CS 6150: HW1 – Data structures, recurrences

Submission date: Monday, Sep 13, 2021 (11:59 PM)

This assignment has 6 questions, for a total of 50 points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question	Points	Score
Basics	7	
Bubble sort basics	5	
Deletion in prefix trees	6	
Binary search and test pooling	10	
Recurrences, recurrences	16	
Dynamic arrays: is doubling important?	6	
Total:	50	

Note. *When asked to describe and analyze an algorithm, you need to first write the pseudocode, provide a running time analysis by going over all the steps (writing recurrences if necessary), and provide a reasoning for why the algorithm is correct.* Skipping or having incorrect reasoning will lead to a partial credit, even if the pseudocode itself is OK.

Question 1: Basics .... **[7]**

In (b)-(d) below, answer YES/NO, along with a line or so of reasoning. Simply answering YES/NO will fetch only half the points.

(a) **[1]** Sign up for the course on Piazza!

(b) **[2]** Let $f(n)$ be a function of integer parameter n, and suppose that $f(n) \in O(n \log n)$. Is it true that $f(n)$ is also $O(n^2)$?

Answer. Yes, if $f(n) \in O(n \log n)$ then $f(n) \in O(n^2)$. This is because $n \log n < n^2$.

(c) **[2]** Suppose $f(n) = \Omega(n^{2.5})$. Is it true that $f(n) \in o(n^5)$?

Answer. No. Because we could have $f(n) = n^6$, and so a lower bound could be $f(n) \in \Omega(n^2)$, but clearly $f(n) \notin o(n^5)$. Note that the lower bounds do not give any information on the upper bounds.

(d) **[2]** Let $f(n) = n^{\log n}$. Is $f(n)$ in $o(2^{\sqrt{n}})$?

Answer. We have to look at the limit $\lim_{n\to\infty} \frac{n^{\log n}}{2^{\sqrt{n}}} = 0$; The limit must converge to 0 for $f(n)$ to be in $o(2^{\sqrt{n}})$.

$$\lim_{n\to\infty} \frac{n^{\log n}}{2^{\sqrt{n}}} = \lim_{n\to\infty} \frac{2^{\log n \times \log n}}{2^{\sqrt{n}}} = \lim_{n\to\infty} 2^{(\log n)^2 - \sqrt{n}} = 2^{\lim_{n\to\infty} (\log n)^2 - \sqrt{n}} = 2^{-\infty} = 0$$

As $(\log n)^2 - \sqrt{n} \to -\infty$, the limit above converges to 0. Thus, $f(n) \in o(2^{\sqrt{n}})$

Question 2: Bubble sort basics .... **[5]**

Recall the bubble sort procedure we saw in Lecture 1 (see notes): while the input array $A[]$ is not sorted, go over the array from left to right, swapping i and $(i+1)$ if they are out of order. As I mentioned in class, the running time of the algorithm depends on the input.

Given a parameter $1 < k < n$, give an input array $A[]$ for which the bubble sort procedure takes time $\Theta(nk)$. (Recall that to prove a $\Theta(\cdot)$ bound, you need to show upper and lower bounds.)

Answer. Suppose we have the following array,

$$A = [n - (k = 1), n - (k - 2), \dots, n, 1, 2, \dots, n - k]$$

If we run Bubble Sort on this array we will first bubble up the n and then bubble up $n - 1$ and so on until $n - (k - 1)$, at which point we would have done k iterations and we would have a sorted array. We can easily see that each iteration takes $\Theta(n)$ and thus the overall time complexity is $\Theta(nk)$.

Question 3: Deletion in prefix trees .... **[6]**

In class, we saw how a prefix tree can be used to store a *set of strings* (which we called a dictionary) over some alphabet Σ. Specifically, we saw how to implement the `add` and `query` operations on such a data structure. (See the lecture notes for more details.) Now, consider implementing the `delete` operation. As suggested in the notes, one way to do this is to mimic the query, and for the node corresponding to the word, set the "IsWord" boolean to false.

However, if we are adding and removing multiple strings, this can lead to many tree paths that were created, but don't correspond to any word currently in the dictionary.

Show how to modify the data structure so that this can be avoided. More formally, if S is the set of words remaining after a sequence of add/delete operations, we would like to ensure space utilization that is the same (possibly up to a constant factor) of the space needed to store only the elements of S. If your modifications impact the running time of the `add`, `query`, and `delete` operations, explain how.

Answer: We begin by observing that we can easily keep track of if any node is a leaf node or not. This can be done by using the pointers to any children: If in some node there exist pointers to children, then this is not a leaf node. In this case, we do not store pointers to all 26 children for each node, but only store pointers to nodes traversed during the `add` operation.

Next we can modify the delete operation. Similar to our `add` and `query` operations, we traverse the trie using the characters of the string at each step.

Algorithm 1 DELETE($W, node, i = 0$)

```
Let $k \leftarrow$Length of $W$.
$node2 \leftarrow$ child of $node$ corresponding to $W[i]$
if $i < k$ then
    DELETE($W, node2, i + 1$)
else if $i = k$ then
    Set $node2.IsWord$ to false
    If $node2$ has no children, delete.
end if
if $node$ has no children and $node.IsWord$ is false, delete
```

- Make k steps down the trie, where k is the length of our word After k steps down the trie, where k is the length of our string for deletion, we arrive at a leaf node to be deleted. We first mark this node as invalid. Next, we can check if this node has no children: if so, we can delete this node.
- We recursively walk up the tree and delete nodes that have no children and are invalid.

This algorithm will not impact the runtime of the delete operation ($O(l)$, where l is the length of the word to be deleted), since determining if a node is valid and has children can be done in $O(1)$.

Correctness: As we recurse down the trie, we find the node corresponding to the delete instruction. After we set this node to invalid, this node can safely be deleted if it has no children, since there is no other word to be traversed using this node.

As we recurse back up the trie, we observe the same: If an invalid node has no children, then it is not necessary to traverse this node in any query, so we can delete the node.

Space complexity: In the worst case (all strings do not share any characters in the same position), the number of nodes in the trie will be nk, where k is the average length of all n words in the trie. This delete operation will remove all invalid nodes with no children, meaning that as n decreases by 1, the number of nodes will decrease by k.

If the trie is not the worst case for space complexity (some strings share characters in the same position), then when a leaf-node becomes invalid, we will remove $\leq k$ nodes when n decreases by 1. Either way, we maintain the $O(nk)$ bound on space complexity.

Rubric: 3 points for reasonable algorithm and adaption, 2 points for argument of correctness, 1 point for space complexity.

Question 4: Binary search and test pooling .... **[10]**

"Test pooling" is a trick that is used when testing for a disease is expensive or has limited availability. The idea is the following: suppose we have n people (numbered $1, 2, \ldots, n$ for convenience), instead of testing each one, samples from a subset S of the people are combined and tested, where a test runs in $O(1)$ time regardless of the size of S. If at least one of the people in S has the disease, the test comes out positive, and if none of the people in S has the disease, it comes out negative. (Let us ignore the test error for this problem.)

It turns out that if only a "few" people have the disease, this is much better than testing all n people.

(a) [**4**] Suppose we know that *exactly one* of the n people has the disease and our aim is to find out which one. Describe an algorithm that runs in time $O(\log n)$ for this problem. (For this part, pseudocode suffices, you don't need to analyze the runtime / correctness.)

Answer: We can use an algorithm which is quite similar to binary search. We divide the samples into 2 halves, pool each halves, and then test the 2 pooled sample. Whichever pooled sample tests positive, do the same thing again.

Algorithm 2 $find1(samples, start, end)$

```
if start == end then
    return start
end if

sample = $pool(samples, start, floor(end/2))$                ▷ O(1)

if $test(sample)$ then                                         ▷ O(1)
    return $find1(samples, start, floor(end/2))$
else
    return $find1(samples, floor(end/2) + 1, end)$
end if
```

$T(n) = T(n/2) + O(1)$

The above recurrence is similar to binary search, which has a time complexity of $\theta(lg(n))$

Ruberics:

[+2] Pseudo code: Terminating step.

[+2] Pseudo code: Dividing it into 2 sub-problems

[-n] if T.C $\notin \mathcal{O}(lg(N))$

(b) [**6**] Now suppose we know that *exactly two* of the n people have the disease and our aim is to identify the two infected people. Describe **and analyze** (both runtime and correctness) an algorithm that runs in time $O(\log n)$ for this problem.

Answer:

Solution 1: use the above algo

Assuming the correctness of the above $find1$ algorithm, we can use it to ID 1 of the 2 infected samples. After IDing the blood sample, we empty out that particular sample. This will leave only 1 infected sample among the n samples. We can use the $find1$ algorithm again, to help identify the 2nd/remaining infected sample. So, we end up calling the $find1$ co-routine twice, to ID both the infected sample. Since $find1$ has a Time complexity of $\theta(lg(n))$ and throwing out the sample has time complexity of $O(1)$, which means the time complexity of $find2$ is also $\theta(lg(n))$.

Algorithm 3 $find2(samples, start, end)$

```
sampleNumber1 = $find1(samples, start, end)$            ▷ $\theta(lg(n))$
throwOut(samples, sampleNumber1)        ▷ Throw out the infected sample we ID'd, O(1)
sampleNumber2 = $find1(samples, start, end)$            ▷ $\theta(lg(n))$
```

But this all hinges on the correctness of the $find1$ algorithm. So to prove the correctness of $find2$, we need to prove the correctness of $find1$.

Proof by induction:

For n = 1, $find1$ returns the 1st sample ID, which is correct.

For $1 < n \leq N$, lets assume $find1$ algorithms works.

When we apply the $find1$ algorithm for N+1 samples, we divide it into 2 sub problems of size (N+1)/2. This sample size is less than the size of N.

Proof:

$(N + 1)/2 < N$

$N + 1 < 2 * N$

$1 < N$, which is true (from our original assumption).

Hence, the sample size (N+1)/2 is less than N. But through our original assumption, we know that for sample size 1 to N, this algorithm works. Therefore, it works for sample size of N+1 as well.

Solution 2:

Algorithm 4 $find(samples, start, end)$

```
if start == end then
    return [start]
end if

sample1 = $pool(samples, start, floor(end/2))$                    ▷ O(1)
if $test(sample1)$ then                                          ▷ O(1)
    report1 = $find(samples, start, floor(end/2))$
end if

sample2 = $pool(samples, floor(end/2) + 1, end)$                 ▷ O(1)
if $test(sample2)$ then                                          ▷ O(1)
    report2 = $find(samples, floor(end/2) + 1, end)$
end if

return [] + report1 + report2
```

Algorithm gist: For every positive test, we end up doing 2 more tests, until we reach a singular sample.

Consider a binary tree with N leaf nodes.

- Leaf nodes represent the individual sample.
- Node with children, represent the pooling of samples from the children nodes.
- Height of this tree will be $lg(N)$

We know that only 2 leaf nodes are positive for the disease. Therefore, all the ancestors of these 2 leaf nodes will test positive, i.e. $\approx 2 * lg(N)$ nodes. For every positive node, we end up testing at least 2 child nodes. Therefore we have done $\approx 4 * lg(N)$. So the T.C is $O(lg(N))$.

Ruberics:

[+2] For correct pseudo code.

- [+1] for test case: both infected samples lies on either half

- [+1] for test case: both infected samples lies in the same half of array

[+2] For proof of correctness.

[+2] For T.C calculations.

Question 5: Recurrences, recurrences .... **[16]**

Solve each of the recurrences below, and give the best $O(\cdot)$ bound you can for each of them. You can use any theorem you want (Master theorem, Akra-Bazzi, etc.), but please state the theorem in the form you use it. Also, when we write $n/2$, $n/3$, etc. we mean the floor (closest integer less than or equal to the number) of the corresponding quantity. **Please show how you obtained your answer.**

(a) **[4]** $T(n) = 3T(n/3) + n^2$. As the base case, suppose $T(n) = 1$ for $n < 3$.

Answer:

Solution 1: Plug and chug

$$\begin{aligned}
T(n) &= 3T(n/3) + n^2 \\
&\le 3\left(3T\left(n/3^2\right) + (n/3)^2\right) + n^2 && \text{(evaluated } T(n/3)) \\
&= 3^2 T\left(n/3^2\right) + \frac{1}{3}n^2 + n^2 \\
&\le 3^2\left(3T\left(n/3^3\right) + (n/3^2)^2\right) + \frac{1}{3}n^2 + n^2 && \text{(evaluated } T\left(n/3^2\right)) \\
&= 3^3 T\left(n/3^3\right) + \frac{1}{3^2}n^2 + \frac{1}{3}n^2 + n^2 \\
&\dots \\
&\le 3^{\log_3 n}T(1) + \frac{1}{3^{\log_3 n-1}}n^2 + \cdots + \frac{1}{3^2}n^2 + \frac{1}{3}n^2 + n^2 \\
&= n + n^2\left(1 + \frac{1}{3} + \frac{1}{3^2} + \cdots + \frac{1}{3^{\log_3 n-1}}\right) \\
&\le n + n^2\left(\frac{1}{1-\frac{1}{3}}\right) && \text{(geometric series)} \\
&= n + \frac{3}{2}n^2 = O(n^2)
\end{aligned}$$

Notice that for any constant $0 < r < 1$, geometric series $\sum_{k=0}^{\infty} r^k = 1/(1-r)$ is also a constant.

Solution 2: Master theorem

Let $T(n) = aT(n/b) + f(n)$ be a recurrence with constants $a \ge 1, b > 1$ and an asymptotically positive function $f(n)$. The master theorem states that if $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$ and $af(n/b) \le cf(n)$ for some constant $c < 1$, then $T(n) = \Theta(f(n))$. In our case, $a = b = 3$ and $f(n) = n^2$. Because $\log_b a = \log_3 3 = 1$, we have $f(n) = n^2 = \Omega(n^{1+\epsilon})$ for any $0 < \epsilon < 1$. Also, because $af(n/b) = a(n/b)^2 = n^2/3 = f(n)/3$, $af(n/b) \le cf(n)$ for any $\frac{1}{3} \le c < 1$. Thus, $T(n) = \Theta(f(n)) = O(n^2)$.

Rubrics: [+2] Gives the $O(n^2)$ bound. [+1] Gives a bound larger than $O(n^2)$. [+2] Proper reasoning. [-1] Minor arithmetic errors. [-1] Minor flaw in the proof. [-1] Improper use of theorems. [-1] Major stylistic errors (e.g., misuse of terms/notations).

(b) [**6**] $T(n) = 2T(n/2) + T(n/3) + n$. As the base case, suppose $T(0) = T(1) = 1$.

Answer:

First, we solve the case when $n = 2$.

$T(2) = 2T(\lfloor 2/2 \rfloor) + T(\lfloor 2/3 \rfloor) + 2 = 2T(1) + T(0) + 2 = 5$.

We use $T(1) = 1$ and $T(2) = 5$ as the base case because $n = 0$ is inconvenient to find a bound function.

Solution 1: Guess and prove

Assume that $T(n) = \Theta(n^p)$ for some constant $p \ge 1$.

A natural approach to show the upper-bound would be to assume the existence of a constant $c > 0$ such that $T(n) \le cn^p$ for every n. However, the induction with this

assumption does not work out due to the n term in $T(n)$. Instead, we suppose that there exist constants $c, d > 0$ such that $T(n) \leq cn^p - dn$ for all $n \geq 1$.
To suffice the base case, those constants will have the following constraints.

$$\begin{aligned}
T(1) &= 1 \leq c \cdot 1^p - d \cdot 1 = c - d && \implies c \geq d + 1 \\
T(2) &= 5 \leq c \cdot 2^p - d \cdot 2 && \implies p \geq \log_2 \left(\frac{2d + 5}{c} \right)
\end{aligned}$$

For the inductive step, suppose $T(\tilde{n}) \leq c\tilde{n}^p - d\tilde{n}$ for all $1 \leq \tilde{n} < n$. Then

$$\begin{aligned}
T(n) &= 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + n \\
&\leq 2\left[c\left(\frac{n}{2}\right)^p - d\left(\frac{n}{2}\right)\right] + \left[c\left(\frac{n}{3}\right)^p - d\left(\frac{n}{3}\right)\right] + n \\
&= cn^p \left[2\left(\frac{1}{2}\right)^p + \left(\frac{1}{3}\right)^p\right] - dn\left(2\left(\frac{1}{2}\right) + \left(\frac{1}{3}\right)\right) + n \\
&= cn^p \left[2\left(\frac{1}{2}\right)^p + \left(\frac{1}{3}\right)^p\right] + n\left(1 - \frac{4}{3}d\right)
\end{aligned}$$

Because we want to show $T(n) \leq cn^p - dn$, we set our constants as follows:

$$\begin{aligned}
2\left(\frac{1}{2}\right)^p + \left(\frac{1}{3}\right)^p &= 1 \\
n\left(1 - \frac{4}{3}d\right) &\leq -dn && \implies d \geq 3
\end{aligned}$$

Let us define a bijective function $\phi : \mathbb{R} \to \mathbb{R}_{>0}$ as $\phi(x) = 2\left(\frac{1}{2}\right)^x + \left(\frac{1}{3}\right)^x$. Then, we get $p = \phi^{-1}(1) \approx 1.3646$ (unfortunately, there is no simpler form). Also, let $c = 5$, $d = 3$. Make sure that these constants satisfy the base case.

$$\begin{aligned}
T(1) &= 1 \leq c \cdot 1^p - d \cdot 1 = 5 - 3 = 2 \\
T(2) &= 5 \leq c \cdot 2^p - d \cdot 2 = 5 \cdot 2^p - 3 \cdot 2 \approx 6.8752
\end{aligned}$$

Therefore, $T(n) \leq 5n^p - 3n$ for all $n \geq 1$, and $T(n) = O(n^p)$, where $p = \phi^{-1}(1)$ defined above.
Next, we will show that $O(n^p)$ is actually the tight bound. Suppose there exists a constant $c > 0$ such that $T(n) \geq c(n + 1)^p$ for all $n \geq 0$ (here we use $n + 1$ instead of ordinary n to avoid perturbations caused by the floor function). We set $c = 0.2$ and use $n = 0, 1$ as the base case.

$$\begin{aligned} T(0) &= 1 \geq c \cdot (0+1)^p = c = 0.2 \\ T(1) &= 1 \geq c \cdot (1+1)^p = c \cdot 2^p \approx 0.5150 \end{aligned}$$

Inductively, suppose $T(\tilde{n}) \geq c(\tilde{n}+1)^p$ for all $0 \leq \tilde{n} < n$. Then,

$$\begin{aligned} T(n) &= 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + n \\ &\geq 2c\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right)^p + c\left(\left\lfloor \frac{n}{3} \right\rfloor + 1\right)^p + n \\ &\geq 2c\left(\frac{n-1}{2} + 1\right)^p + c\left(\frac{n-2}{3} + 1\right)^p + n && \left(\left\lfloor \frac{n}{2} \right\rfloor \geq \frac{n-1}{2}, \left\lfloor \frac{n}{3} \right\rfloor \geq \frac{n-2}{3}\right) \\ &= 2c\left(\frac{n+1}{2}\right)^p + c\left(\frac{n+1}{3}\right)^p + n \\ &= c(n+1)^p \left[2\left(\frac{1}{2}\right)^p + \left(\frac{1}{3}\right)^p\right] + n \\ &= c(n+1)^p \phi(p) + n \\ &= c(n+1)^p + n && (\phi(p) = 1) \\ &\geq c(n+1)^p \end{aligned}$$

Thus, $T(n) \geq c(n+1)^p$ for every n, and $T(n) = \Theta(n^p) \approx O(n^{1.3646})$.

Solution 2: Akra-Bazzi method

We can apply the Akra-Bazzi method to this recurrence as it has the following form:

$$T(x) = g(x) + \sum_{i=1}^{k} a_i T(b_i x + h_i(x))$$

where $a_i > 0$, $0 < b_i < 1$ for all i, $|g(x)| \in O(n^c)$ for some constant c, and $|h_i(x)| \in O\left(\frac{x}{(\log x)^2}\right)$ for all i. In this problem, $g(x) = x$, $k = 2$, $a_1 = 2$, $a_2 = 1$, $b_1 = 1/2$, $b_2 = 1/3$, and $h_i(x) = 0$ for all i.

Then, $T(x) \in \Theta\left(x^p\left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du\right)\right)$, where p satisfies $\sum_{i=1}^{k} a_i b_i^p = 1$.

Let us define a bijective function $\phi : \mathbb{R} \to \mathbb{R}_{>0}$ as $\phi(x) = \sum_{i=1}^{k} a_i b_i^x = 2\left(\frac{1}{2}\right)^x + \left(\frac{1}{3}\right)^x$. Then, we get $p = \phi^{-1}(1) \approx 1.3646$. Now, simplify the following formula.

$$\begin{aligned}
x^p\left(1+\int_1^x \frac{g(u)}{u^{p+1}}du\right) &= x^p\left(1+\int_1^x \frac{u}{u^{p+1}}du\right)\\
&= x^p\left(1+\int_1^x u^{-p}du\right)\\
&= x^p\left(1+\frac{x^{-p+1}-1}{-p+1}\right)\\
&= \left(1-\frac{1}{-p+1}\right)x^p+\frac{1}{-p+1}\cdot x\\
&\approx 3.7427x^p - 2.7427x \in \Theta(x^p)
\end{aligned}$$

Thus, $T(n)=\Theta(n^p)\approx O(n^{1.3646})$, where $p=\phi^{-1}(1)$ defined above.

Rubrics: [+3] Gives the $O(n^{1.3646})$ bound (we will accept up to $O(n^{1.5})$ as the correct answer). [+1] Gives a bound larger than $O(n^{1.5})$. [+3] Proper reasoning. [-1] Minor arithmetic errors. [-1] Minor flaw in the proof. [-2] Major flaw in the proof. [-1] Improper use of theorems. [-1] Major stylistic errors (e.g., misuse of terms/notations).

(c) **[6]** $T(n)=2(T(\sqrt{n}))^2$. As the base case, suppose $T(1)=4$.

Answer:

First, we solve the case when $n=2$.

$T(2)=2\left(T\left(\lfloor\sqrt{2}\rfloor\right)\right)^2=2(T(1))^2=2\cdot 4^2=2^5$.

Similarly, $T(3)=2^5$.

Solution 1: Plug and chug

Here, we treat n as a real number and disregard the floor function. Please see Solution 2 for a more rigorous argument.

$$
\begin{aligned}
T(n) &= 2(T(\sqrt{n}))^2 \\
&= 2\left[T\left(n^{1/2}\right)\right]^2 \\
&= 2\left[2\left[T\left(n^{1/2^2}\right)\right]^2\right]^2 = 2\cdot 2^2\left[T\left(n^{1/2^2}\right)\right]^{2^2} \\
&= 2\cdot 2^2\left[2\left[T\left(n^{1/2^3}\right)\right]^2\right]^{2^2} = 2\cdot 2^2\cdot 2^{2^2}\left[T\left(n^{1/2^3}\right)\right]^{2^3} \\
&= 2\cdot 2^2\cdot 2^{2^2}\left[2\left[T\left(n^{1/2^4}\right)\right]^2\right]^{2^3} = 2\cdot 2^2\cdot 2^{2^2}\cdot 2^{2^3}\left[T\left(n^{1/2^4}\right)\right]^{2^4} \\
&\cdots \\
&= 2\cdot 2^2\cdot 2^{2^2}\cdots 2^{2(k-1)}\left[T\left(n^{1/2^k}\right)\right]^{2^k} \\
&= 2^{1+2+2^2+\cdots 2^{(k-1)}}\left[T\left(n^{1/2^k}\right)\right]^{2^k} \\
&= 2^{2^k-1}\left[T\left(n^{1/2^k}\right)\right]^{2^k} && \text{(geometric sum)} \\
&= \frac{1}{2}\cdot 2^{2^k}\left[T\left(n^{1/2^k}\right)\right]^{2^k}
\end{aligned}
$$

Now, our task is to find k such that $T(\cdot)$ leads to a base case. It is natural to think $n^{1/2^k} = 1$, but there is no $k \in \mathbb{R}$ that satisfies this equation. Instead, we use $n = 2, 3$ as the base case. Base case $n = 2$ gives:

$$
\begin{aligned}
n^{1/2^k} &= 2 \\
\implies \frac{1}{2^k}\log_2 n &= \log_2 2 = 1 \\
\implies \log_2 n &= 2^k \\
\implies \log_2\log_2 n &= k
\end{aligned}
$$

Similarly, base case $n = 3$ gives $k = \log_2\log_3 n$. But because $T(2) = T(3)$ and $k = \log_2\log_2 n$ grows faster than $k = \log_2\log_3 n$, we just stick to $k = \log_2\log_2 n$.

$$\begin{aligned}
T(n) &= \frac{1}{2} \cdot 2^{2^k} \left[T\left(n^{1/2^k}\right)\right]^{2^k} \\
&= \frac{1}{2} \cdot 2^{2^{\log_2 \log_2 n}} \left[T\left(2\right)\right]^{2^{\log_2 \log_2 n}} \\
&= \frac{1}{2} \cdot 2^{2^{\log_2 \log_2 n}} \left[2^5\right]^{2^{\log_2 \log_2 n}} \\
&= \frac{1}{2} \cdot 2^{\log_2 n} \left[2^5\right]^{\log_2 n} \\
&= \frac{1}{2} \cdot 2^{\log_2 n} \left[2^{\log_2 n}\right]^5 \\
&= \frac{1}{2} \cdot n \cdot n^5 = \frac{n^6}{2} = O(n^6)
\end{aligned}$$

Solution 2: Change of variables

Now, consider a function $S(k) := T\left(2^{2^k}\right)$ for $k \in \mathbb{Z}_{\geq 0}$ with the base case $S(0) = T\left(2^{2^0}\right) = T(2)$. We have:

$$\begin{aligned}
S(k) &= T\left(2^{2^k}\right) \\
&= 2\left(T\left(\left\lfloor\sqrt{2^{2^k}}\right\rfloor\right)\right)^2 \\
&= 2\left(T\left(\left\lfloor 2^{2^{k-1}}\right\rfloor\right)\right)^2 \\
&= 2\left(T\left(2^{2^{k-1}}\right)\right)^2 && \text{(the inside of the floor is always integral)} \\
&= 2\left(S(k-1)\right)^2
\end{aligned}$$

Furthermore, we define a function $f(k) := \log_2 S(k)$ for $k \in \mathbb{Z}_{\geq 0}$. If $k > 0$, then:

$$\begin{aligned}
f(k) &= \log_2 S(k) = \log_2\left(2(S(k-1))^2\right) = \log_2 2 + 2\log_2 S(k-1) \\
&= 2f(k-1) + 1
\end{aligned}$$

And $f(0) = \log_2 S(0) = \log_2 T(2) = \log_2 2^5 = 5$.
Let's solve this recurrence with plug and chug.

$$\begin{aligned}
f(k) &= 2f(k-1) + 1 \\
&= 2\left(2f(k-2) + 1\right) + 1 = 2^2 f(k-2) + 2 + 1 \\
&= 2^2\left(2f(k-3) + 1\right) + 2 + 1 = 2^3 f(k-3) + 2^2 + 2 + 1 \\
&\cdots \\
&= 2^k f(0) + 2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2 + 1 \\
&= 2^k \cdot 5 + \sum_{i=0}^{k-1} 2^i \\
&= 5 \cdot 2^k + 2^k - 1 && \text{(geometric sum)} \\
&= 6 \cdot 2^k - 1
\end{aligned}$$

Thus, $S(k) = 2^{f(k)} = 2^{6\cdot 2^k - 1} = \frac{1}{2} \cdot \left(2^{2^k}\right)^6$.

Finally, because $S(k) = T\left(2^{2^k}\right)$, we obtain $T(n) = \frac{1}{2}n^6$ when $n \in \left\{2^{2^k} \mid k \in \mathbb{Z}_{\geq 0}\right\}$.

This remains us to prove that $T(n) = O(n^6)$ for all $n > n_0$ for some n_0. We will show that $T(n) \leq \frac{1}{2}n^6$ for all $n \geq 2$ by induction on n.

Base case $n = 2, 3$: $T(2) = 2^5 \leq \frac{1}{2} \cdot 2^6$; $T(3) = 2(T(1))^2 = 2^5 \leq \frac{1}{2} \cdot 3^6$.

Note that we need to check $T(3)$ because not all numbers end up with $T(2)$. For example, $T(9)$ requires to compute $T(3)$ and $T(1)$, but not $T(2)$.

Inductive step: Suppose $T(\tilde{n}) \leq \frac{1}{2}\tilde{n}^6$ for all $2 \leq \tilde{n} < n$. Then,

$T(n) = 2\left(T(\lfloor\sqrt{n}\rfloor)\right)^2 \leq 2\left(\frac{1}{2}\left(\lfloor\sqrt{n}\rfloor\right)^6\right)^2 = \frac{1}{2}\left(\lfloor\sqrt{n}\rfloor\right)^{12} \leq \frac{1}{2}\left(\sqrt{n}\right)^{12} = \frac{1}{2}n^6$.

Thus, $T(n) = O(n^6)$ holds, and because $T(n) = \Theta(n^6)$ when $n \in \left\{2^{2^k} \mid k \in \mathbb{Z}_{\geq 0}\right\}$, $O(n^6)$ is the tight bound.

Rubrics: [+3] Gives the $O(n^6)$ bound. [+1] Gives a bound larger than $O(n^6)$. [+3] Proper reasoning. [-1] Minor arithmetic errors. [-1] Minor flaw in the proof. [-2] Major flaw in the proof. [-1] Improper use of theorems. [-1] Major stylistic errors (e.g., misuse of terms/notations).

Question 6: Dynamic arrays: is doubling important? .... **[6]**

Consider the 'add' procedure for dynamic arrays (DA) that we studied in class. Every time the add operation was called and the array was full, the procedure created a new array of twice the size and copied all the elements, and then added the new element.

Suppose we consider an alternate implementation, where the array size is always a multiple of some integer, say 32. Every time the add procedure is called and the array is full (and of size n), suppose we create a new array of size $n + 32$, copy all the elements and then add the new element.

For this new add procedure, analyze the asymptotic running time for N consecutive add operations.

Answer.

Assume we have an array filled with n values, and our next insertion will require the array to be grown by 32. We can then insert 31 elements into the array in constant time. These operations have the following costs:

- Allocation of size $n + 32$
- Copy of n elements
- Insertion of 32 elements

for $N = 32$ consecutive insert operations, we have a cost of

$$n + 32 + n + 32 = 2(n + 32) \in O(n)$$

On average, we have $2(n + 32)/32 \in O(n)$ time for each insertion. As long as $n \gg 32$, our average insertion time is linear in the size of the array. (Note that this is much larger than $O(1)$, which is what one obtains if we create a new array only when the size doubles.)

Alternate: If we assume $N = n$ (we begin with an empty array and insert n elements), we have a cost similar to the following, where each k-th term contains an allocation/copy of size $32k$ and 32 constant-time insertions:

$$\begin{aligned}
T(n) &= (32+32) + (64+32) + (96+32) + ... + (n+32) \\
&= \sum_{i=1}^{k=n/32} (32i + 32) = n + 32 \sum_{i=1}^{k=n/32} i \\
&= \frac{(n+32)n}{64} + n \\
T(n) &\in O(n^2)
\end{aligned}$$

Inserting n elements takes $O(n^2)$, or $O(n)$ per element.

Rubric: 2 points for correct cost of growing array. 2 points for cost of insertion. 2 points for correct evaluation of sum and asymptotic running time.