# CS:3330 Homework 5, Fall 2015

## Due in class on Tue, Dec 1

---

1. Let $G$ be a directed, edge-weighted graph such that every edge has a weight that belongs to the set $\{0, 1, \ldots, W\}$, where $W$ is a non-negative integer.

   (a) Carefully describe a modified implementation of Dijkstra's algorithm so that the SSSP problem can be solved in $O(n \cdot W + m)$ time for this type of graphs. Here, as usual, the input graph has $n$ vertices and $m$ edges.

   (b) Separately argue that the worst case running time of your algorithm is indeed $O(n \cdot W + m)$.

   **Note:** This is a problem that appears in the practice problem set on Dijkstra's Shortest Path algorithm, that I posted just before Exam 2. You may also recall that I sketched out this algorithm in class during review for Exam 2.

2. Problem 20 from Chapter 4 (Pages 199-200).

3. Problem 21 from Chapter 4 (Page 200).

4. An *extendable array* is a data structure that stores a sequence of items and supports the following operations.

   - `AddToFront`($x$) adds $x$ to the beginning of the sequence.
   - `AddToEnd`($x$) adds $x$ to the end of the sequence.
   - `Lookup`($k$) returns the $k$-th item in the sequence, or `Null` if the current length of the sequence is less than $k$.

   Describe a simple data structure that implements an extendable array. Your `AddToFront` and `AddToBack` algorithms should take $O(1)$ *amortized* time, and your `Lookup` algorithm should take $O(1)$ worst-case time. The data structure should use $O(n)$ space, where $n$ is the current length of the sequence.

   **Notes:** Programming languages such as Python provide an extendable array data structure (e.g., called `list` in Python). There are "hidden" running time costs to such a data structure that beginning programmers are usually unaware off. This problem make some of these costs explicit. The typical implementation of the extendable array data structure uses fixed-sized arrays. During initialization, a fixed-sized array of size, say 100, is constructed. As items are added to the array, it may become full and we need to seek a new, larger, fixed-sized array and copy elements from the old array to the new (larger) array. Thus, some calls to the `AddToFront` and `AddToEnd` operations can be quite costly. Hence, the problem does not expect you to show that the *worst case* running time of these operationsis $O(1)$, only that the amortized running time is $O(1)$. You may recall from our *amortized analysis* of the *disjoint set union-find* data structure, it may be possible to "charge" the cost of a costly operation to elements of the data structure and then argue that each element is assigned $O(1)$ charge, over the course of many operations. This is the approach you should attempt to use for this problem. One final hint I want to provide is that when the array becomes full and it is time to seek a new array, you don't want to seek a new array that is just slightly larger. If you do so, the new array will also quickly fill up and you'll end up seeking a new array and copying elements very soon.

5. Problem 1 in Chapter 5.

6. In class (and in Chapter 5), we discussed the solutions of recurrences of the form $T(n) \leq q \cdot T(n/2) + c \cdot n$, when $n > 2$ and $T(n) \leq 2$ for $n \leq 2$. It is easy to extend this approach to situations in which the subproblem size is $n/3$, $n/4$, etc., instead of $n/2$. Use these ideas to solve the recurrences:

(a) $T(n) \le 7 \cdot T(n/5) + c \cdot n$, when $n > 2$ and $T(n) \le c$ for $n \le 2$.

(b) $T(n) \le 7 \cdot T(n/10) + c \cdot n \log_2 n$, when $n > 2$ and $T(n) \le c$ for $n \le 2$.

7. Your friend tells you that she's come up with a simple, modification to *binary search* whose running time is modeled by the recurrence: $T(n) \le T(\sqrt{n}) + 10$ for $n > 2$ and $T(n) \le 10$ for $n \le 2$. Based on the running time of your friend's modified binary search, should you start using her version of binary search, instead of your binary search? Carefully justify your answer.

---