which of the following statements is true or false?

$$n^2 + 4n \log n = O(n^2) \quad (1)$$
$$2^n = O(n^2) \quad (2)$$
$$\log n = O(n) \quad (3)$$
$$n^3 + 3n^2 = O(n^2) \quad (4)$$

   Solution:

$$n^2 + 4n \log n = O(n^2) \ \text{ True}$$
$$2^n = O(n^2) \ \text{ False}$$
$$\log n = O(n) \ \text{ True}$$
$$n^3 + 3n^2 = O(n^2) \ \text{ False}$$

2. **Recurrence relations**

   Solve the following recurrence relations (c is a constant).

   (a) $T(n) = 5 \cdot T(\frac{n}{4}) + cn^2$
   (b) $T(n) = 3 \cdot T(\frac{n}{2}) + cn$
   (c) $T(n) = 27 \cdot T(\frac{n}{3}) + cn^3$
   (d) $T(n) = 2 \cdot T(\frac{n}{2}) + \sqrt{n}$
   (e) $T(n) = 3 \cdot T(\frac{n}{3}) + cn^2$

Solution:

(a) $T(n) = 5 \cdot T(\frac{n}{4}) + cn^2$: $T(n) = O(n^2)$
(b) $T(n) = 3 \cdot T(\frac{n}{2}) + cn$: $T(n) = O(n^{\log_2 3})$
(c) $T(n) = 27 \cdot T(\frac{n}{3}) + cn^3$: $T(n) = O(n^3 \log n)$
(d) $T(n) = 2 \cdot T(\frac{n}{2}) + \sqrt{n}$: $T(n) = O(n)$
(e) $T(n) = 3 \cdot T(\frac{n}{3}) + cn^2$: $T(n) = O(n^2)$

3. **Divide and Conquer** Given a sorted array $A[1], \dots, A[n]$ containing distinct integer values, design and analyse an $O(\log n)$ time algorithm that finds an index i such that $A[i] = i$, if such an index i exists.

Solution: The main idea is that, since the array contains distinct integers, we always have $A[i+1] \geq A[i] + 1$. This means that if $A[i] > i$, then $A[i+1] > i + 1$, and so it is not possible to have $A[j] = j$ for any $j \geq i$. Similarly, if $A[i] < i$, then we must have $A[j] < j$ for every $j \leq i$. This leads to the following algorithm

```
def fixed-point(A,n):
    first=1
    last=n
    while first ≤ last:
        middle = ⌈(first+last)/2⌉
        if A[middle] = middle]: return middle
        else if A[middle] > middle: last= middle − 1
        else first=middle + 1
    return ⊥
```

The algorithm maintains the invariant that an index i such that $A[i] = i$, if it exists, satisfies $first \leq i \leq last$. When we have $first > last$ we correctly conclude that such an index does not exist, and we return the error symbol $\perp$. If we find an index $middle$ such that $A[middle] = middle$ we correctly return it. The algorithm converges because the range of indices from $first$ to $last$ decreases at each iteration. The algorithm performs $O(1)$ work in each iteration of the while loop. In each iteration, the range $A[first], \dots, A[last]$ being explored decreases by a factor of 2, so the number of iterations is $O(\log n)$ and the total time is $O(\log n)$.

4. **Strongly Connected Components**

One of the following statements is true. Say which one and prove it.

(a) If a directed graph G has k strongly connected components, by adding one more edge to G the number of strongly connected components can drop at most by 1 (i.e. the new graph obtained from G by adding one edge has at least $k - 1$ strongly connected components).

(b) For every k, there exists a graph G that has k strongly connected components and such that if we add one particular edge to G, we can make it be strongly connected (i.e. the new graph has only 1 strongly connected component).

Solution: The second statement is correct. An example is a path with k vertices $v_1, \ldots, v_k$ and edges $(v_i, v_{i+1}$ for $i = 1, \ldots, k-1)$. Such a graph is acyclic and so each of the k vertices is a strongly connected component. Adding the edge (v_k, v_1) turns the graph into a cycle, which is strongly connected.

5. **Minimum Spanning Tree**

Prove that the following algorithm for the minimum spanning tree problem is correct, or show an example of a graph where the algorithm fails. In either case, discuss how to efficiently implement the algorithm, and what is the resulting running time. Assume the graph is represented with adjacency lists.

```
Algorithm A(G=(V,E): graph, w: weights)
    sort the edges of G into non-increasing order of weight
    T = E
    for all e ∈ E in non-increasing order of weight do
        if T − {e} is connected then T = T − {e}
    return T
```

Solution: There are a few possible approaches to prove correctness.

We can prove by induction on k that, after the algorithm has deleted k edges, there is an optimal solution which is a subset of the residual edges.

This is true when 0 edges are removed. If this is true when k edges are removed, call G_k the graph at that point, and call T an optimal solution which is a subset of the edges of G_k.

Let (u, v) be the $(k+1)$-th edge to be removed and call G_{k+1} the graph obtained from G_k by deleting (u, v). We need to prove that there is an optimal solution T' that uses a subset of the edges of G_{k+1}. If T does not use (u, v) then we are done. If T uses (u, v), then remove (u, v) from T. This splits the vertices into two connected components, call them A and B.

We claim that, of the edges considered after the k-th removed one and before (u, v), none of them go between A and B. Indeed, if there was an edge (a, b) in G_k such that $a \in A$, $b \in B$ and such that (a, b) comes before (u, v) in the ordering, then removing (a, b) from G_k would not disconnect G_k, because the edges of T (which is a subset of G_k) suffice to connect the vertices within A and the vertices within B, and (u, v) goes between A and B. Thus, (a, b) would have been the $(k+1)$-st edge to be removed instead of (u, v) and we have a contradiction.

Furthermore, G_k must contain an edge (a, b) such that $a \in A$ and $b \in B$, otherwise the removal of (u, v) would disconnect G_k, and we would not have removed (u, v) from G_k.

In conclusion, there is an edge (a, b) in G_k such that the cost of (a, b) is $\leq$ than the cost of (u, v) and such that $a \in A$ and $b \in B$. Add (a, b) to T to reconnect it, and obtain a new tree T'. The new tree is a subset of G_{k+1}, it is at least as good as T, and hence optimal, and we have proved the inductive step.

Edges can be sorted in $O(|E| \log |E|)$ time, and each of the $|E|$ steps can be performed in $O(|V| + |E|)$ time, leading to a running time of $O(|E|^2)$.