1 Big-O Notation

Prove each of the following using the definition of big-O notation (find constants $c$ and $n_0$ such that $f(n) \leq c \times g(n)$ for $n > n_0$).

- $3n^3 + 9n^2 + n + 1 = O(n^3)$
- $5n \log_2 n + 8n - 200 = O(n \log_2 n)$

Solution

- For all $n \geq 1$, we have $f(n) = 3n^3 + 9n^2 + n + 1 \leq 3n^3 + 9n^3 + n^3 + n^3 = 14n^3$. So let $n_0 = 1$, for all the $n > n_0$, we have $f(n) \leq c \times g(n)$, where $c = 14$, and $g(n) = n^3$, and we get $f(n) = O(n^3)$.

- For all $n \geq 2$, we have $f(n) = 5n \log_2 n + 8n - 200 \leq 5n \log_2 n + 8n \log_2 n = 13n \log_2 n$. We let $n_0 = 2$, for all $n > n_0$, we have $f(n) \leq c \times g(n)$, where $c = 13$ and $g(n) = n \log_2 n$, so we get $f(n) = O(n \log_2 n)$.

2 More Big-O

Order the following by their growth rates from smallest to largest.

1. $O(n^{1.9})$
2. $O(n^3)$
3. $O(\log n)$
4. $O(n)$
5. $O(n^n)$
6. $O(\sqrt{n})$
7. $O(2^n)$
8. $O(n!)$
9. $O(n \log n)$

Solution 3, 6, 4, 9, 1, 2, 7, 8, 5

3 And some more...

Write each of the following in big-O notation. No proof necessary.

- $2n + 3n \log n + 15n^{1.1} =$
- $3n + 5n(\log n)^{555} + n^3 =$
- $2n(3 + \log n + n^2) =$
- $2n^{55} + 2^n =$
- $2n(\log n)^4 + n^2 =$

Solution

- $2n + 3n \log n + 15n^{1.1} = O(n^{1.1})$
- $3n + 5n(\log n)^{555} + n^3 = O(n^3)$
- $2n(3 + \log n + n^2) = O(n^3)$
- $2n^{55} + 2^n = O(2^n)$
- $2n(\log n)^4 + n^2 = O(n^2)$
4 Merge sort

Slow-Merge-Sort is an algorithm that works the same as the merge-sort we looked at in class, but in the merge step it uses an algorithm called Slow-Merge that takes $O(n^2)$ time.

- Write a recurrence for the runtime of Slow-Merge-Sort
- Solve the recurrence using the substitution method
- Express the runtime of Slow-Merge-Sort using Big-O notation

Solution

4.1 a

The recurrence for the runtime of Slow-Merge-Sort is $T(n) = 2T\left(\frac{n}{2}\right) + O(n^2)$.

4.2 b

By using the substitution method, first, we guess $T(n)$ is $O(n^2)$, so there is $c > 0$ such that $T(n) \leq cn^2$ for all sufficiently high $n$. We have

\[
T(n) = 2T\left(\frac{n}{2}\right) + n^2
\leq 2c\left(\frac{n^2}{4}\right) + n^2
= c\frac{n^2}{2} + n^2
= \left(c\frac{1}{2} + 1\right)n^2
\leq cn^2 \text{ (when } c \geq 2)\
\]

4.3 c

So the runtime of Slow-Merge-Sort is $O(n^2)$. 

3
5  Recurrences

Solve the following recurrences. Assume $T(n) \leq c$ for some constant $c$ and for all $n \leq 10$.

- $T(n) = 2T\left(\frac{n}{4}\right) + n^{0.3}$
- $T(n) = 4T\left(\frac{n}{2}\right) + n^2 \sqrt{n}$

Solution

5.1  a

\[
T(n) = 2T\left(\frac{n}{4}\right) + n^{0.3}
\]

\[
= 2(2T\left(\frac{n}{16}\right) + \left(\frac{n}{4}\right)^{0.3}) + n^{0.3} = 4(2T\left(\frac{n}{64}\right) + \left(\frac{n}{16}\right)^{0.3}) + 2\left(\frac{n}{4}\right)^{0.3} + n^{0.3}
\]

\[
= 8T\left(\frac{n}{64}\right) + 4\left(\frac{n}{16}\right)^{0.3} + 2\left(\frac{n}{4}\right)^{0.3} + n^{0.3}
\]

\[
\ldots
\]

\[
= 2^iT\left(\frac{n}{4^i}\right) + 2^{i-1}\left(\frac{n}{4^{i-1}}\right)^{0.3} + 2^{i-2}\left(\frac{n}{4^{i-2}}\right)^{0.3} + \ldots + 2^0\left(\frac{n}{4^0}\right)^{0.3}
\]

\[
= 2^iT\left(\frac{n}{4^i}\right) + \sum_{k=0}^{i-1} 2^k \left(\frac{n}{4^k}\right)^{0.3}
\]

The basis case is $T(1) \leq c$, in which $\frac{n}{4^i} = 1$. So we have $4^i = n$ and $i = \log_4 n$. Therefore we have

\[
T(n) = \sqrt{n}T(1) + \sum_{k=0}^{i-1} 2^{0.4k} = \sqrt{n}T(1) + n^{0.3} \frac{1 - n^{0.2}}{1 - 2^{0.4}}
\]

\[
= (T(1) - \frac{1}{1 - 2^{0.4}})\sqrt{n} + \frac{1}{1 - 2^{0.4}} n^{0.3}
\]

\[
= O(\sqrt{n})
\]
5.2 b

\[ T(n) = 4T\left(\frac{n}{2}\right) + n^2 \sqrt{n} \]
\[ = 4(4T\left(\frac{n}{4}\right) + \frac{n^2}{4} \sqrt{\frac{n}{2}}) + n^2 \sqrt{n} = 16T\left(\frac{n}{4}\right) + n^2 \sqrt{\frac{n}{2}} + n^2 \sqrt{n} \]
\[ = 16(4T\left(\frac{n}{8}\right) + \frac{n^2}{16} \sqrt{\frac{n}{4}}) + n^2 \sqrt{\frac{n}{2}} + n^2 \sqrt{n} \]
\[ = 64T\left(\frac{n}{8}\right) + n^2 \sqrt{\frac{n}{4}} + n^2 \sqrt{\frac{n}{2}} + n^2 \sqrt{n} \]
\[ = 4^iT\left(\frac{n}{2^i}\right) + n^{\frac{i}{2}} \sum_{k=0}^{i-1} \left(\frac{1}{\sqrt{2}}\right)^k \]

The basis case is \( T(1) \leq c \), in which \( \frac{n}{2^i} = 1 \), we have \( 2^i = n \) and \( i = \log_2 n \). And because \( 0 < \frac{1}{\sqrt{2}} < 1 \), so we get

\[ T(n) \leq n^2T(1) + n^{\frac{3}{2}} \frac{1}{1 - \frac{1}{\sqrt{2}}} \]
\[ = O(n^{\frac{5}{2}}) \]

6 Compare two lists

You are given two unsorted lists of integers, \( L_1 \) and \( L_2 \). There may be duplicates in the lists and the length of \( L_1 \) and \( L_2 \) are both \( n \). Now we want to test whether \( L_1 \) and \( L_2 \) contain exactly the same list of integers (with duplicates allowed). For example, let \( L_1 = \{3, 1, 1, 5\} \) and \( L_2 = \{1, 5, 3, 1\} \). Then we say \( L_1 \) and \( L_2 \) contain the same list of integers. But \( \{1, 1, 2, 2\} \) and \( \{1, 2, 2, 2\} \) do not.

Now, write down an efficient algorithm and analyze the running time.

**Solution** One solution takes \( O(n^2) \) time by iterating through the first list, and for each element checking whether it is in the second list (and removing to deal with duplicates easily).

Notice that if \( L_1 \) and \( L_2 \) are both sorted then it is simpler to compare them. Therefore a more efficient algorithm is to first sort the two lists using a \( O(n \log n) \) sorting algorithm.
Then we start by comparing the first elements, then the second elements, and so on. If each such pair is equal then the lists’ contents are the same. Otherwise we find 2 elements that don’t match and the lists don’t contain the same integers. This algorithm takes $2n \log n$ time to sort the lists and $n$ time to scan them and check corresponding elements. Overall this is $O(n \log n)$.

This can also be done in $O(n)$ time using hashing, but the sorting solution is fine for now.

7 Partition

The following array has been partitioned. Which elements could have been the pivot value?

\[16, 25, 8, 40, 32, 42, 55, 67, 59, 73\]

**Solution** In a partition operation, all elements with values less than the pivot locate at one side of the pivot while all elements greater than the pivot locate at another side. In the array, 42 or 55 could have been the pivot.

8 Pancake flipping

A stack of $n$ pancakes is placed in front of you. You have a spatula which you can insert anywhere into the stack and flip over all the pancakes above the spatula. You want to arrange the pancakes in order of their diameter (they are perfectly round), and you want to use as few flips as possible. As an example suppose $n = 6$, and the pancakes are numbered 1 through 6 in order of their diameter with 1 the smallest and 6 the largest. Suppose the original order is 346215, and the left end of the sequence represents the top of the stack. In one flip I can get 643215 (by flipping the first three pancakes: 346), then in the next flip 512346, then 432156, then 123456, so four flips are enough in this case.

Let $F(n)$ be the worst case number of flips needed to arrange a stack of $n$ pancakes. Find an efficient algorithm for this problem, where efficiency is measured by the worst case number of flips. Remember that your algorithm should work for pancakes with any order. To start you off you should easily be able to show that $F(n)$ is at most $2n$. Next, reduce that bound a little more if you can.
**Solution**  Remember that we can only flip the top part of the pancakes. One approach is to go incremental: we make some progress each time toward what we want to achieve (that is, pancakes in sorted order). One way to make progress is to move the largest pancake to bottom (if it is not already there). That is, at least the largest one is in place. It takes at most two flips to do this. The first flip the pancakes where the lowest one is the largest one we want to move. After this flip, the largest pancake moves to the top. Then we flip the whole stack of pancake. Now how should we continue? Since the largest one is in place, we will now only consider the remaining $n - 1$ pancakes. We do this by simply re-apply the previous technique: each time move the largest (but not in place) pancake to its right position by two flips at most. During this, we will not touch the pancakes that are already in place.

Now analysis: there are $n$ pancakes, and moving each to its right position in the above order takes no more than 2 flips. So clearly we can do it with no more than $2n$. This can be improved easily to $2n - 2$ since the smallest pancake needs no flip: it is already in place once every other pancake is in place. We can reduce this to $2n - 3$ (or slightly smaller) by considering what happens to the second smallest and so on.

A side note: Bill Gates (co-founder of Microsoft) studied this problem and wrote a research paper with a CS professor on this problem about 30 years ago, when he was an undergraduate student at Harvard University. They show that at most $(5n + 5)/3$ flips are needed, which is much smaller than $2n - 3$ when $n$ is large. Read their paper if you want to see how they did it: Gates, W., Papadimitriou, C. (1979). “Bounds for Sorting by Prefix Reversal”. Discrete Mathematics 27: 47-57.

9 Glass Jar Problem

Suppose you have 2 identical glass jars and you’re in a skyscraper with a 100 floor spiral stairwell. Your goal is to determine the highest floor from which you can drop a jar without it breaking. Describe an algorithm to do this with the fewest drops in the worst case. Repeat for a general building with $n$ floors.

**Solution**  First note that you can not take the binary search approach: you only have two jars and the binary search may need more than two jars. Here is the basic observation. Suppose you throw the first jar at floor $f$. If it does not break, you can test floors $f + 1$ and above. What if it breaks? Since you only have one jar left, you have to try throwing the jar from the lowest floor that you are not sure whether the jar will break or not, and each time move up to the floor right above. That is, you can not skip any floor you are not sure about when you only have one jar left. So this is what you have to do. Suppose you will need to throw $x$ times in the worst case. Then we need to first pick the floor $x$ to throw a jar. Why? Let the first floor to throw a jar be $f_1$. If the first jar breaks at $f_1$,
we have to throw $f_1 - 1$ times in the worst case. So $1 + (f_1 - 1) = x$ (here 1 refers to the first throwing). For each $2 \leq i \leq x$, if the previous throwing at $f_{i-1}$ does not break, pick the floor $f_i = f_{i-1} + (x - i + 1)$ and throw a jar; otherwise, start from the floor $f_{i-2} + 1$ and test each floor up to floor $f_{i-1} - 1$ (which will surely find the highest safe rung) and stop. Why setting $f_i = f_{i-1} + (x - i + 1)$? Note that there are $f_i - f_{i-1} - 1$ floors between $f_i$ and $f_{i-1}$ (excluding these two floors), which we may have to test each of these floors if the jar breaks at $f_i$. Since we have already thrown $i$ times (till the throwing at $f_i$), we have $x = (f_i - f_{i-1} - 1) + i$.

By summing up, we have $f_x = \sum_{i=1}^{x} (x - i + 1) = x(x + 1)/2$. Since $f_x$ can be as large as $n$. The obvious choice for $x$ is the smallest integer s.t. $x(x + 1)/2 \geq n$. When $n = 100$, $x = 14$. When $n$ is large, $x \approx \sqrt{2n}$. 
