Homework 2 Solutions

Chris Wu

The following solution were prepared by me (Chris), so if you find a typos email me at wu@cs.nyu.edu and not the professor. One comment is that these solutions are *complete* and contain many steps that are written for explanatory purposes so don't worry if you didn't write everything here.

Question 1

First let's find out how many ops we can do in one hour. So 1 hour is 60 minutes , which is 3600 seconds. So we can do, at most, $3.6 \cdot 10^{13}$ operations in one hour. The rest of the question is just solving for n in each instance.

- 6000000
- 33019.2725
- 60000
- $\approx 9.06 \cdot 10^{11}$
- ≈ 45
- ≈ 5.49

Aside: I just discovered that you can put "third root of 5" into google and it will give you $\sqrt[3]{5}$. Is there anything google can't do?

Question 2

First, let's notice there's a typo in the book. The third function is labeled g_4; just in case this causes confusion.

So the easiest way to do this is to take base 2 logs across the board. The important thing to note is that you're no longer working with $O(\cdot)$. So below we'll say that $f \leq g$ when there is a constant such that $f \leq g + c$. This c corresponds to the multiplicative constant if we did this with $O(\cdot)$. This leaves you with:

$$\sqrt{\log n}, n, \frac{4}{3}\log n, \log n + 3\log\log n, (\log n)^2, 2^n, n^2$$

Now, some structure should be familiar to you, namely:

$$\frac{4}{3}\log n \leq n \leq n^2 \leq 2^n$$

Now, for the last three. Let's first consider $\sqrt{\log n}$. Taking square roots certainly makes this smaller than $\frac{4}{3}\log n$. So we get:

$$\sqrt{\log n} \leq \frac{4}{3}\log n \leq n \leq n^2 \leq 2^n$$

What about $\log n + 3\log\log n$? Well, it's smaller than $\frac{4}{3}\log n$ because $\frac{4}{3}\log n = \log n + \frac{1}{3}\log n$ and $\frac{1}{3}\log n$ is bigger than $3\log n \log n$. It's clearly large than $\sqrt{\log n}$ though. So we have:

$$\sqrt{\log n} \leq \log n + 3\log\log n \leq \frac{4}{3}\log n \leq n \leq n^2 \leq 2^n$$

Finally, we have the term $(\log n)^2$. It's obvious that it's less than linear. Is it bigger than $\frac{4}{3}\log n$? Yes. It's same comparison as $\frac{4}{3}x \leq x^2$.

$$\sqrt{\log n} \leq \log n + 3\log\log n \leq \frac{4}{3}\log n \leq (\log n)^2 \leq n \leq n^2 \leq 2^n$$

Translating back to the original functions we have:

$$2^{\sqrt{\log n}} \leq n(\log n)^3 \leq n^{\frac{4}{3}} \leq n^{\log n} \leq 2^n \leq 2^{n^2} \leq 2^{2^n}$$

Question 3

Part a)

Let's first consider having just one jar. It's pretty clear that with only one jar you can't do better than simply going from the bottom of the building up and dropping the jar from every floor.

For two jars, the trick to beating $O(n)$ time is to use the first jar to find a small interval for the second jar. Drop the first jar from the $\sqrt{n}$th floor. If it breaks the do a simple scan from the first floor to the $\sqrt{n}$th one. If it doesn't break then proceed to the $2 * \sqrt{n}$th floor. If it breaks, you only need to start from the $(\sqrt{n}+1)$st floor.

There are at most $\sqrt{n}$ "chunks" of $\sqrt{n}$ floors. So what's the worst case for this? Well, if the highest floor at which the jar breaks is the $n-1$st one, then this approach will take $\sqrt{n}+\sqrt{n} = 2\sqrt{n}$ time. The first $\sqrt{n}$ corresponds to the dropping of the first jar all the way up. The second $\sqrt{n}$ corresponds to searching the last $\sqrt{n}$ or so floors for the $n-1$st.

Part b)

The generalization of this question turned out to be a little more tricky than anticipated.

The idea is that if you have k jars, you can look at groups of floors of size $n^{\frac{k-1}{k}}$. Then, using one jar and time at most $n^{\frac{1}{k}}$, you can reduce you search problem to one of size $n^{\frac{k-1}{k}}$ and $k-1$ jars.

So, in general, using one jar you reduce a problem of size $n^{\frac{k-i-1}{k}}$ to one of size $n^{\frac{k-1}{k}}$ in time $n^{\frac{1}{k}}$. So, in total, it will take $kn^{\frac{1}{k}}$ which is $O(n^k)$ since k is a fixed constant.

Question 4

Part a)

Let's recap what we're trying to show: we want a function f of h such that for any tree of height h, the number of nodes n is bound by $f(h)$. If we draw complete trees of height $0, 1, 2, 3$, we'll get sizes $1, 3, 7, 15$. Let's try and show that $f(h) = 2^{h+1} - 1$.

For the base case we start at $h = 0$. Clearly a tree of this height can't have more than 1 node. So $1 \leq 2^{0+1} - 1$. So we're finished the base case.

For the induction assume that all trees of height less than k are bounded by our function. What happens when we have a tree of height k, say T? Well, T will have a root and it's two children will be roots of subtrees of maximal height $k-1$. By our inductive hypothesis, these two children will have at most $2^k - 1$ nodes each. So $size(T) = 1 + size(left) + size(right) \leq 1 + 2(2^k - 1) = 2^{k+1}$. This completes the induction.

Part b)

This question is a little easier. We want a function f of n such that for a tree with n nodes, it can't be taller than h. Here $f(n) = n - 1$ is the answer. It corresponds to the case where the tree is just a chain of nodes.

Question 5

Part a)

As suggested, we first show that i) $n^k \preceq n^{k'}$. For domination we need to show there is a positive C as well as a starting point n_0 such that

$$n^k \leq C \cdot n^{k'} \qquad \forall n > n_0$$

I choose $C = 1$ and $n_0 = 1$. Then is it true that

$$n^k \leq n^{k'} \qquad \forall n > 1$$

Well, $n^{k'-k} > 1$ for any $n > 1$ since we assumed that $k' > k$. So, yeah, it is.

For ii), we do a contradiction, assume there is a C and an n_0 such that

$$n^{k'} \leq C \cdot n^k \qquad \forall n > n_0$$

Well, consider a value of n where $n > C^{\frac{1}{k'-k}}$. If $C^{\frac{1}{k'-k}} < n_0$, then just choose $n_0 + 1$. In this case,

$$\begin{aligned} n^{k'} &> C^{\frac{k'}{k'-k}} \\ &= C^{\frac{k'-k+k}{k'-k}} \\ &= C^{1+\frac{k}{k'-k}} \\ &= C \cdot C^{\frac{k}{k'-k}} \\ &= C \cdot n^k \end{aligned}$$

This is a contradiction.

Aside: Where did I get the value of $C^{\frac{1}{k'-k}}$? Well, you can solve for n in

$$n^{k'} \leq C \cdot n^k \qquad \forall n > n_0$$

by taking logs of both sides and isolating n. Then you'll have something like $n < t$ for some term t which is independent of n. Just negate that statement and you'll have your number n.

Part b)

The trick is to group the terms of the harmonic series as follows: the first, the next two, the next four, the next eight and so on. I'll start counting from 0 so that (1) is the 0th group.

Before I do the proofs let's notice a small fact about this grouping. If G_i refers to the sum of all the terms in the ith group then

$$1/2 \leq G_i \leq 1$$

This isn't too hard to see. Notice that the smallest term of the ith group is $\frac{1}{2^{i+1}-1}$. The group has size 2^i. Since $\frac{2^i}{2^{i+1}-1} > \frac{1}{2}$, we know that if we replace all the items of a group with the smallest, we get something more than $\frac{1}{2}$.

Similarly, the largest item of the group is $\frac{1}{2^i}$ and $2^i * \frac{1}{2^i} = 1$. So if we replaced all the items of the group with copies of the largest, we'd get 1.

Let's first show that $H_n \preceq \log n$. We have $H_n = (1) + (\frac{1}{2} + \frac{1}{3}) + (\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}) + \ldots$.Now notice that there are $\lfloor \log n \rfloor + 1$ groups if n is a power of 2. Otherwise, there's one more.

So using this we know that:

$$H_n \leq \lfloor \log n \rfloor + 2 \leq \log n + 2$$

We're not quite done yet since this doesn't follow the definition of dominance/eventuality. So we can replace the 2 with another $\log n$ term if $n > 4$. Then we can conclude that $H_n \leq 2 * \log n$ for $n > 4$. So setting $C = 2$ and $n_0 = 4$ we've shown that

$$H_n \preceq \log n$$

Now to show that $\log n \preceq H_n$, we do something similar. This time, instead of working on the largest term of a group, we're going to work on the smallest term of a group.

So we know that

$$\begin{aligned}\log n &= 1 + 1 + \ldots + 1 \qquad (\log n \text{ 1's}) \\ &\leq 2[(1) + (\frac{1}{2} + \frac{1}{3}) + (\frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7}) + \ldots] \\ &= 2H_n\end{aligned}$$

For completion, we state that $C = 1$ and $n_0 = 1$ to get the result.

Aside: Here, if n isn't a power of 2, the proof still works. If the last group is incomplete, the smallest member is larger than if the group had been complete.

Question 6

Part a)

Remember that when we do an extract-min, we remove the root node, then replace it with the last node in the tree and then fix the heap property down the length of the tree. There are $\lceil \log 200 \rceil = 8$ levels and we do two comparisons at each level so that's 16 comparisons total.

Part b)

The cleanest way to do this is the following approach. Assume that we have the function $heapify(A, i)$ as in the book. This assumes that i children are heaps themselves and simply creates a heap from the two subtrees and the parent by pushing the parent down (if necessary).

Now, given a tree, all the leaves are trivially heaps so we are done with them. Starting from the $n-1$st level, we heapify each node.

The formal pseudocode is below. I assume the input is in an array A and the n is the size of the input.

```
for $i = \lceil \frac{n}{2} \rceil$ to 1 do
    heapify(A,i)
end for
```

Part c)

Here's how it looks at each iteration:

Start

17	3	1	11	7	5	19	13	16	4	2	10	8

$Heapify(A, 6)$

17	3	1	11	7	5	19	13	16	4	2	10	8

$Heapify(A, 5)$

17	3	1	11	2	5	19	13	16	4	7	10	8

$Heapify(A, 4)$

17	3	1	11	2	5	19	13	16	4	7	10	8

$Heapify(A, 3)$

17	3	1	11	2	5	19	13	16	4	7	10	8

$Heapify(A, 2)$

17	2	1	11	3	5	19	13	16	4	7	10	8

$Heapify(A, 1)$

1	2	17	11	3	5	19	13	16	4	7	10	8
1	2	5	11	3	17	19	13	16	4	7	10	8
1	2	5	11	3	8	19	13	16	4	7	10	17

Part d)

$Heapify(A, 6)$: 2 comparisons
$Heapify(A, 5)$: 2 comparisons
$Heapify(A, 4)$: 2 comparisons
$Heapify(A, 3)$: 2 comparisons
$Heapify(A, 2)$: 2 comparisons
$Heapify(A, 1)$: 6 comparisons
For a total of 16 comparisons

Part e)

Since $heapify(A, i)$ may take up to $\log n$, a simple analysis would yield an $O(n \log n)$ running time. If we are more careful, we can do better.

Intuitively, we see that the $\log n$ bad cases only appear near the top of the tree: where there are very few nodes. Most nodes are taken care of at the bottom: where the height is low.

At a height of h, there are at most $\frac{n}{2^{h+1}}$ nodes. A call to $heapify$ can take as much time as the height of the input node. So the total running time can be at most $n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^h}$.

From summations we know that

$$\sum_{i=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Which, for us gives

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

So we can upper bound the summation as

$$n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^h} \leq 2n$$

This means the running time is $O(n)$.