1 Recurrences

1.1 Substitution method

A lot of things in this class reduce to induction. In the substitution method for solving recurrences we

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

1.1.1 Example

Recurrence: $T(1) = 1$ and $T(n) = 2T(\lfloor n/2 \rfloor) + n$ for $n > 1$.

We guess that the solution is $T(n) = O(n \log n)$. So we must prove that $T(n) \leq cn \log n$ for some constant c. (We will get to n_0 later, but for now let's try to prove the statement for all $n \geq 1$.)

As our inductive hypothesis, we assume $T(n) \leq cn \log n$ for all positive numbers less than n. Therefore, $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor))$, and

$$\begin{aligned}
T(n) &\leq 2(c\lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\
&\leq cn \log(n/2) + n \\
&= cn \log n - cn \log 2 + n \\
&= cn \log n - cn + n \\
&\leq cn \log n \quad (\text{for } c \geq 1)
\end{aligned}$$

Now we need to show the base case. This is tricky, because if $T(n) \leq cn \log n$, then $T(1) \leq 0$, which is not a thing. So we revise our induction so that we only prove the statement for $n \geq 2$, and the base cases of the induction proof (which is not the same as the base case of the recurrence!) are $n = 2$ and $n = 3$. (We are allowed to do this because asymptotic notation only requires us to prove our statement for $n \geq n_0$, and we can set $n_0 = 2$.)

We choose $n = 2$ and $n = 3$ for our base cases because when we expand the recurrence formula, we will always go through either $n = 2$ or $n = 3$ before we hit the case where $n = 1$. So proving the inductive step as above, plus proving the bound works for $n = 2$ and $n = 3$, suffices for our proof that the bound works for all $n > 1$.

Plugging the numbers into the recurrence formula, we get $T(2) = 2T(1) + 2 = 4$ and $T(3) = 2T(1) + 3 = 5$. So now we just need to choose a c that satisfies those constraints on $T(2)$ and $T(3)$. We can choose $c = 2$, because $4 \leq 2 \cdot 2 \log 2$ and $5 \leq 2 \cdot 3 \log 3$.

Therefore, we have shown that $T(n) \leq 2n \log n$ for all $n \geq 2$, so $T(n) = O(n \log n)$.

1.1.2 Warnings

Warning: Using the substitution method, it is easy to prove a weaker bound than the one you're supposed to prove. For instance, if the runtime is $O(n)$, you might still be able to substitute cn^2 into the recurrence and prove that the bound is $O(n^2)$. Which is technically true, but don't let it mislead you into thinking it's the best bound on the runtime. People often get burned by this on exams!

Warning: You must prove the exact form of the induction hypothesis. For example, in the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + n$, we could falsely "prove" $T(n) = O(n)$ by guessing $T(n) \leq cn$ and then arguing $T(n) \leq 2(c\lfloor n/2 \rfloor) + n \leq cn + n = O(n)$. Here we needed to prove $T(n) \leq cn$, not $T(n) \leq (c+1)n$. Accumulated over many recursive calls, those "plus ones" add up.

1.2 Recursion tree

A recursion tree is a tree where each node represents the cost of a certain recursive subproblem. Then you can sum up the numbers in each node to get the cost of the entire algorithm.

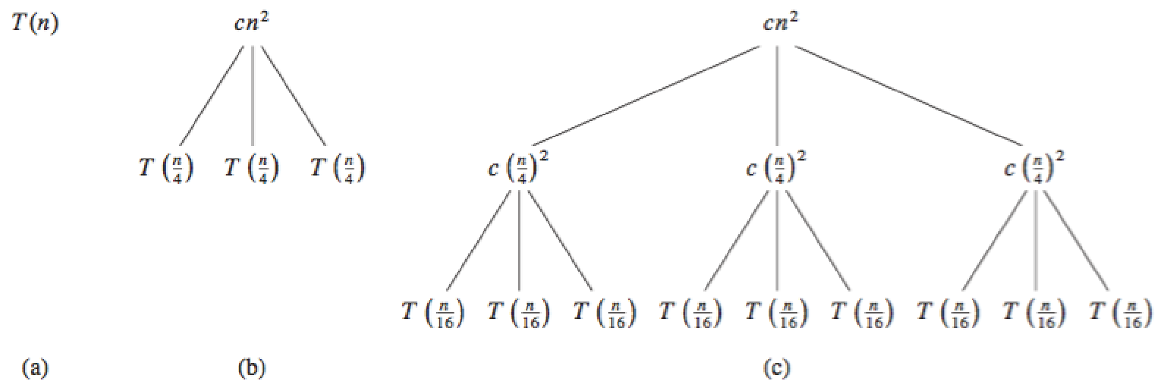
Note: We would usually use a recursion tree to generate possible guesses for the runtime, and then use the substitution method to prove them. However, if you are very careful when drawing out a recursion tree and summing the costs, you can actually use a recursion tree as a direct proof of a solution to a recurrence.

If we are only using recursion trees to generate guesses and not prove anything, we can tolerate a certain amount of "sloppiness" in our analysis. For example, we can ignore floors and ceilings when solving our recurrences, as they usually do not affect the final guess.

1.2.1 Example

Recurrence: $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

We drop the floors and write a recursion tree for $T(n) = 3T(n/4) + cn^2$.



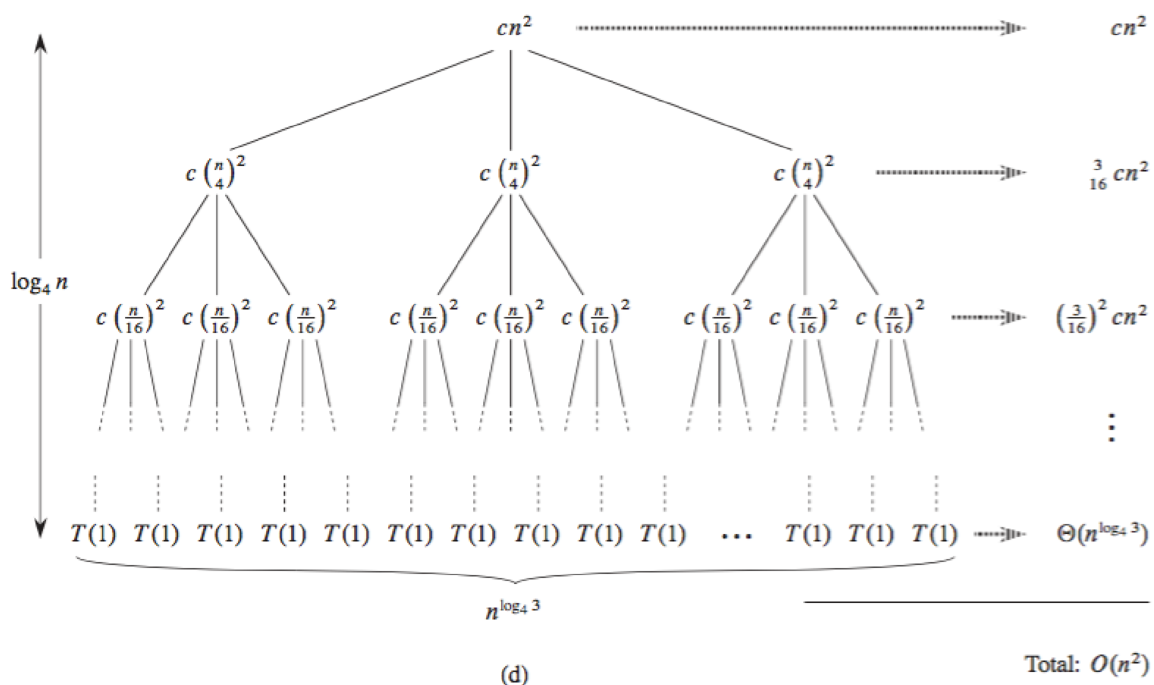


Figure 4.5 Constructing a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

The top node has cost cn^2, because the first call to the function does cn^2 units of work, aside from the work done inside the recursive subcalls. The nodes on the second layer all have cost $c(n/4)^2$, because the functions are now being called on problems of size $n/4$, and the functions are doing $c(n/4)^2$ units of work, aside from the work done inside their recursive subcalls, etc. The bottom layer (base case) is special because each of them contribute $T(1)$ to the cost.

Analysis: First we find the height of the recursion tree. Observe that a node at depth i reflects a subproblem of size $n/4^i$. The subproblem size hits $n = 1$ when $n/4^i = 1$, or

$i = \log_4 n$. So the tree has $\log_4 n + 1$ levels.

Now we determine the cost of each level of the tree. The number of nodes at depth i is 3^i. Each node at depth $i = 0, 1, \ldots, \log_4 n - 1$ has a cost of $c(n/4^i)^2$, so the total cost of level i is $3^i c(n/4^i)^2 = (3/16)^i cn^2$. However, the bottom level is special. Each of the bottom nodes contribute cost $T(1)$, and there are $3^{\log_4 n} = n^{\log_4 3}$ of them.

So the total cost of the entire tree is

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \end{aligned}$$

The left term is just the sum of a geometric series. So $T(n)$ evaluates to

$$\frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3})$$

This looks complicated but we can bound it (from above) by the sum of the infinite series

$$\sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) = \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3})$$

Since functions in $\Theta(n^{\log_4 3})$ are also in $O(n^2)$, this whole expression is $O(n^2)$. Therefore, we can guess that $T(n) = O(n^2)$.

1.2.2 Back to the substitution method

Now we can check our guess using the substitution method. Recall that the original recurrence was $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. We want to show that $T(n) \leq dn^2$ for some constant $d > 0$. By the induction hypothesis, we have that $T(\lfloor n/4 \rfloor) \leq d\lfloor n/4 \rfloor^2$. So using the same constant $c > 0$ as before, we have

$$\begin{aligned} T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leq 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\leq dn^2 \quad \text{(when } c \leq (13/16)d, \text{ i.e. } d \geq (16/13)c) \end{aligned}$$

Note that we would also have to identify a suitable base case and prove the recurrence is true for the base case.

2 Universal hashing

In a hash table, you assign each element in the data structure to one of b buckets. The procedure that assigns elements to buckets is called a hash function. In order to search for an element in a hash table, you simply need to use the hash function to compute the bucket that the element was assigned to, and then search for the key within that bucket.

While executing the hash function (normally) takes constant time, the time it takes to search for an element in a bucket depends on the number of items in each bucket. For this reason, it is important to choose a hash function that sort of "evenly distributes" items among buckets.

The problem is, if you say what the hash function is in advance, it is possible for a malicious adversary (or a bad programmer) to deliberately insert items that they know will all hash to the same bucket.

This is particularly true if the number of distinct items in the **universe** $\mathcal{U}$ (i.e. the set of all possible hashable items that might potentially be inserted) is high compared to the number of buckets. If there are 31 distinct items in the universe, and only 10 buckets, then by the pigeonhole principle, at least one bucket must have more than 3 elements that might hash to it. In practice, the hash function may still achieve good performance and evenly distribute the items that are actually inserted. But a particularly malicious programmer would choose to insert the 4 elements that he knows would all hash to one bucket, and slow the hash table down.

2.1 Universal hashing

Universal hash families protect us from this nonsense. The idea is that you have a collection of hash functions, and every time you create a new hash table, you pick one of the hash functions at random, and use that to hash all of the keys in the table. Because the hash function is chosen at runtime, and different hash functions in the family have bad performance on different inputs, it's impossible for a particularly bad programmer to deliberately insert items that will give him the worst case performance.

Specifically, universal hash families give you an upper bound on the probability of two elements colliding with each other. A hash family $\mathcal{H}$ is **universal** if for any pair of elements in the universe x_i and x_j, the fraction of hash functions in $\mathcal{H}$ that hash x_i and x_j to the same bucket is smaller than 1/(number of buckets).

In other words, for all $x_i, x_j \in \mathcal{U}$,

$$Pr_{h \sim \mathcal{H}}[h(x_i) = h(x_j)] \leq 1/b$$

This means that if you pick a hash function at random, you would not expect there to be a lot of hash collisions in the table.

2.2 Proof that this guarantee constrains the expected number of items in a bucket

Formally, we can prove this as follows.

Theorem 11.3 (CLRS). Suppose that a hash function h is chosen uniformly at random from a universal collection of hash functions and has been used to hash n keys into a table with b buckets, using chaining to resolve collisions. If key k is not in the table, then the expected length $E[n_{h(k)}]$ of the list that key k hashes to is at most n/b. If key k is in the table, then the expected length of the list containing key k is at most $1 + n/b$.

Proof: To find $n_{h(k)}$, the length of the list that k hashes to, we must count the number of keys (other than k) that collide with k. Let $I_{h(k)=h(l)}$ be the indicator random variable that is 1 if l hashes to the same bucket as k, and 0 otherwise. Then $E[n_{h(k)}] = E[\sum_{l \neq k} I_{h(k)=h(l)}] = \sum_{l \neq k} E[I_{h(k)=h(l)}] = \sum_{l \neq k} P(h(k) = h(l)) \leq \sum_{l \neq k} 1/b$.

Now if key k is not in the table, then there are n terms in that sum, since there are n keys in the table total, and thus n keys that aren't equal to key k. The sum represents all items in the list, so the expected length of the list is $\leq n/b$. However, if k is in the table, then there are $n - 1$ terms in that sum (because one of the n keys is equal to k, so should not be counted), but the sum does not account for the fact that k is in the list. So we get $(n - 1)/b + 1 < 1 + n/b$.

2.3 An example of a universal hash family

You saw two examples of universal hash families on your homework, in question 4a and 4b. However, these universal hash families are impractical, since each hash function requires many bits to store. More often, we choose a large prime number p, large enough that every possible key k is in the range 0 to $p - 1$, and use the universal hash family defined by the set of functions

$$h_{xy}(k) = ((xk + y) \mod p) \mod b$$

for all $x \in \{1, 2, \ldots, p - 1\}$ and y in $\{0, 1, 2, \ldots, p - 1\}$.

In order to use this universal hash family, we only need to choose x and y at random, and then use the function h_{xy} to hash all the keys k in the table. At runtime, this only requires us to store the numbers x, y, p, and b, which allows us to represent the hash function in a compact form.

Theorem 11.5 in CLRS shows that this hash family is, in fact, universal.

Proof: Consider two distinct keys k and l. For a given hash function h_{xy}, we let

$$\begin{aligned} r &\equiv xk + y \mod p \\ s &\equiv xl + y \mod p \end{aligned}$$

We will show that there is a one-to-one correspondence between the pairs (x, y) where $x \neq 0$, and the pairs (r, s) where $r \neq s$. This means that if you choose your hash function uniformly

at random, this is equivalent to choosing values of r and s uniformly at random (where r and s are distinct).

A hash collision happens when $r \equiv s \mod b$. So the probability of a hash collision between (distinct) keys k and l is the probability that $r \equiv s \mod b$ when r and s are randomly chosen as distinct values mod p. If you do the math on this, you get that that probability is $\leq 1/b$, which shows this family is a universal hash family.

So how do we show that there is a one-to-one correspondence? First of all, if you know what x and y are, that uniquely determines r and s. So that's one direction. For the other direction, you are given r and s, and if you can solve for x and y, then you know that the values uniquely determine each other.

So if we are given r and s, then we can subtract the equations, getting $r - s \equiv x(k - l) \mod p$. Now when p is prime, the set of integers mod p is a field, which means that every nonzero element has a multiplicative inverse that is also in the field. (We won't prove that here.) So if we multiply both sides by the multiplicative inverse of $k - l$ (since $k \neq l$), we can solve for x. Once we have x, we can solve for y, using the equation $y \equiv r - xk$.

(Note that in these correspondences, hash functions where $x = 0$ correspond to hash functions where $r = s$. Because if $r = s$, then multiplying by $(k - l)^{-1}$ gives you $x = 0$, and if $x = 0$, then you get $r = s = y$. However, we don't like hash functions where $x = 0$ because they hash all the keys to the same bucket, so we exclude them from our hash family.)