CMPSC 623 Problem Set 2 and 3
Solutions to problems not discussed in class.
by Prof. Honggang Zhang

HW-2 Problem 8. (Probabilistic Analysis) Page 118, Problem 5-2. Only do part (a), (b), (c).

Part (a). Simply transfer the description into code. Need to keep an array for recording if a particular item is checked. Also need a counter to record the number of distinct elements that have been checked.

Part (b). Suppose we want to search for x whose index is i, that is, $A[i] = x$. We are doing a random search, so at each probe, any element is equally likely to be picked up. Then, the probability that we find the target x at the first pick or probe is simply $1/n$. The prob. that we find it at the second pick is $(n-1)/n * 1/n$ (this means, we are not successful at the first probe but successful at the second probe). Let X denote the number of probes until we find the target. Then it is easy to see that X is a geometric random variable. Then $E[X] = 1/(1/n) = n$.

Part (c). Element x now has k replicas in the total n elements. Again, we assume that each elem. is equally likely to be picked up at a probe. Then at the first probe, we can find x with probability k/n. Similar to Part (b), the probability of being successful at the second probe is $(n-k)/n * k/n$, so on so forth. X is now a geometric random variable with $p = k/n$. Then $E[X] = n/k$.

Hw-2 Problem 10. Page 160, Problem 7-2. Only do part (a), (b), (c), and (e). Note, you need to use substitution method for (e), and you can use the result in part (d) without proof.

We have discussed Part a,b,c in class.

Part (e) is a little tricky. Different prints of the 2nd edition of the textbook give you different problem descriptions and hints! So I didn't grade it. One hint is a tricky way to handle substitution method. I don't recall that I discussed it in class when I was discussing substitution method. So I didn't grade it. In any case, here is the idea:

For inductive step, you assume that $ET(n) \leq an\log n - bn, a > 0, b > 0, \forall n < k$. Then you want to prove that $ET(n) \leq an\log n - bn$.

$$
\begin{aligned}
ET(n) &= 2/n \sum_{q=2}^{n-1} ET(n) + \Theta(n) \qquad \text{(from part (c))} && (1)\\
&\leq 2/n \sum (aq\log q - bq) + \Theta(n) \qquad \text{(from our inductive hypothesis)} && (2)\\
&\leq 2/n(1/2an^2\log n - a/8n^2 - b(n-1)n/2) + \Theta(n) \qquad \text{(from part (d))} && (3)\\
&\leq an\log n - bn + b + \Theta(n) && (4)\\
&\leq an\log n - bn - (a/4n - cn - b) && (5)
\end{aligned}
$$

In the above, we let $\Theta(n) = cn$. Then, one can find $a \geq 4c + 4b/n$. One also needs to check base case.

HW-3 Problem 5. Problem 7-4 on page 162.

a). discussed in class.

b).

The stack depth of $QUICKSORT'$ will be $\Theta(n)$ on an n-element input array if there are $\Theta(n)$ recursive calls to $QUICKSORT'$. This happens if every call to $PARTITION(A, p, r)$ returns $q = r$. The sequence of recursive calls in this scenario is:

$$QUICKSORT'(A, 1, n), QUICKSORT'(A, 1, n-1), QUICKSORT'(A, 1, n-2),$$

$$...QUICKSORT'(A, 1, 1).$$

As we discussed in class, applying the partition procedure discussed in class to a sorted list will always produce the worst stack depth.

c).

The problem demonstrated by the scenario in (b) is that each call of $QUICKSORT'$ calls $QUICKSORT'$ again with almost the same range (size only reduced by one). To avoid such behavior, we must change $QUICKSORT'$ so that recursive call is on a smaller interval of the array. The following variation of $QUICKSORT'$ checks which of the two sub-arrays returned from $PARTITION$ is smaller and recurses on the smaller sub-array, which is at most half the size of the current array. Since the array size is reduced by at least half on each recursive call, the number of recursive calls, and hence the stack depth, is $\Theta(\lg n)$ in the worst case. Note that this works no matter what PARTITION algorithms we use.

Note that a rough even split at each step produces the worst case stack depth. Compare this with the regular quicksort which has the best case running time when each split is a even split.

```
QUICKSORT''(A, p, r)
1.  while p < r
2.      do // Partition and sort the smaller subarray first
3.          q ← PARTITION(A, p, r)
4.          if q − p < r − q
5.              then QUICKSORT''(A, p, q − 1)
6.                   p ← q + 1
7.              else QUICKSORT''(A, q + 1, r)
8.                   r ← q − 1
```

The expected running time is not affected, because exactly the same work is done as before: The same partitions are produced, and the same sub-arrays are sorted.