Recursive Algorithms

Reductions

Reduction is the single most common technique used in designing algorithms. Reducing one problem from problem X to problem Y means writing an algorithm for X that uses an algorithm for Y as a black box or subroutine.

Simplify and Delegate

Recursion is a powerful kind of recursion and can be described loosely as

1. If the given instance can be solved directly → solve directly
2. Otherwise, reduce to one or more simpler instances of the same problem
3. *Imagine* someone else is solving the simpler problem

Proof of correctness by induction

The recursive reductions must lead to an elementary **base case** that can be solved by some other method or it will *loop* forever.

Reduce to one or more smaller instances of solving the **exact same problem**.

*E.g. Cannot make recursive call to **multiplying** two integers when the original problem wants to compute **square** of a number*

Towers of Hanoi

How many moves does it require to transfer n disks of different sizes from rod 1 to rod 3 with the following rules

1. Move 1 disk at a time
2. Never place a larger disk on a smaller disk

No solution for 2 rods

Recursion

There must be a step where the biggest disk n moves to rod 3.

What must happen before that?

All disks smaller than disk n must be moved to Rod 2. Rod 2 will now have n-1 disks in the same fashion that Rod 1 had n disks stacked up. Notice that this is a smaller instance of the **same** problem.

What must happen after this?

This reduces to a problem of moving all n-1 disks from Rod 2 to Rod 3 using Rod 1, whereas previously, it was moving n disks from Rod 1 to Rod 3 using Rod 2.

Now, we have an idea to solve the problem via recursion. We don't have to tell the algorithm how to move the n-1 disks before moving the nth disk as this is taken care of by **recursion**.

To prove the recursion, we first look at the base case where $n = 0$

Base case: Move 0 disks from Rod 1 to Rod 3 using Rod 2

As you can see, there is nothing to do at this step and the base case is solved.

Algorithm

$HANOI(n, 1, 3, 2)$ which is to move n disks from Rod 1 to Rod 3 using Rod 2.

If $n > 0$

1. $HANOI(n-1, 1, 2, 3)$ → Move $n-1$ disks from Rod 1 to Rod 2
2. Move disk n from Rod 1 to Rod 3
3. $HANOI(n-1, 2, 3, 1)$ → Move the remaining disks on Rod 2 to Rod 3

Let $T(n)$ denote the number of moves required to transfer n disks → Running time of algorithm

$$T(0) = 0$$

$$T(n) = 2T(n-1) + 1$$

We have already seen the solution to the recurrence is

$$T(n) = 2^n - 1$$

Exercise: Show that this algorithm takes the fewest possible moves.

——PROBABLY NOT LEGIT——

Claim: $T(n) < 2^n - 1$

Base case: $n = 0$

$$T(0) < 2^0 - 1$$
$$0 < 0 \quad ✗$$

——PROBABLY NOT LEGIT——

——Lecture Proof——

Claim: Assume $S(n)$ exists where $S(n)$ is the number of steps the best possible algorithm takes in moving n disks from any rod to another.

As shown earlier, to solve Hanoi, we must first move $n-1$ disks to Rod 2 ($S(n-1)$), before moving the nth disk to Rod 3 (1). Afterwords, it is a matter of calling the same algorithm on the disks in Rod 2 to shift them to Rod 3 ($S(n-1)$).

$$S(n) \geq S(n-1) + 1 + S(n-1)$$

$$S(n) \geq 2S(n-1) + 1$$

Now we know the form $2S(n-1) + 1$ reduces to $2^n - 1$ and hence,

$$S(n) \geq 2^n - 1$$

Proof by Induction:

When $n = 0$, there is no disk to be moved and 0 steps are needed. $S(0) = 0$, $2^0 - 1 = 0$

$$S(0) \geq 0 \quad ✓$$

Assume that $S(k) \geq 2^k - 1$ for $1 \leq k \leq n-1$

$$\begin{aligned} S(n) &\geq 2S(n-1) + 1 \\ &\geq 2(2^{n-1} - 1) + 1 \\ &= 2^n - 2 + 1 \\ &= 2^n - 1 \quad \checkmark \end{aligned}$$

By induction, $S(n) \geq 2^n - 1$ where $S(n)$ is the best possible algorithm for this problem. Showing that the lower bound for the problem is $2^n - 1$ steps.

——Lecture Proof——

Mergesort

Mergesort splits the array into 2 and the same mergesort algorithm is called on the two smaller arrays. After receiving two sorted arrays, it scans simultaneously through the two lists, adding the smaller of the two.

The recurrence relation is

$$T(n) = \begin{cases} \Theta(1) & if\ n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n) & otherwise \end{cases}$$

Solution has been shown previously as $T(n) = \Theta(n\ log\ n)$

Quicksort

1. Choose a pivot element from array
2. Partition array into 3 subarray. One smaller than the pivot, the pivot, and one larger than the pivot
3. Recursively quicksort the first and last subarray

The partition returns the **rank r** of the pivot element chosen as we know the number of elements smaller than it.

Analysis

$$T(n) = T(r-1) + T(n-r) + O(n)$$

If we managed to choose the pivot to be the *median* element of the array, we would have $r = \lceil n/2 \rceil$, giving us

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil - 1) + T(\lfloor n/2 \rfloor) + O(n) \\ &\leq 2T(n/2) + O(n) \\ &= O(n\ log\ n) \qquad\qquad By\ Master's\ Theorem \end{aligned}$$

Pattern

Mergesort and Quicksort follow the general pattern of **divide and conquer**

1. **Divide** the given instance into several smaller instance of the exact same problem
2. Make a **recursive call** to the same problem for smaller instances
3. **Combine** the solutions for the smaller instances into the final solution for the given instance

In Mergesort, the Divide step takes $O(1)$ and Combine takes $O(n)$
In Quicksort, the Divide step takes $O(n)$ and Combine takes $O(1)$

Algorithm	Divide	Combine
Mergesort	$O(1)$	$O(n)$
Quicksort	$O(n)$	$O(1)$

Linear Time Selection

Problem: Selecting the k-th smallest element in an n-element array

Idea: Sort and output k-th smallest → $O(n\ log\ n)$

Better Idea: Modify Quicksort to give Quickselect that runs the partition step and only recurse on relevant subarray

Analysis

$$T(n) \leq max_{1 \leq r \leq n}\ max\{T(r-1), T(n-r)\} + O(n)$$

Letting l denote the length of the recursive subproblem:

$$T(n) \leq max_{0 \leq l \leq n-1}\ T(l) + O(n)$$

This means that if we always choose the largest or smallest element in the array, the recurrence will simplify to $T(n) = T(n-1) + O(n)$ which implies $T(n) = O(n^2)$

```
Find the 4th smallest element (6). In this case n = 5
[3,6,2,9,5]
Suppose we select element 2, our subarrays will be
Smaller: []
Pivot: 2
Larger: [3,6,9,5]
We have to recurse on the larger array of size n-1
```

We can see that it would be a linear path of selecting either the largest / smallest element with $O(n)$ work done every time for n steps.

Good Pivot

If the pivot is chosen such that the **subproblem** that the algorithm calls recursively has size $a \cdot n$ for some constant $a < 1$, then the time complexity is given by the recurrence

$$T(n) = T(a \cdot n) + O(n)$$

By Master's Theorem with $a = a$, $b = 1$, $f(n) = O(n) = Cn$ for some constant $C > 0$

$$\begin{aligned} a \cdot f(n/b) &= aCn \\ &\leq \alpha Cn\ for\ some\ \alpha < 1 \end{aligned}$$

Thus, $T(n) = O(f(n)) = O(n)$

Goal: Find a set of elements of size $a \cdot n$ for some constant $a < 1$ in $O(n)$ time such that the set contains the k-th smallest element.

Choosing a good pivot

Idea by Blum-Floyd-Pratt-Rivest-Tarjan

1. Partition input set into groups of 5 elements each arbitrarily. If n is not multiple of 5, add a few ∞ elements and remove at the end.
2. Sort each group and find the median (3rd smallest element)
3. The pivot is the median of these medians i.e $\frac{n}{10}th$ smallest element among $\frac{n}{5}$ elements. We will recursively call the Quickselect algorithm on the medians of the set to find the median.

Walkthrough

Warning *This portion mostly assumes n is a multiple of 5 which is not recommended by the professor and I also drop some constants.*

Suppose we have n elements where n is a multiple of 5.

We will get $\frac{n}{5}$ number of sets which we will call s.

We will sort all s sets and take the median from each set, giving us s medians.

Out of these s medians, we sort them and select the median of medians which we will call m.

By selecting the median m, we will have $\frac{s-1}{2}$ sets (assume s is odd) with medians smaller than m and another $\frac{s-1}{2}$ sets with medians larger than m.

For the $\frac{s-1}{2}$ sets with medians smaller than m, they each contain 3 elements smaller than m which will give us $\frac{3s-3}{2}$ elements smaller than m. We also need to add the 2 elements in the set that m is in which will give us $\frac{3s+1}{2}$ elements in total.

$$\begin{aligned}\frac{3s+1}{2} &= \frac{1}{2}(3\frac{n}{5}+1) \\ &= \frac{3n}{10}+0.5 \\ &\geq \frac{3n}{10}\end{aligned}$$

We can reason the same for elements larger than m as well.

Now, the rank of the pivot r is between $3n/10$ and $7n/10$

So the recursive call is made to $max(r-1, n-r) < 7n/10$ elements

The algorithm requires $T(n/5)+O(n)$ before making the recursive call instead of the desired $O(n)$

More formally, the recurrence is

$$T(n) \leq T(n'/5) + T(7n'/10) + O(n)$$

where $n' < n+5$ is the smallest integer greater than n that is a multiple of 5. Here, $T(n'/5)$ is the time required to find the median in your sets of 5 elements. $O(n)$ is the time required to partition the array after finding the pivot. $T(7n'/10)$ is the time required to recurse on the larger subarray after partitioning.

Analysis

Using domain substitution, $S(n) = T(n+10)$, we get for some constant C

$$S(n) \leq S(7n/10) + S(n/5) + Cn$$

This gives us a recursion tree and summing up the work done at each level.

Level 0: n

Level 1: $\frac{n}{5} + \frac{7n}{10} = \frac{9n}{10}$

Level 2: $(\frac{1}{5}+\frac{7}{10})\frac{n}{5} + (\frac{1}{5}+\frac{7}{10})\frac{7n}{10} = \frac{9}{10}\cdot\frac{9}{10}\cdot n$

Notice that this is a geometric series with ratio $r = \frac{9}{10}$

$$
\begin{aligned}
S(n) &\leq C \cdot n + \frac{9}{10} \cdot C \cdot n + \frac{9^2}{10^2} \cdot C \cdot n + \dots \\
&\leq \frac{Cn}{1 - \frac{9}{10}} \\
&= 10Cn \\
&= O(n)
\end{aligned}
$$

Claim: $S(n) \leq 20Cn$ for $n \geq 1$

At $n = 1$, the time taken would be constant so it is true.

Assuming the statement $S(k) \leq 20Ck$ is true for $2 \leq k \leq n - 1$

$$
\begin{aligned}
S(n) &\leq S(7n/10) + S(n/5) + Cn \\
&\leq 14Cn + 4Cn + Cn \\
&\leq 19Cn \\
&\leq 20Cn \quad \checkmark
\end{aligned}
$$

$$
T(n) = S(n - 10) = 10C(n - 10) = O(n)
$$

Question: Groups of 3

$s = \frac{n}{3}$, $\frac{s-1}{2}$ sets with median smaller than m with each set containing 2 elements smaller than m. This gives us $s - 1$ elements smaller than m and we have to add the 1 element smaller than m in the set that m was in. This will give us

$$
n/3 < r < 2n/3
$$

Here, we get

$$
T(n) \leq T(n'/3) + T(2n'/3) + Cn
$$

where $n' < n + 5$ is the smallest integer greater than n that is a multiple of 3.

We use domain substitution $S(n) = T(n + 6)$, giving us

$$
S(n) \leq S(2n/3) + S(n/3) + Cn
$$

This gives us a recursion tree where if we sum up the levels we get

Level 0: n

Level 1: $\frac{n}{3} + \frac{2n}{3} = n$

Level 2: $(\frac{1}{3} + \frac{2}{3})\frac{n}{3} + (\frac{1}{3} + \frac{2}{3})\frac{2n}{3} = n$

Given the longest branch is of $log_{3/2}n$ height, we most likely have $O(nlog_{3/2}n)$

Question: Groups of 7

$s = \frac{n}{7}$, $\frac{s-1}{2}$ sets with median smaller than m with each set containing 4 elements smaller than m. This gives us $2s - 2$ elements smaller than m and we have to add the 3 element smaller than m in the set that m was in. This will give us

$$
\begin{aligned}
2s + 1 &= \frac{2n}{7} + 1 \\
&\geq \frac{2n}{7}
\end{aligned}
$$

And we get rank r in range of $\frac{2n}{7} < r < \frac{5n}{7}$

We get the recurrence relation

$$
T(n) \leq T(n'/7) + T(5n'/7) + Cn
$$

where $n' < n + 7$ is the smallest integer greater than n that is a multiple of 7.

We use domain substitution $S(n) = T(n+14), giving us

$$S(n) \leq S(5n/7) + S(n/7) + Cn$$

This gives us a recursion tree where if we sum up the levels we get

Level 0: n

Level 1: $\frac{n}{7} + \frac{5n}{7} = \frac{6n}{7}$

Level 2: $(\frac{1}{7} + \frac{5}{7})\frac{n}{7} + (\frac{1}{7} + \frac{5}{7})\frac{5n}{7} = \frac{6}{7} \cdot \frac{6}{7} \cdot n$

Notice that this is a geometric series with ratio $r = \frac{6}{7}$

$$\begin{aligned} S(n) &\leq C \cdot n + \frac{6}{7} \cdot C \cdot n + \frac{6^2}{7^2} \cdot C \cdot n + \dots \\ &\leq \frac{Cn}{1 - \frac{6}{7}} \\ &= 7Cn \\ &= O(n) \end{aligned}$$

Question: Groups of k where k is odd

$s = \frac{n}{k}$, $\frac{s-1}{2}$ sets with median smaller than m with each set containing $(k+1)/2$ elements smaller than m.

This gives us $\frac{(k+1)(s-1)}{4}$ elements smaller than m and we have to add the $(k-1)/2$ elements smaller than m in the set that m was in. This will give us

$$\begin{aligned} \frac{(k+1)(s-1)}{4} + (k-1)/2 &= \frac{1}{4}(ks - k + s - 1 + 2k - 2) \\ &= \frac{1}{4}(\frac{n}{k}(k+1) + k - 3) \\ &= \frac{1}{4}(n + \frac{n}{k} + k - 3) \end{aligned}$$

Ignoring constants will give us

$$\frac{1}{4}(n + \frac{n}{k} + k - 3) = \frac{nk + n}{4k}$$

If we very loosely convert this into the largest possible ratio we get

$$\begin{aligned} n - \frac{nk + n}{4k} &= \frac{4kn - nk - n}{4k} \\ &= \frac{n(3k-1)}{4k} \end{aligned}$$

That is to say $\frac{n(k+1)}{4k} < r < \frac{n(3k-1)}{4k}$

Each level's work done can be, again *loosely*, represented as

Level 0: n

Level 1: $\frac{n}{k} + \frac{n(3k-1)}{4k} = \frac{3(kn+n)}{4k} = \frac{3}{4}n + \frac{3}{4k}n$

For the geometric series to converge, we must have $\frac{3}{4k}n < \frac{1}{4}n$ and therefore, $k > 3$

Example: Integer Multiplication

The straightforward algorithm for multiplying two n-digi integers requires n^2 multiplications and $O(n)$ additions and runs in $O(n^2)$

We want to try to get a recursive algorithm by exploiting

$$(10^{n/2}a + b)(10^{n/2}c + d) = 10^n ac + 10^{n/2}(ad + bc) + bd$$

Think about splitting an integer into 2 parts and multiplying the smaller parts instead.

```
Given n = 8 and two numbers 12345678 and 87654321

This will give us
a = 1234
b = 5678
c = 8765
d = 4321

Now we need
a*c = 1234 * 8765
a*d = 1234 * 4321
b*c = 5678 * 8765
b*d = 5678 * 4321

Where each can be recursed as two numbers where n = 4
```

This gives us the recurrence relation

$$T(n) = 4T(\lceil n/2 \rceil) + O(n)$$

And using Master's Theorem, we get $T(n) = O(n^{log_2 4}) = O(n^2)$

This offers `0` improvement but we can observe the following identity.

$$\begin{aligned}(a + b)(c + d) &= ac + (ad + bc) + bd \\ ad + bc &= (a + b)(c + d) - ac - bd\end{aligned}$$

Now, after computing ac and bd, we only have to compute one more term $(a + b)(c + d)$ in order to obtain $ad + bc$. This reduces the number of multiplications we have to do by 1.

In $ad + bc$, we first multiply $a \cdot d$ and then $b \cdot c$ which gives us **two** multiplications.

In $(a + b)(c + d) - ac - bd$, we already have ac and bd from computation so we only need to add $a + b$ and $c + d$ which is cheaper than multiplication. Finally, only **one** more multiplication is needed for $(a + b) \cdot (c + d)$

This will give us

$$T(n) \leq 3T(\lceil n/2 \rceil) + O(n)$$

By Master's Theorem, $T(n) = O(n^{log_2 3}) = O(n^{1.58496})$

Exercise

Give an alternate algorithm for squaring an n-digit integer that makes recursive calls to a squaring algorithm (Don't use the above)

An integer with n digits can be represented as the following where $m = \lceil n/2 \rceil$

$$10^m a + b$$

Squaring this will give us

$$(10^m a + b)^2 = 10^{2m} a^2 + 10^m (2ab) + b^2$$

Using the identity

$$\begin{aligned}(a + b)^2 &= a^2 + 2ab + b^2 \\ (a + b)^2 - a^2 - b^2 &= 2ab\end{aligned}$$

We can obtain $2ab$ from the squares of $(a+b)$, a and b which again leads us to only 3 multiplications.

Number of digits in a, b → $\lceil n/2 \rceil \leq n/2 + 1$

Number of digits in $a+b$ → $\lceil n/2 \rceil + 1 \leq n/2 + 2$

Thus the max number of digits we have to recurse on is $n/2 + 2$

The recurrence relation will be as follows

$$T(n) \leq 3T(\frac{n}{2} + 2) + O(n)$$

Taking the domain substitution $S(n) = T(n+4)$

$$\begin{aligned} S(n) &\leq 3T(\frac{n+4}{2} + 2) + O(n) \\ &= 3T(\frac{n}{2} + 4) + O(n) \\ &= 3S(n/2) + O(n) \end{aligned}$$

Master's Theorem implies $S(n) = O(n^{log_2 3})$

$$T(n) = S(n-4) = C(n-4)^{log_2 3} = O(n^{log_2 3})$$

Exponentiation

Given a number a and a positive integer n, we want to compute a^n.

Suppose we multiply a by itself n times, this would require $n-1$ multiplications or $O(n)$ multiplications.

Note: $a^{16} = ((((a^2)^2)^2)^2$ which is **4** multiplications

$$a^n = \begin{cases} (a^{n/2})^2 & if\ n\ is\ even \\ (a^{(n-1)/2})^2 \cdot a & if\ n\ is\ odd \end{cases}$$

Here, we bound the number of multiplications by $T(n)$

When n is odd, we need to find $a^{n/2}$ which gives us $T(n/2)$ as well as multiply it by itself which will give us $+1$

When n is even, we need to find $a^{n/2}$ which gives us $T(n/2)$, multiply it by itself which will give us $+1$ and also multiply it by a which gives us $+1$.

Thus, $T(n) \leq T(n/2) + 2$

By Master's Theorem, we get $T(n) = O(log_2 n)$

Claim: $T(n) \leq 2log_2 n$ for $n \geq 1$ and some constant C

When $n = 1$, $T(1) = 0$ and $2log_2(1) = 0$ and the statement is true for $n = 1$

Assume $T(k) \leq 2log_2 k$ for $2 \leq k \leq n-1$,

$$\begin{aligned} T(n) &\leq T(n/2) + 2 \\ &\leq 2log_2(n/2) + 2 \\ &= 2log_2(n) - 2log_2(2) + 2 \\ &= 2log_2(n) \quad \checkmark \end{aligned}$$

Note This cost is strictly in terms of **number of multiplications**, the actual cost depends on the **cost of multiplication**.