Introduction to Algorithms: 6.006
Massachusetts Institute of Technology — March 7, 2008
Professors Srini Devadas and Erik Demaine — Handout 6

Quiz 1 Practice Problems

1 Asymptotic Notation

Decide whether these statements are **True** or **False**. You must briefly justify all your answers to receive full credit.

1. If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $h(n) = \Theta(f(n))$

   Solution: True. Θ is transitive.

2. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $h(n) = \Omega(f(n))$

   Solution: True. O is transitive, and $h(n) = \Omega(f(n))$ is the same as $f(n) = O(h(n))$

3. If $f(n) = O(g(n))$ and $g(n) = O(f(n))$ then $f(n) = g(n)$

   Solution: False: $f(n) = n$ and $g(n) = n + 1$.

4. $\dfrac{n}{100} = \Omega(n)$

   Solution: True. $\frac{n}{100} < c * n$ for $c = \frac{1}{200}$.

5. $f(n) = \Theta(n^2)$, where $f(n)$ is defined to be the running time of the program `A(n)`:

```
def A(n):
  atuple = tuple(range(0, n)) # a tuple is an immutable version of a
                              # list, so we can hash it
  S = set()
  for i in range(0, n):
    for j in range(i+1, n):
      S.add(atuple[i:j])   # add tuple (i,...,j-1) to set S
```

Solution: False: Inside the two for loops, both slicing and hashing take linear time.

2 Linked List Equivalence

Let S and T be two sets of numbers, represented as unordered linked lists of distinct numbers. All you have are pointers to the heads of the lists, but **you do not know the list lengths**. Describe an $O(\min\{|S|, |T|\})$-expected-time algorithm to determine whether $S = T$. You may assume that any operation on one or two numbers can be performed in constant time.

Solution: First, check that both sets are the same size. If they are not, then they cannot be equal. To do this check in $O(min\{|S|, |T|\})$ time, just iterate over both lists in parallel. That is, advance one step in S and one step in T. If both lists end, the lengths are the same. If one list ends before the other, they have different lengths.
If both lists are the same size, then we want to check whether the elements are the same. We create a hash table of size $\Theta(|S|)$ using universal hashing with chaining. We iterate over S, adding each element from S to the hash table. Then we iterate over T. For each element $x \in T$, we check whether x belongs to the hash table (that is, whether it is also in S). If not, then we return that the sets are not identical. If so, then continue iterating over T.
Any sequence of $|S| = |T|$ `Insert` and `Search` operations in the table take $O(|S|)$ time in expectation (see CLRS p.234), so the total runtime is $O(\min\{|S|, |T|\})$ in expectation.

3 Hash Table Analysis

You are given a hash table with n keys and m slots, with the simple uniform hashing assumption (each key is equally likely to be hashed into each slot). Collisions are resolved by chaining.
What is the probability that the first slot ends up empty?

Solution: Independently, each key has a $1/m$ probability of hashing into the first slot, or $(m-1)/m$ probability of not hashing into the first slot. Thus, the probability that no key hashes into the first slot is

$$\left(\frac{m-1}{m}\right)^n.$$

4 Heaps

1. The sequence $\langle 20, 15, 18, 7, 9, 5, 12, 3, 6, 2\rangle$ is a max-heap.
   True **False**

   Explain:

   Solution: **True.** For every node with 1-based index $i > 1$, the node with index $\lfloor \frac{i}{2} \rfloor$ is larger.

2. Where in a max-heap can the smallest element reside, assuming all elements are distinct? Include both the location in the array and the location in the implicit tree structure.

   Solution: The smallest element will be a leaf (because if it had a child, that child would have to be smaller). As seen in the quiz review, the leaves are the nodes indexed by $\lfloor \frac{n}{2} \rfloor + 1, \ldots, n$.

3. Suppose that instead of using `Build-Heap` to build a max-heap in place, the `Insert` operation is used n times. Starting with an empty heap, for each element, use `Insert` to insert it into the heap. After each insertion, the heap still has the max-heap property, so after n `Insert` operations, it is a max-heap on the n elements.

   (i) Argue that this heap construction runs in $O(n \log n)$ time.

   Solution: `Insert` takes $O(\log n)$ time per operation, and gets called $O(n)$ times.

   (ii) Argue that in the worst case, this heap construction runs in $\Omega(n \log n)$ time.

Solution: If you insert the elements in sorted order (starting with 1), then each insert puts the element at a leaf of the heap, before bubbling it up all the way to the root. This takes $\Theta(\log k)$ swaps, where k is the number of elements already in the heap.

$$\sum_{k=1}^{n} \log k = \Theta(n \log n)$$

5 Python and Asymptotics

1. Give an example of a built-in Python operation that does not take constant time in the size of its input(s). What time does it take?

   Solution: There are plenty of examples. One is the slice operator `s[i:j]` to take a substring of string s. This takes time $\Theta(j - i)$

2. Since dictionary lookup takes constant expected time, one can output all n keys in a dictionary in sorted order in expected time $O(n)$.
   True **False**

   Explain:

   Solution: False. Dictionaries do not store elements in sorted order. They can be oupput only in some random order.

3. Write a recurrence for the running time $T(n)$ of $f(n)$, and solve that recurrence. Assume that addition can be done in constant time.

```
def f(n):
  if n == 1:
    return 1
  else:
    return f(n-1)+f(n-1)
```

Solution: $T(n) = T(n-1) + T(n-1) + \Theta(1)$. If you draw the recurrence tree, in has n levels, where the amount of work done at each of the levels is $1+2+2^2+\cdots+2^{n-1} = \Theta(2^n)$

4. Write a recurrence for the running time of $g(n)$, and solve that recurrence. Assume that addition can be done in constant time.

```
def g(n):
  if n == 1:
    return 1
  else:
    x = g(n-1)
    return x+x
```

Solution: $T(n) = T(n-1) + \Theta(1)$. The recurrence tree is just a linked list, were the amount of work done at each level is $\Theta(1)$ for a total of $\Theta(n)$ work.

5. Now assume addition takes time $\Theta(b)$ where b is the number of bits in the larger number. Write a new recurrence for the running time of $g(n)$, and solve that recurrence. Express your final answer in Θ-notation.

Solution: The work done at each level is now $\Theta(n)$ because we have to add to numbers of $\Theta(n)$ bits.

$T(n) = T(n-1) + \Theta(n)$. The recurrence tree looks similar to the one in the previous part, but now at each step we have to do work proportional to the size of the problem: $n + (n-1) + \cdots + 1 = \Theta(n^2)$

6 Hashing

1. Given a hash table with more slots than keys, and collision resolution by chaining, the worst case running time of a lookup is constant time.
   True **False**

   Explain:

   Solution: False. In the worst case we get unlucky and all the keys hash to the same slot, for $\Theta(n)$ time.

2. Linear probing satisfies the assumption of uniform hashing.
   True **False**

   Explain:

   Solution: False. The second probe can be determined exactly from the first probe, while uniform hashing says that the entire permutation of probes is random.

3. Assume that $m = 2^r$ for some integer $r > 1$. We map a key k into one of the m slots using the hash function $h(k) = k \bmod m$. Give one reason why this might be a poorly chosen hash function.

   Solution: If our keys are all even, then we won't be using any of the odd slots.