*Design and Analysis of Algorithms* March 22, 2012
Massachusetts Institute of Technology 6.046J/18.410J
Profs. Dana Moshkovitz and Bruce Tidor Handout 14

# Problem Set 5 Solutions

This problem set is due **at 9:00pm** on **Wednesday, March 21, 2012**.

---

**Problem 5-1. High Probability Bounds on Randomized Select**

Recall that in lecture, we discussed an algorithm for randomized select, for which you can find pseudocode in CLRS on page 216 in section 9.2. In this problem, you'll explore the problem of high probability bounds for randomized select.

We'll go through a few steps to disprove the following statement, which we will refer to as $(*)$:

Let $T(n)$ be the running time of RANDOMIZED-SELECT on an input of size $n$. Then there exist integers $n_0$, $c \geq 1$ such that for all $n \geq n_0$, $P(T(n) > cn) \leq \frac{1}{n}$.

**(a)** Let $b$ be a real number such that $1/2 < b < 1$, and let $A_i$ be the array in the $i^{th}$ recursion. Define a *bad pivot choice* in the $i^{th}$ recursion as one that results in $|A_{i+1}| > b|A_i|$. (Note that in class, we have been using $b = 9/10$.)
Give a lower bound on the probability of having $k$ bad pivot choices in a row. (The lower bound should hold for any input to RANDOMIZED-SELECT. Make it as tight as you can; you'll need it in part (c).)

**Solution:** A lower bound is $(1-b)^k$.
For most inputs, the probability of getting a single bad pivot will actually be $2(1-b)$, as there are two regions the pivot could be picked from that would leave an array of size greater than $b|A_i|$. However, in cases where the rank of the element being looked for is very small ($< (1-b)n$) or very large ($> bn$), one of these regions may not be applicable for finding bad pivots, and therefore when Select is called on these elements, the probability of getting a bad pivot is only $(1-b)$.

**(b)** If our initial array size is $n$, then after one bad pivot choice, the next array is of size at least $bn$. Recall that the running time at each recursive call requires time equal to at least the size of the array.
Give a precise lower bound (do not use big-$O$ notation) on the total running time of $k$ recursive calls to RANDOMIZED-SELECT, if in every recursive call we chose a bad pivot.

**Solution:** After $i$ recursive calls where each pivot chosen is a bad pivot, the smallest possible array size we could have at that point is $b^i n$. Because the running time at iteration $i$ is at least the size of the array $|A_i|$, we have a lower bound on the running time of $\sum_{i=0}^{k} |A_i| = \sum_{i=0}^{k} b_i n = n(\sum_{i=0}^{k} b^i) = n((1-b^{k+1})/(1-b))$.

**(c)** Use a proof by contradiction to disprove the statement $(*)$ above.

(Hint: Suppose there exists a $c$, $n_0$ satisfying the equation above. Then pick $b = 1 - \frac{1}{2c}$.)

**Solution:** As suggested in the hint, suppose there exists a $c$ and $n_0$ satisfying the equation above, and pick $b = 1 - \frac{1}{2c}$.

Now we will investigate the running time and probability of choosing $k$ bad pivots in a row from the start, where $k$ is some constant to be determined later. By part (b), the running time for the first $k$ recursive calls is at least $n((1 - b^{k+1})/(1 - b)) = 2cn(1 - b^{k+1})$.

Pick $k$ to be a constant large enough so that $b^{k+1} < 1/2$. Then $(1 - b^{k+1}) > 1/2$, so the total running time is greater than $cn$.

By part (a), this situation occurs with probability at least $(1 - b)^k$, which is a constant. Therefore we can pick an $n >> k$ (to ensure that the algorithm does not terminate after $k$ recursive calls) such that $n > n_0$ and $(1 - b)^k < 1/n$, contradicting the statement $(*)$ (which must hold for all $n > n_0$). Therefore, we have a contradiction, and no pair $c, n_0$ can exist satisfying the conditions.

### Problem 5-2. Random Vectors and Matrices

Random vectors are good choices for hashing and testing, because they are unlikely to be orthogonal to a given (non-zero) input vector. In this problem, we will perform all operations in $\mathbb{Z}_p$ for some prime number $p$ (which is to say we take the result of any arithmetic operation $\mod p$).

**(a)** You are given a non-zero vector $\vec{u} \in \mathbb{Z}_p^n$, and some number $c \in \mathbb{Z}_p$. Prove that if another vector $\vec{v} \in \mathbb{Z}_p^n$ has each element chosen independently and uniformly at random from $\mathbb{Z}_p$, then the probability that $\vec{v} \cdot \vec{u} = c$ is $1/p$.

(Hint: Observe that any non-zero vector $\vec{u}$ of size $n$ has $n - 1$ elements which can be arranged into a non-zero vector of size $n - 1$. Then use induction to prove the claim.)

**Solution:** First, we show the base case, for $v, u \in \mathbb{Z}_p$. The goal for the base case is to show that the probability that $vu = c$ is $1/p$ for any $c \in \mathbb{Z}_p$. Note that, because we know that $u$ is nonzero, it has a multiplicative inverse in $\mathbb{Z}_p$, so we can divide this whole equation by $u$. So we want to find the probability that $v = c/u$, where $c/u$ is some fixed number in $\mathbb{Z}_p$. Because $v$ is chosen uniformly at random from $\mathbb{Z}_p$, this probability is therefore $1/p$.

Now we show the inductive step. Assume the property is true for vectors of size $n-1$. Additionally, note that if $\vec{u}$ of size $n$ is non-zero, then it must have at least one non-zero element. If we take this element together with any other $n - 2$ elements of $\vec{u}$, then the resulting $\vec{u'}$ of size $n - 1$ is non-zero. So, without loss of generality assume that the non-zero element is at some index less than $n$.

Then

$$\Pr(\vec{v} \cdot \vec{u} = c) = \Pr(\sum_{i=1}^{n} v_i u_i = c) = \Pr(\sum_{i=1}^{n-1} v_i u_i = c - v_n u_n) = \Pr(\sum_{i=1}^{n-1} v_i u_i = c')$$

We can do the last step because the elements of $\vec{v}$ are independent. Now using the induction hypothesis, we can conclude that the probability is $1/p$.

In class, we have seen a couple examples of universal hash families. We will now devise another universal hash family based on random matrices.

**(b)** You are given two vectors $\vec{x}, \vec{y} \in \mathbb{Z}_p^n$ such that $\vec{x} \neq \vec{y}$. Using the result from part (a), show that if $\vec{v}$ is a random vector as before, then $\Pr(\vec{v} \cdot \vec{x} = \vec{v} \cdot \vec{y}) = 1/p$.

**Solution:**

$$\vec{v} \cdot \vec{x} = \vec{v} \cdot \vec{y} \iff \vec{v} \cdot (\vec{x} - \vec{y}) = 0$$

Using the result from part (a) with $\vec{u} = \vec{x} - \vec{y}$ and $c = 0$, this happens with probability $1/p$.

**(c)** Using the result from part (b), show that if $A$ is an $m \times n$ matrix with each element chosen independently and at random from $\mathbb{Z}_p$, then $\Pr(A\vec{x} = A\vec{y}) = 1/p^m$.

**Solution:** Note that $A\vec{x} = A\vec{y}$ holds if and only if, for every row $\vec{a_i}$ of $A$, it holds that $\vec{a_i} \cdot \vec{x} = \vec{a_i} \cdot \vec{y}$. For each $\vec{a_i}$, the probability that $\vec{a_i} \cdot \vec{x} = \vec{a_i} \cdot \vec{y}$ is $1/p$ by part (b). Because each of the $m$ rows of $A$ are independent, we can multiply these probabilities to get a total probability of $1/p^m$ that $A\vec{x} = A\vec{y}$.

**(d)** Conclude that the family $\mathcal{H}$ of all such functions $h_A(\vec{x}) = A\vec{x}$ where $A$ is an $m \times n$ matrix with elements in $\mathbb{Z}_p$, is universal.

**Solution:** Each function in $\mathcal{H}$ maps an input vector to a hash value that is a vector of size $m$ (in other words, a vector in $\mathbb{Z}_p^m$). The total number of possible hash values is therefore $p^m$. Choosing one of these functions at random results in us getting a random matrix $A$. Using the result from part (c), the probability that $h_A(\vec{x}) = h_A(\vec{y})$ for a given pair of vectors $\vec{x}$ and $\vec{y}$ that are not equal is $1/p^m$, fulfilling the definition of a universal hash family.

In the implementation of the encryption algorithm BitWhipper™, the key step is to perform the composition of two hashes to compute the vector $\vec{y} \in \mathbb{Z}_2^k$ such that $\vec{y} = h_B(h_A(\vec{x}))$, where $\vec{x} \in \mathbb{Z}_2^n$, $A$ is an $m \times n$ matrix, $B$ is a $k \times m$ matrix, and $h_A$ and $h_B$ are defined as in part (d) with $p = 2$.

Note that all operations here are performed in $\mathbb{Z}_2$, which means all additions are bit-wise XOR.

Ben Bitdiddle gives you a $k \times n$ matrix $C$ over $\mathbb{Z}_2$, which he says is equal to $B \cdot A$. If he's right, you could just use $h_C(\vec{x})$ in the implementation of BitWhipper™, in lieu of $h_B(h_A(\vec{x}))$. You want to make sure that $C = B \cdot A$, but you don't want to multiply the matrices because it will take you $O(kmn)$ time.

In part (e), you will come up with a randomized algorithm that runs much faster than $O(kmn)$ time, and also runs faster than any known algorithm for rectangular matrix multiplication.

**(e)** Using the result from part (a), devise a randomized algorithm to determine if $C = B \cdot A$. Show that your algorithm is correct with probability at least 90%.

**Solution:** Pick a random input vector $\vec{x}$ and compare the output of $C\vec{x}$ to $B(A\vec{x})$. If they are unequal, we conclude that the Ben's matrix is incorrect, otherwise we conclude it's correct.

The output of this algorithm will be incorrect if $C \neq B \cdot A$, but the two outputs generated are equal. To analyze the probability of this happening, suppose that $C \neq B \cdot A$ and consider $(C - B \cdot A)\vec{x}$, which is the product of a nonzero $k \times n$ matrix with a vector.

This nonzero matrix must have at least one nonzero row. Consider the dot product of this row with $\vec{x}$. Using (a), the probability that this product is equal to 0 is $1/p = 1/2$, which is an upper bound for when the algorithm will be wrong. If we repeat 4 times, the probability of all four tries being wrong is 1/16, meaning we have at least 15/16 probability of correctness, which is more than 90%.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012