

DSC 190 - Final Exam

Due: Friday, March 19

Write your solutions to the following problems by either typing them up or handwriting them on another piece of paper. Unless otherwise noted by the problem's instructions, show your work or provide some justification for your answer.

 Unlike the homeworks, we ask that you please do not collaborate on this final exam. However, you may still come to office hours to ask for help.

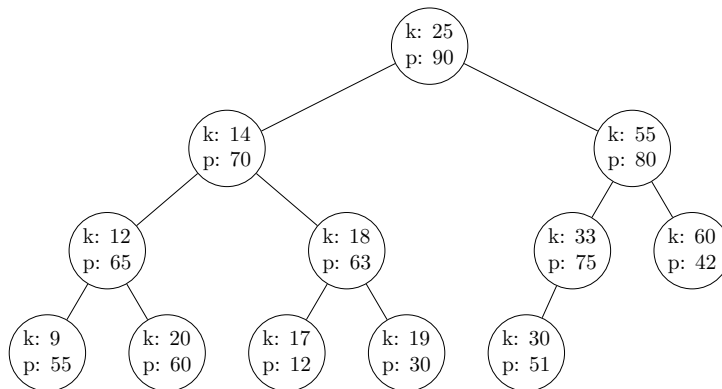
Problem 1.

For each part below, state the data structure that you would use in order to efficiently support the given operations, and give a short justification of *why* you would choose that data structure over the alternatives.

- a) You need a collection that supports inserting a new element and removing the largest element.
- b) You need a collection that supports inserting a new element, removing the largest element, and performing membership queries.
- c) You need a collection that supports membership queries and insertions.
- d) You need a collection that supports membership queries and insertions on *massive* data sets, and you'll tolerate some amount of false positives.
- e) You need to perform (approximate) nearest neighbor queries on high dimensional data.

Problem 2.

- a) Is this a valid treap? Why or why not?



- b) Recall that a randomized binary search tree is a treap where each node's priority is chosen randomly at the time of its insertion. This helps ensure that the treap is roughly balanced with high probability. Instead of using a random number, we might instead use the time of insertion as a numerical timestamp as the node priority. Is this a good idea? Why or why not?

Problem 3.

For each problem below, describe a greedy algorithm to solve it in polynomial time. State the algorithm's asymptotic time complexity.

- a) You are a mail carrier in a one-dimensional universe. You are given a collection of points, $\{p_1, p_2, \dots, p_n\}$ on a line which represent houses you must deliver mail to. Problem: choose an order in which to visit the points so as to minimize the total distance you travel. You can choose to start wherever you'd like.
- b) You need to run a long-running compute job that will take h hours. You have the choice of n servers to run the job on. Each server has a cost per hour and a maximum time that you can use it, but you're allowed to move the job from one server to another for free. You are billed for the exact amount of time you use: for instance, if a server costs 10 cents per hour and you use it for half an hour, you're billed 5 cents. Problem: choose which servers to use and for how long in order to minimize your costs.

Problem 4.

In a file called `bloom.py`, implement a class named `BloomFilter` with the following methods:

- `BloomFilter(k, c)`: create a Bloom filter using an underlying array of size c and k randomly-chosen hash functions.
- `.insert(x)`: insert an element x into the Bloom filter.
- `.query(x)`: return `True` if the Bloom filter believes the element to be in the collection; otherwise return `False`.

Hint: here is a function that randomly draws a hash function from a family of hash functions which map objects to the range $\{0, \dots, c - 1\}$:

```
import random

def generate_hash_function(c):
    mask = random.getrandbits(32)

    def h(x):
        return (hash(x) ^ mask) % c

    return h
```

To use it, first call `h_1 = generate_hash_function(c)`; now h_1 is your hash function. You can repeat this to generate multiple hash functions.

Problem 5.

Here is a common interview problem. You are given a set of `valid_words` and a string, `s`. Your goal is to count the maximum number of matches of a word in `valid_words` in `s` such that no two matches overlap.

For example, suppose

```
valid_words = {"banana", "an", "ana", "ice", "cream"}
```

and

```
s = "icedbanana".
```

One way to count non-overlapping matches is to see that `"ice"` and `"banana"` both appear once, for a total of two matches. Similarly, we can match `"ice"` and `"ana"` once each without overlaps (if we allowed overlaps, `"ana"` would match twice). But neither of these are optimal. The most non-overlapping counts can be achieved by matching `"an"` twice and `"ice"` once, for a total of three matches.

The code below implements an inefficient (exponential time) algorithm for counting the number of non-overlapping matches:

```
def count_non_overlapping(s, valid_words, start=0):
    """Count number of non-overlapping matches.

    Parameters
    -----
    s : str
        The string to find matches in.
    valid_words : Set[str]
        A set of strings that should be considered valid words.
    start : int
        An index into `s`. The substring s[start:] will be searched.

    Returns
    -----
    count
        Maximum number of non-overlapping matches.

    """
    best = 0
    for stop in range(start+1, len(s) + 1):
        is_word = int(s[start:stop] in valid_words)
        n = is_word + count_non_overlapping(s, valid_words, stop)
        best = max(best, n)
    return best
```

In a file named `count_non_overlapping.py`, create a function

```
count_non_overlapping_td(s, valid_words, start)
```

which implements an top-down dynamic programming solution for this problem. Your code should be a modification of the code above, and it should run in time polynomial in the length of the list.