Homework 1

Zicong Mo (UID: 804654167)

January 19th, 2017

2. Since the computer can compute 10^{10} operations per second and there are 3600 seconds in one hour, the computer can compute a total of $10^{10} \times 3600 = 3.6 \times 10^{13}$ operations. To get the largest input size n for which we would get the result within an hour, we solve the inequality $f(n) \leq 3.6 \times 10^{13}$, and take the largest integer n for which the inequality is true.

   (a) $n^2 \leq 3.6 \times 10^{13} \Rightarrow n \leq 6 \times 10^6$. The largest input size is $n = 6 \times 10^6$.

   (b) $n^3 \leq 3.6 \times 10^{13} \Rightarrow n \leq 33019.27$. The largest input size is $n = 33019$.

   (c) $100n^2 \leq 3.6 \times 10^{13} \Rightarrow n \leq 6 \times 10^5$. The largest input size is $n = 6 \times 10^5$.

   (d) $n \log_2 n \leq 3.6 \times 10^{13} \Rightarrow n \leq 9.06 \times 10^{11}$. The largest input size is $n = 9.06 \times 10^{11}$.

   (e) $2^n \leq 3.6 \times 10^{13} \Rightarrow n \leq 45.03$. The largest input size is $n = 45$.

   (f) $2^{2^n} \leq 3.6 \times 10^{13} \Rightarrow n \leq 5.49$. The largest input size is $n = 5$.

3.

$$
\begin{aligned}
f_2(n) &= \sqrt{2n} \\
f_3(n) &= n + 10 \\
f_6(n) &= n^2 \log n \\
f_1(n) &= n^{2.5} \\
f_4(n) &= 10^n \\
f_5(n) &= 100^n
\end{aligned}
$$

7. A song with n total words and c words per line has a total of n/c lines. Suppose we have just finished iteration k of the song. The next line is the new addition to the song, before repeating iteration k again. Therefore the number of lines at the end of iteration k is the kth triangular number. The total number of iterations can then be solved with the equation:

$$\frac{k(k+1)}{2} = \frac{n}{c}$$

$$k = \frac{1}{2}(\sqrt{1 + \frac{8n}{c}} - 1)$$

An example of the algorithm in Python:

```
# Assume the words are in a list of length n
num_iterations = 0.5 * (sqrt(1 + 8*n/c) - 1)
for i in range(num_iterations):
    line_number = i * (i+1) /2
    for j in range(c):
        print(words[line_number*c+j], end=" ")
    print("\n")
```

For each of the k iterations, we print out c words. Therefore the complexity of this algorithm is $O(kc) = O(c\sqrt{\frac{n}{c}})$.

8. (a) Let x be the smallest value such that $x(x+1) \geq 2n$.
Start by dropping a jar at rung x.
If it breaks, then we can do a linear search with the other jar from rung 1 to rung $x-1$. We can find exactly which of these is the highest safe rung in at most $1+x-1=x$ drops.
Otherwise, drop a jar at rung $x+(x-1)$. If this jar breaks, then we can do a linear search from rung $x+1$ to rung $2x-2$. The highest safe rung in this range can be found in at most $2+(2x-2)-(x+1)+1=x$ drops.
If the jar didn't break, continue on at rung $x+(x-1)+(x-2)$, and so on.

We show that the number of steps needed to find the highest safe rung is at most x. To do this, we prove that every rung is reached in at most x drops. Note that the first x rungs are covered by at most $1+(x-1)=x$ drops, the next $x-1$ rungs are covered by at most $2+(2x-2)-(x+1)+1=x$ drops, the next $x-2$ rungs are covered by at most $3+(3x-4)-2x+1=x$ drops, etc.
Because we define x as the smallest number such that $x+(x-1)+...+1 \geq n$, the entire ladder can be covered by at most x drops.

Since the drop-time of this algorithm is at most x, and the value of x depends on $\sqrt{n}$, our algorithm runs in $O(\sqrt{n})$, which is better than linear time.

(b) We apply a similar strategy in the general case.
Let x be the smallest integer greater than $n^{1-\frac{1}{k}}$. Let $f_k(n)$ be the minimum number of drops needed to locate the highest safe rung using this strategy in a ladder with n rungs and k jars. We drop the first jar at integer multiples of x, until we reach the top rung or the jar breaks at some lower rung.
Because $x \geq n^{1-\frac{1}{k}}$, note that

$$nx \geq n \cdot n^{1-\frac{1}{k}}$$
$$\frac{n}{n^{1-\frac{1}{k}}} \geq \frac{n}{x}$$

Because we drop the first jar at integer multiples of x, the number of times we can drop the first jar is bounded by n/x:

$$d_1 \leq n/x \leq n/n^{1-\frac{1}{k}} = n^{\frac{1}{k}}$$

Once the first jar breaks, we have $k-1$ jars to search through x floors, which can be done in $f_{k-1}(x)$ drops. This recursion can be written as:

$$\begin{aligned} f_k(n) &= d_1 + f_{k-1}(x) \\ &\leq n^{\frac{1}{k}} + f_{k-1}(x) \\ &\leq n^{\frac{1}{k}} + f_{k-1}(n^{1-\frac{1}{k}}) \end{aligned}$$

We guess that the runtime of our algorithm is bounded by some polynomial function whose degree decreases as $k \to \infty$, as this type of function gives us the desired limiting behavior of f. Because of the prevalence of $n^{1/k}$ in our recurrence, we guess that, for some value C_k,

$$f_k(n) \leq C_k n^{1/k}.$$

Assume that $f_{k-1}(n) \leq C_{k-1} n^{1/(k-1)} \Rightarrow f_{k-1}(n^{1-\frac{1}{k}}) \leq C_{k-1}(n^{1-\frac{1}{k}})^{1/(k-1)}$. Then, by induction,

$$\begin{aligned}
f_k(n) &\leq n^{\frac{1}{k}} + f_{k-1}(n^{1-\frac{1}{k}}) \\
&\leq n^{\frac{1}{k}} + C_{k-1}(n^{1-\frac{1}{k}})^{1/(k-1)} \\
&\leq n^{\frac{1}{k}} + C_{k-1}(n^{\frac{k-1}{k}})^{1/(k-1)} \\
&\leq n^{\frac{1}{k}} + C_{k-1} n^{\frac{1}{k}} \\
&\leq n^{\frac{1}{k}}(1 + C_{k-1})
\end{aligned}$$

Since we want $f_k(n) \leq C_k n^{1/k}$, if we can solve $1 + C_{k-1} = C_k$, our inductive step is complete. We see from inspection that $C_k = k$ has this property.

$$\begin{aligned}
f_k(n) &\leq n^{\frac{1}{k}}(1 + C_{k-1}) \\
&\leq n^{\frac{1}{k}}(1 + k - 1) \\
&\leq n^{\frac{1}{k}}(k) \\
&\leq C_k n^{\frac{1}{k}}
\end{aligned}$$

To finish the proof, we show that this inequality holds for the base case $k = 2$. We showed in part (a) that this strategy produces a runtime of $O(n^{1/2})$, so the runtime must also be bounded by $2n^{1/2}$. Therefore we have shown that the maximum number of drops needed by this strategy to find the critical rung is

$$f_k(n) \leq k n^{1/k}$$

Note that

$$\lim_{n\to\infty} \frac{f_{k+1}(n)}{f_k(n)} = \lim_{n\to\infty} \frac{k+1}{k} \frac{n^{1/(k+1)}}{n^{1/k}} = \lim_{n\to\infty} \frac{k+1}{k} n^{-\frac{1}{k(k+1)}} = 0,$$

so each function f_{k+1} grows slower than the previous function f_k.