CS 1510 Midterm 1
Fall 2015

1. (40 points) The setting for this problem is storage system with a fast memory consisting of k pages and a slow memory consisting of n pages. At any time, the fast memory can hold copies of up to k of the pages in slow memory. The input consists of a sequence of pages from slow memory, think of these as being accesses to memory. If an accessed page is not in fast memory, then it must be swapped into fast memory, and if the fast memory was full, some page must selected to be evicted from fast memory. The goal is to determine the pages to evict so as to minimize the total number of evictions.

   Consider for example that $k = 2$, $n = 4$ pages are named A, B, C and D, and the access sequence is A B C B A. Then after the first two pages, the fast memory contains A and B. When C is accessed then either A or B must be evicted. If A is evicted then no eviction is required when B is accessed, but either B or C must be evicted when A is accessed, and the total number of evictions is 2.

   Consider the following two algorithms:

   Algorithm Least Recently Used (LRU): Evict the page in fast memory who most recent access happened furthest in the past. So in the example above, when C is accessed, A would be evicted by LRU because of the two choices of what to evict, A was most recently accessed two time units in the past, while A was most recently accessed just one time unit in the past.

   Algorithm Furthest in the Future (FF): Evict the page in fast memory whose next access is furthest in the future. So in the example above, when C is accessed, A would be evicted by FF because A is next accessed two time units in the future, while B is accessed 1 time unit in the future.

   For each algorithm either prove or disprove that this algorithm correctly solves the problem. Your proof of correctness must use an exchange argument. I want to know whether you know how an exchange argument works, so for full credit you must give the full exchange argument, as well as explaining what the exchange is and why it is correct.

2. (20 points) The input to this problem consists of positive integers $x_1, \ldots, x_n$. Think of these as being the values of n jewels. Let $L = \sum_{i=1}^{n} x_i$, the sum of the x_i's. The problem is to partition the jewels into two sets A and B (so every jewel must be in exactly one of A and B) such that that the value of the jewels is set A is twice the value of the jewels in set B. If such a partition is not possible, the output should be "impossible".

   (a) Give pseudo-code implementing a dynamic-programming array-based algorithm to determine whether such a partition is possible. Your algorithm doesn't not need to find the partition. The running time should be $O(nL)$. Before the pseudo-code, you must define the meaning of each array entry. For example: "We define $A[i, j]$ to be the jth most valuable jewel among the first i jewels. Here is my pseudo-code to compute $A[i, j]$:".

   (b) Give pseudo-code to compute the actual partition from the array. Include a couple of sentences of English explanation about what your code is doing and why it works. The running time should be $O(nL)$.

3. (20 points) Consider the problem where the input is n integer keys $K_1, \ldots, K_n$, where $K_1 < K_2 < \ldots < K_n$, and n associated probabilities $p_1, \ldots, p_n$, where $\sum_{i=1}^{n} p_i = 1$. Here p_i is the probability that key K_i is accessed. The output should be the binary search tree T that minimizes the expected access time, which is the sum over the keys of the depth of the key in the tree times the probability that key is accessed, $\sum_{i=1}^{n}(\text{depth of } K_i \text{ in } T) * (p_i)$. Give an algorithm for this problem that runs in time $O(n^3)$. For 15 points, you can give an algorithm that computes the optimal expected access time, but not the actual tree.

4. (20 points) Consider a state consisting of n counties $C_1, \ldots, C_n$, with populations $P_1, \ldots P_n$. You have to partition the counties in k districts. But to avoid charges of gerrymandering, each district has to contain consecutive counties. So if a district contains county C_i and county C_j then it must contain each county C_k where $k \in [i, j]$. The objective is to minimum the difference between the county with the most aggregate people and the county with the least aggregate people.

   For example, if $n = 7$, $k = 3$ and $P_1 = 5$, $P_2 = 6$, $P_3 = 1$, $P_4 = 7$, $P_5 = 5$, $P_6 = 12$, and $P_7 = 20$. One feasible partition into districts is $\{C_1, C_2, C_3\}$, $\{C_4, C_5, C_6\}$, and $\{C_7\}$. The aggregate sizes of the districts would then be 12, 24 and 20. And the objective value would be $24 - 12 = 12$. This is clearly not an optimal solution.

   Give a dynamic programming algorithm to find the optimal objective value. Your algorithm need not compute the actual partition of the county into districts. The running time of your algorithm must be bounded by a polynomial in n. The running time should not depend on the size of the populations.