Homework #8

Analysis of Algorithms

Illya Starikov

Due Date: November 27th, 2017

Contents

Listings

Question #1

Let the samples space $S = \{X_{\text{free}} \implies$ prisoner X is going free, $Y_{\text{free}} \implies$ prisoner Y is going free, Z_{fr} prisoner Z is going free, $\}$. Furthermore, let y be event that Y is to be executed. Using Bayes Theorem, we get the following:

$$\begin{aligned}
\Pr(X_{\text{free}}|y) &= \frac{\Pr(y|X_{\text{free}}) \times \Pr(X_{\text{free}})}{\Pr(y \cap X_{\text{free}}) + \Pr(y \cap Y_{\text{free}}) + \Pr(y \cap Z_{\text{free}})} \\
&= \frac{\frac{1}{3} \times \frac{1}{2}}{\frac{1}{3} \times \frac{1}{2} + 0 + \frac{1}{3} \times 1} \\
&= \frac{1}{3}
\end{aligned}$$

We see that the prisoner X still has a probability of $\frac{1}{3}$.

Question #2

$$\text{minimum} = 2^h \qquad \text{maximum} = 2^{h+1} - 1$$

Question #3

A simple way to achieve an $\mathcal{O}(n \lg k)$ algorithm.

1. Sort all of the arrays in ascending order.
2. Create a minimum heap with all of the minimum elements in the k individual lists. Remove these elements from the respective arrays.
3. Remove the minimum element (name it d_{min}) from the heap mentioned in Step 2, and replace it with the minimum from it's respective list. Add d_{min} to the solution.
4. While the heap is not empty, repeat Step 3.

Listing 1: Problem #8.2

```
#!/usr/local/bin/python3
#
# problem-3.py
# source
#
# Created by Illya Starikov on 11/16/17.
# Copyright 2017. Illya Starikov. All rights reserved.
#

from collections import defaultdict


class MinHeap():
    """A Min-Max Heap Implementation"""
    __heap_list = []
    __current_size = 0

    def __init__(self):
        self.__heaplist = [0]
        self.__current_size = 0

    def heapify(self, list_):
        i = len(list_) // 2
        self.__current_size = len(list_)
        self.__heap_list = [0] + list_[:]
        while (i > 0):
            self.percolate_down(i)
            i = i - 1

    def percolate_down(self, i):
        while (i * 2) <= self.__current_size:
            minimum_child = self.min_child(i)
            if self.__heap_list[i] > self.__heap_list[minimum_child]:
                tmp = self.__heap_list[i]
                self.__heap_list[i] = self.__heap_list[minimum_child]
```

```
36                  self.__heap_list[minimum_child] = tmp
37              i = minimum_child
38
39      def percolate_up(self, i):
40          while i // 2 > 0:
41              if self.__heap_list[i] < self.__heap_list[i // 2]:
42                  tmp = self.__heap_list[i // 2]
43                  self.__heap_list[i // 2] = self.__heap_list[i]
44                  self.__heap_list[i] = tmp
45              i = i // 2
46
47      def insert(self, k):
48          self.__heap_list.append(k)
49          self.__current_size = self.__current_size + 1
50          self.percolate_up(self.__current_size)
51
52      def min_child(self, i):
53          if i * 2 + 1 > self.__current_size:
54              return i * 2
55          else:
56              if self.__heap_list[i*2] < self.__heap_list[i*2+1]:
57                  return i * 2
58              else:
59                  return i * 2 + 1
60
61      def delete_min(self):
62          to_return = self.__heap_list[1]
63
64          self.__heap_list[1] = self.__heap_list[self.__current_size]
65          self.__current_size = self.__current_size - 1
66          self.__heap_list.pop()
67          self.percolate_down(1)
68
69          return to_return
70
71      def print_heap(self):
72          print(self.__heap_list)
73
74      def __len__(self):
75          return self.__current_size
76
77  def empty_tree(input_list):
78      """Recursively iterate through values in nested lists."""
79      for item in input_list:
80          if not isinstance(item, list) or not empty_tree(item):
81               return False
82      return True
83
84  def k_way_merge(enumerables):
85      heap_values = defaultdict(list)
86      heap = MinHeap()
87      solution = []
```

```
88
89      for list_ in enumerables:
90          list_ = sorted(list_)
91          min_value = list_.pop(0)
92
93          heap_values[min_value].append(list_)
94
95      heap.heapify(list(heap_values.keys()))
96
97      while len(heap) > 0:
98          minimum = heap.delete_min()
99          solution.append(minimum)
100
101         minimum_list = [] if heap_values[minimum] == [] else
        heap_values[minimum][-1]
102         del heap_values[minimum]
103
104         if minimum_list != []:
105             new_minimum = minimum_list.pop(0)
106             heap_values[new_minimum].append(minimum_list)
107
108             heap.insert(new_minimum)
109
110     return solution
111
112 def main():
113     lists = [[3, 2, 1, 0], [5, 4], [9, 8, 7, 6]]
114     print(k_way_merge(lists))
115
116
117 if __name__ == "__main__":
118     main()
```

Question #4

Recall Stirling's Approximation for factorials:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \tag{1}$$

We skip the $\frac{1}{2}$ and $\frac{1}{n}$ case; for if we can't prove $\frac{1}{2^n} \times n! \in \Omega(n)$, then neither are $\frac{1}{2}$ and $\frac{1}{n}$.

Theorem 1.

$$\forall n \in \mathbb{R}^+, \frac{1}{2^n} \times n! \notin \Omega(n)$$

Proof. Suppose not. That is, suppose $\exists c \in \mathbb{R}, \frac{1}{2^n} \times n! \leq c\,n$. Therefore, $\frac{1}{2^n} \times n!$ must be in the same growth class as n. By definition, the limit of the two functions must be convergent to some number M.

Because we cannot directly take the limit, we use Equation 1. From this we get the result

$$\lim_{n\to\infty} \frac{\frac{1}{2^n} \times n!}{c\,n} = \frac{\frac{1}{2^n} \times \sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{c\,n} = \infty \qquad \forall c \in R^+$$

This has led us to a contradiction. □

Question #5

Proof. Suppose we have n integers of length L (we can make this assumption because we can always pad the leading digits with 0s).

Base Case Solving with $L = 1$ is trivial. Because it assumed our sorting algorithm is correct, this will simply sort the digits.

Inductive Hypothesis Suppose that it is true for $n - 1$; we will show it to be true for n.

When comparing the nth digit, call it d_1 and d_2 from the two respective lists, it appears something like such:

$\cdots$	$\cdots$	d_1	$\cdots$	$\cdots$
$\cdots$	$\cdots$	d_2	$\cdots$	$\cdots$

From this, there are three cases:

$d_1 < d_2$ Then d_1 is placed before d_2.

$d_1 > d_2$ Then d_1 is placed after d_1.

$d_1 = d_2$ Then the numbers go unchanged. **Because our sorting algorithm is stable**, the order is preserved from the lower digits. This is shown below.

$\cdots$	$\cdots$	42	64	$\cdots$
$\cdots$	$\cdots$	42	16	$\cdots$

□

Question #6

We can accomplish $n + \lg n - 2$ runtime as such:

1. Split all elements into pairwise elements ($S \implies (s_1, s_2), (s_3, s_4), \ldots, (s_{n-1}, s_n)$).
2. Compare all of the elements with respect to the smallest element in the pair. This will get the smallest. This requires n iterations.
3. The second smallest value must have been an element that lost to the original minimum, so we cache all of the lost contenders. We then run the previous step on this to get the second minimum. This requires at most $\lg n - 1$ iterations.

We first the following sample input $n = 10$:

342	133	40	365	796	716	354	38	256	614

We get the following output:

$$\begin{aligned} \text{Number of Comparison} &\in \Omega\left(n + \lceil \lg n \rceil - 2\right) \\ &\leq 10 + \lg 10 - 2 \\ &\leq 12 \\ &= 7 \end{aligned}$$

Listing 2: Problem #5

```
#!/usr/local/bin/python3
#
# problem-5.py
# source
#
# Created by Illya Starikov on 11/24/17.
# Copyright 2017. Illya Starikov. All rights reserved.
#

from random import sample
from math import log2, ceil


def static_vars(**kwargs):
    def decorate(func):
        for k in kwargs:
            setattr(func, k, kwargs[k])
        return func
    return decorate

```

```
21
22 def pairwise(iterable):
23     """s -> (s0, s1), (s2, s3), (s4, s5), ..."""
24
25     a = iter(iterable)
26     return list(zip(a, a))
27
28
29 @static_vars(comparison_counter=0)
30 def find_smallest_value(iterable, pairwised=False):
31
32     if iterable != []:
33         if pairwised is False:
34             iterabale_pairwise = pairwise(iterable)
35         else:
36             iterabale_pairwise = iterable
37
38         minimum = iterabale_pairwise.pop(0)
39
40         for element in iterabale_pairwise:
41             find_smallest_value.comparison_counter += 1
42
43             if min(element) < min(minimum):
44                 minimum = element
45
46         return min(minimum), max(minimum), [x for x in iterable if x
    not in minimum], find_smallest_value.comparison_counter
47
48     return None, None, None, None
49
50
51 def find_second_smallest_element(iterable):
52     """Returns second smallest of two inputs"""
53     smallest, pair_of_smallest, new_iterable, _ = find_smallest_value(
    iterable)
54     smallest_second_iteration, _, _, comparison_counter =
    find_smallest_value(new_iterable)
55
56     second_smallest = pair_of_smallest if smallest_second_iteration is
    None else min(pair_of_smallest, smallest_second_iteration)
57     return second_smallest, comparison_counter
58
59
60 def generate_input(of_length):
61     """Generates random input for problem #7. Returns (iterable,
    second_smallest_element)"""
62     input_ = sample(range(of_length * 100), of_length)
63
64     smallest = min(input_)
65     second_smallest = min([x for x in input_ if smallest != x])
66
67     return input_, second_smallest
```

```
68
69
70  def main():
71      length_of_input = 10
72
73      input_iterable, second_smallest_from_input = generate_input(
        length_of_input)
74      second_smallest, comparisons = find_second_smallest_element(
        input_iterable)
75      print("Input: {}".format(input_iterable))
76
77      print("Length: {}\nElements {} = {} => {}\nNumber of Comparisons
        ({}) <= n + lg(n) - 2 ({}) => {}".format(
78          length_of_input,
79          second_smallest_from_input, second_smallest,
        second_smallest_from_input == second_smallest,
80          comparisons, length_of_input + ceil(log2(length_of_input)) - 2,
         (comparisons <= length_of_input + ceil(log2(length_of_input)) - 2)
81          ))
82
83
84  if __name__ == "__main__":
85      main()
```

Question #7

Theorem 2. *The Set-Partition Problem is NP-Complete.*

Proof. It is trivial to prove that The Set-Partition problem is NP. Given two candidate solutions A and $\bar{A}$, verify that following things:

$$\sum_{x \in A} x - \sum_{x \in \bar{A}} x = 0 \quad \text{and} \quad A \cup \bar{A} = S$$

Recall The Subset Sum problem as follows:

> Given a set of natural numbers S and a natural number t, is there a subset of S that sums to t

We will prove that The Set-Partition problem reduces down to The Subset Sum problem.

Suppose we have $p, t \in \mathbb{R}$ and a set P, where t is a particular number and p is $p = \sum_{x \in P} x$. We give input to Set-Partition $P^* = P \cup s - 2t$. From this, we have two cases.

1. If there exists $t \in P$, then remaining numbers in P sum to $s - t$. This satisfies the Set-Partition problem.
2. There exists a partition of X^* into two sets Q, Q^* such that the $\sum_{x \in Q} = \sum_{x \in Q^*} = s - t$. This implies $s - 2t \in Q * \cup Q$. Removing this number, one set sums to t, and we can use the previous case.

Because we have proved The Set-Partition Problem is NP and it reduces to an NP-Complete problem, we conclude that the Set-Partition problem is NP Complete. □

Question #8

Problem #8.1

There exists an algorithm. Because there are only two denominations, we can enumerate all possible solutions. By producing pairs of all values in amount of x and the amount of y, we get a solution that's $\mathcal{O}\left(\text{amount of x} \times \text{amount of y}\right) = \mathcal{O}\left(n^2\right)$.

Listing 3: Problem #8.1

```
#!/usr/local/bin/python3
#
# problem-8-1.py
# source
#
# Created by Illya Starikov on 11/26/17.
# Copyright 2017. Illya Starikov. All rights reserved.
#


def all_combinations(list_one, list_two):
    """Returns all two-element combinations of both lists"""

    all_combinations = []
    for element_one in list_one:
        for element_two in list_two:
            all_combinations += [[element_one, element_two]]

    return all_combinations


def bonnie_and_clyde(x_worth, y_worth, x_amount, y_amount):
    """Literally a brute force solution"""
    all_combinations_of_x_y = all_combinations(range(x_amount), range(y_amount))

    for candidate_x, candidate_y in all_combinations_of_x_y:
        if (candidate_x * x_worth + candidate_y * y_worth) == (x_worth * (x_amount - candidate_x) + y_worth * (y_amount - candidate_y)):
            return (True, candidate_x, candidate_y)

    return (False, None, None)


def main():
    # Impossible Case
    x_worth, y_worth = 100, 7
    x_amount, y_amount = 1, 2

    print(bonnie_and_clyde(x_worth, y_worth, x_amount, y_amount))

    # Possible Case (Literally one of each)
```

```
    x_worth, y_worth = 100, 7
    x_amount, y_amount = 2, 2

    print(bonnie_and_clyde(x_worth, y_worth, x_amount, y_amount))

if __name__ == "__main__":
    main()
```

Problem #8.2

There exists an algorithm. And it works like such:

1. Sort the coins in ascending order.
2. Pop off the largest coin, give it to Bonnie.
3. Keep taking smallest coins and assign them to Clyde. If finished, and still have coins to assign, go to Step 2.
4. If at any there is not enough coins to match Bonnie when assigning to Clyde in Step 3, there exists no such configuration.

Listing 4: Problem #8.2

```
#!/usr/local/bin/python3
#
# problem-8-2.py
# source
#
# Created by Illya Starikov on 11/26/17.
# Copyright 2017. Illya Starikov. All rights reserved.
#


def bonnie_and_clyde(coins):
    coins_sorted = sorted(coins)

    bonnie_coins, clyde_coins = [], []

    while coins_sorted != []:
        bonnie_coins += [coins_sorted.pop()]

        while sum(bonnie_coins) != sum(clyde_coins):
            if not coins_sorted:
                return [None, None]

            clyde_coins += [coins_sorted.pop(0)]

    return [bonnie_coins, clyde_coins]


def main():
    # possible configuration
    coins = [1, 1, 2, 4, 8]
    print(bonnie_and_clyde(coins))

    # impossible configuration
    coins = [1, 2, 4, 64]
    print(bonnie_and_clyde(coins))

```

```
37
38 if __name__ == "__main__":
39     main()
```

Problem #8.3

There exists no polynomial algorithm.

Proof. Note to Reader: We shall call this the Bonnie-Clyde Check problem, and the solution set of checks S_{Bonnie} for Bonnie and S_{Clyde} for Clyde.

It is trivial to prove that this Bonnie-Cylde check problem is NP. Given a solution S_{Bonnie} and S_{Clyde}, we must verify

$$\sum_{x \in S_{\text{Bonnie}}} x\text{'s worth} = \sum_{x \in S_{\text{Cylde}}} x\text{'s worth} \qquad |S_{\text{Bonnie}}| + |S_{\text{Clyde}}| = n$$

where $|A|$ is the cardinality of A[1]. This is clearly a polynomial algorithm.

We will now show that the Bonnie-Clyde Check problem reduces down to the Partition Problem.

Assign every checks value as it's representation in the input set S. Feed S into The Partition problem. All output from The Partition problem is also valid output for the Bonnie Clyde problem.

Because we have proved that the Bonnie-Clyde problem is NP and reduces to an NP Complete problem, we conclude that the Bonnie-Clyde problem is NP complete. □

[1] Cardinality meaning the amount of elements in the set

Problem #8.4

The problem is also NP Complete.

Proof. Note to Reader: We shall call this the Bonnie-Clyde 100 Check problem, and the solution set of checks S_{Bonnie} for Bonnie and S_{Clyde} for Clyde.

If the minimum check value is greater than 100$, the proof is identical to the previous.

If the minimum check is less than 100$, first we must prove it is in NP. Given a solution S_{Bonnie} and S_{Clyde}, we must verify

$$\left| \sum_{x \in S_{\text{Bonnie}}} x\text{'s worth} - \sum_{x \in S_{\text{Cylde}}} x\text{'s worth} \right| = 100 \qquad |S_{\text{Bonnie}}| + |S_{\text{Clyde}}| = n$$

where $|A|$ is the cardinality of A[2]. This is clearly a polynomial algorithm.

We will now show that the Bonnie-Clyde 100 Check problem reduces to the partition problem.

If there is a valid solution via Partition problem, we know there to be a solution. If not, we append a 1 to the distribution of checks, and rerun Set-Partition. Does this until either:

- We reach a valid solution.
- We reach 100.

Either way, this problem reduces to the Partition Problem.

Because we have proved that the Bonnie-Clyde problem is NP and reduces to an NP Complete problem, we conclude that the Bonnie-Clyde problem is NP complete. □

[2]Cardinality meaning the amount of elements in the set

Question #9

A greedy algorithm is as follows:

1. Pick two arbitrary vertices u and v.
2. Add both u and v to the vertex cover.
3. Delete all incident edges to u and v.
4. If the adjacency matrix still has edges, go to Step 1.

For the adjacency matrix

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

We get the minimum vertex cover:

$$\{1, 2, 3, 4, 5\}$$

Listing 5: Problem #9

```
#!/usr/local/bin/python3
#
# problem-9.py
# source
#
# Created by Illya Starikov on 11/27/17.
# Copyright 2017. Illya Starikov. All rights reserved.
#

import random


def vertex_cover(adjacency_matrix):
    if sum([item for sublist in adjacency_matrix for item in sublist])
    != 0:
        random_verties = random.sample(range(len(adjacency_matrix[0])),
     2)

        for i in range(len(adjacency_matrix[0])):
            adjacency_matrix[i][random_verties[0]] = 0
            adjacency_matrix[random_verties[0]][i] = 0

            adjacency_matrix[i][random_verties[1]] = 0
            adjacency_matrix[random_verties[1]][i] = 0

```

```
24          return vertex_cover(adjacency_matrix).union(set(random_verties)
        )
25
26      else:
27          return set()
28
29
30  def main():
31      matrix = [[0, 0, 1, 0, 1],
32                [1, 0, 0, 1, 0],
33                [0, 1, 0, 0, 1],
34                [1, 0, 1, 0, 0],
35                [1, 0, 0, 0, 1]]
36
37      print(vertex_cover(matrix))
38
39
40  if __name__ == "__main__":
41      main()
```