The control of a large force is the same principle as the control of a few men:
it is merely a question of dividing up their numbers.
— Sun Zi, *The Art of War* (c. 400 C.E.), translated by Lionel Giles (1910)

2 Divide and Conquer

2.1 MergeSort

Mergesort is one of the earliest algorithms proposed for sorting. According to Knuth, it was suggested by John von Neumann as early as 1945.

1. Divide the array $A[1..n]$ into two subarrays $A[1..m]$ and $A[m+1..n]$, where $m = \lfloor n/2 \rfloor$.
2. Recursively mergesort the subarrays $A[1..m]$ and $A[m+1..n]$.
3. Merge the newly-sorted subarrays $A[1..m]$ and $A[m+1..n]$ into a single sorted list.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Divide:	S	O	R	T	I	N	G	E	X	A	M	P	L
Recurse:	I	N	O	S	R	T	A	E	G	L	M	P	X
Merge:	A	E	G	I	L	M	N	O	P	S	R	T	X

A Mergesort example.

The first step is completely trivial; we only need to compute the median index m. The second step is also trivial, thanks to our friend the recursion fairy. All the real work is done in the final step; the two sorted subarrays $A[1..m]$ and $A[m+1..n]$ can be merged using a simple linear-time algorithm. Here's a complete specification of the Mergesort algorithm; for simplicity, we separate out the merge step as a subroutine.

```
MERGESORT($A[1..n]$):
  if ($n > 1$)
      $m \leftarrow \lfloor n/2 \rfloor$
      MERGESORT($A[1..m]$)
      MERGESORT($A[m+1..n]$)
      MERGE($A[1..n], m$)
```

```
MERGE($A[1..n], m$):
  $i \leftarrow 1$; $j \leftarrow m+1$
  for $k \leftarrow 1$ to $n$
      if $j > n$
          $B[k] \leftarrow A[i]$; $i \leftarrow i+1$
      else if $i > m$
          $B[k] \leftarrow A[j]$; $j \leftarrow j+1$
      else if $A[i] < A[j]$
          $B[k] \leftarrow A[i]$; $i \leftarrow i+1$
      else
          $B[k] \leftarrow A[j]$; $j \leftarrow j+1$
  for $k \leftarrow 1$ to $n$
      $A[k] \leftarrow B[k]$
```

To prove that the algorithm is correct, we use our old friend induction. We can prove that MERGE is correct using induction on the total size of the two subarrays $A[i..m]$ and $A[j..n]$ left to be merged into $B[k..n]$. The base case, where at least one subarray is empty, is straightforward; the algorithm just copies it into B. Otherwise, the smallest remaining element is either $A[i]$ or $A[j]$, since both subarrays are sorted, so $B[k]$ is assigned correctly. The remaining subarrays—either $A[i+1..m]$ and $A[j..n]$, or $A[i..m]$ and $A[j+1..n]$—are merged correctly into $B[k+1..n]$ by the inductive hypothesis.[1] This completes the proof.

[1]"The inductive hypothesis" is just a technical nickname for our friend the recursion fairy.

Now we can prove MERGESORT correct by another round of straightforward induction.[2] The base cases $n \leq 1$ are trivial. Otherwise, by the inductive hypothesis, the two smaller subarrays $A[1\,..\,m]$ and $A[m+1\,..\,n]$ are sorted correctly, and by our earlier argument, merged into the correct sorted output.

What's the running time? Since we have a recursive algorithm, we're going to get a recurrence of some sort. MERGE clearly takes linear time, since it's a simple for-loop with constant work per iteration. We get the following recurrence for MERGESORT:

$$T(1) = O(1), \qquad T(n) = T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + O(n).$$

2.2 Aside: Domain Transformations

Except for the floor and ceiling, this recurrence falls into case (b) of the Master Theorem [CLR, §4.3]. If we simply ignore the floor and ceiling, the Master Theorem suggests the solution $T(n) = O(n \log n)$. We can easily check that this answer is correct using induction, but there is a simple method for solving recurrences like this directly, called *domain transformation*.

First we overestimate the time bound, once by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:

$$T(n) \leq 2T\big(\lceil n/2 \rceil\big) + O(n) \leq 2T(n/2+1) + O(n).$$

Now we define a new function $S(n) = T(n+\alpha)$, where α is a constant chosen so that $S(n)$ satisfies the Master-ready recurrence $S(n) \leq 2S(n/2) + O(n)$. To figure out the appropriate value for α, we compare two versions of the recurrence for $T(n+\alpha)$:

$$\begin{aligned} S(n) \leq 2S(n/2) + O(n) \quad &\Longrightarrow \quad T(n+\alpha) \leq 2T(n/2+\alpha) + O(n) \\ T(n) \leq 2T(n/2+1) + O(n) \quad &\Longrightarrow \quad T(n+\alpha) \leq 2T((n+\alpha)/2+1) + O(n+\alpha) \end{aligned}$$

For these two recurrences to be equal, we need $n/2+\alpha = (n+\alpha)/2+1$, which implies that $\alpha = 2$. The Master Theorem tells us that $S(n) = O(n \log n)$, so

$$T(n) = S(n-2) = O((n-2)\log(n-2)) = O(n \log n).$$

We can use domain transformations to remove floors, ceilings, and lower order terms from any recurrence. But now that we know this, we won't bother actually grinding through the details!

2.3 QuickSort

Quicksort was discovered by Tony Hoare in 1962. In this algorithm, the hard work is splitting the array into subsets so that merging the final result is trivial.

1. Choose a *pivot* element from the array.
2. Split the array into three subarrays containing the items less than the pivot, the pivot itself, and the items bigger than the pivot.
3. Recursively quicksort the first and last subarray.

[2]Many textbooks draw an artificial distinction between several different flavors of induction: standard/weak ('the principle of mathematical induction'), strong ('the second principle of mathematical induction'), complex, structural, transfinite, decaffeinated, etc. Those textbooks would call this proof "strong" induction. I don't. *All* induction proofs have precisely the same structure: Pick an arbitrary object, make one or more simpler objects from it, apply the inductive hypothesis to the simpler object(s), infer the required property for the original object, and check the base cases. Induction is just recursion for proofs.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	$\boxed{\text{N}}$	G	E	X	A	M	P	L
Partition:	M	A	E	G	I	L	N	R	X	O	S	P	T
Recurse:	A	E	G	I	L	M	N	O	P	S	R	T	X

A Quicksort example.

Here's a more formal specification of the Quicksort algorithm. The separate PARTITION subroutine takes the original position of the pivot element as input and returns the post-partition pivot position as output.

```
QUICKSORT($A[1\,..\,n]$):
  if ($n > 1$)
      *Choose a pivot element* $A[p]$
      $k \leftarrow$ PARTITION($A, p$)
      QUICKSORT($A[1\,..\,k-1]$)
      QUICKSORT($A[k+1\,..\,n]$)
```

```
PARTITION($A[1\,..\,n], p$):
  if ($p \neq n$)
      swap $A[p] \leftrightarrow A[n]$
  $i \leftarrow 0$; $j \leftarrow n$
  while ($i < j$)
      repeat $i \leftarrow i+1$ until ($i = j$ or $A[i] \geq A[n]$)
      repeat $j \leftarrow j-1$ until ($i = j$ or $A[j] \leq A[n]$)
      if ($i < j$)
          swap $A[i] \leftrightarrow A[j]$
  if ($i \neq n$)
      swap $A[i] \leftrightarrow A[n]$
  return $i$
```

Just as we did for mergesort, we need two induction proofs to show that QUICKSORT is correct—weak induction to prove that PARTITION correctly partitions the array, and then straightforward strong induction to prove that QUICKSORT correctly sorts assuming PARTITION is correct. I'll leave the gory details as an exercise for the reader.

The analysis is also similar to mergesort. PARTITION runs in $O(n)$ time: $j - i = n$ at the beginning, $j - i = 0$ at the end, and we do a constant amount of work each time we increment i or decrement j. For QUICKSORT, we get a recurrence that depends on k, the rank of the chosen pivot:

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

If we could choose the pivot to be the median element of the array A, we would have $k = \lceil n/2 \rceil$, the two subproblems would be as close to the same size as possible, the recurrence would become

$$T(n) = 2T\big(\lceil n/2 \rceil - 1\big) + T\big(\lfloor n/2 \rfloor\big) + O(n) \leq 2T(n/2) + O(n),$$

and we'd have $T(n) = O(n \log n)$ by the Master Theorem.

Unfortunately, although it is *theoretically* possible to locate the median of an unsorted array in linear time, the algorithm is fairly complicated, and the hidden constant in the $O()$ notation is quite large. So in practice, programmers settle for something simple, like choosing the first or last element of the array. In this case, k can be anything from 1 to n, so we have

$$T(n) = \max_{1 \leq k \leq n} \big(T(k-1) + T(n-k) + O(n)\big)$$

In the worst case, the two subproblems are completely unbalanced—either $k = 1$ or $k = n$—and the recurrence becomes $T(n) \leq T(n-1) + O(n)$. The solution is $T(n) = O(n^2)$. Another common heuristic is 'median of three'—choose three elements (usually at the beginning, middle, and end of the array), and take the middle one as the pivot. Although this is better in practice than just choosing one element, we can still have $k = 2$ or $k = n - 1$ in the worst case. With the median-of-three heuristic, the recurrence becomes $T(n) \le T(1) + T(n-2) + O(n)$, whose solution is still $T(n) = O(n^2)$.

Intuitively, the pivot element will 'usually' fall somewhere in the middle of the array, say between $n/10$ and $9n/10$. This suggests that the *average-case* running time is $O(n \log n)$. Although this intuition is correct, we are still far from a *proof* that quicksort is usually efficient. I'll formalize this intuition about average cases in a later lecture.

2.4 The Pattern

Both mergesort and and quicksort follow the same general three-step pattern of all divide and conquer algorithms:

1. **Split** the problem into several *smaller independent* subproblems.
2. **Recurse** to get a subsolution for each subproblem.
3. **Merge** the subsolutions together into the final solution.

If the size of any subproblem falls below some constant threshold, the recursion bottoms out. Hopefully, at that point, the problem is trivial, but if not, we switch to a different algorithm instead.

Proving a divide-and-conquer algorithm correct usually involves strong induction. Analyzing the running time requires setting up and solving a recurrence, which often (but unfortunately not always!) can be solved using the Master Theorem, perhaps after a simple domain transformation.

2.5 Multiplication

Adding two n-digit numbers takes $O(n)$ time by the standard iterative 'ripple-carry' algorithm, using a lookup table for each one-digit addition. Similarly, multiplying an n-digit number by a one-digit number takes $O(n)$ time, using essentially the same algorithm.

What about multiplying two n-digit numbers? At least in the United States, every grade school student (supposedly) learns to multiply by breaking the problem into n one-digit multiplications and n additions:

$$\begin{array}{r}
31415962 \\
\underline{\times\ 27182818} \\
251327696 \\
31415962 \\
251327696 \\
62831924 \\
251327696 \\
31415962 \\
219911734 \\
\underline{62831924} \\
853974377340916
\end{array}$$

We could easily formalize this algorithm as a pair of nested for-loops. The algorithm runs in $O(n^2)$ time—altogether, there are $O(n^2)$ digits in the partial products, and for each digit, we spend constant time.

We can do better by exploiting the following algebraic formula:

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

Here is a divide-and-conquer algorithm that computes the product of two n-digit numbers x and y, based on this formula. Each of the four sub-products e, f, g, h is computed recursively. The last line does not involve any multiplications, however; to multiply by a power of ten, we just shift the digits and fill in the right number of zeros.

```
MULTIPLY($x, y, n$):
  if $n = 1$
      return $x \cdot y$
  else
      $m \leftarrow \lceil n/2 \rceil$
      $a \leftarrow \lfloor x/10^m \rfloor$;  $b \leftarrow x \bmod 10^m$
      $d \leftarrow \lfloor y/10^m \rfloor$;  $c \leftarrow y \bmod 10^m$
      $e \leftarrow$ MULTIPLY($a, c, m$)
      $f \leftarrow$ MULTIPLY($b, d, m$)
      $g \leftarrow$ MULTIPLY($b, c, m$)
      $h \leftarrow$ MULTIPLY($a, d, m$)
      return $10^{2m}e + 10^m(g + h) + f$
```

You can easily prove by induction that this algorithm is correct. The running time for this algorithm is given by the recurrence

$$T(n) = 4T(\lceil n/2 \rceil) + O(n), \qquad T(1) = 1,$$

which solves to $T(n) = O(n^2)$ by the Master Theorem (after a simple domain transformation). Hmm... I guess this didn't help after all.

But there's a trick, first published by Anatoliĭ Karatsuba in 1962.[3] We can compute the middle coefficient $bc + ad$ using only *one* recursive multiplication, by exploiting yet another bit of algebra:

$$ac + bd - (a - b)(c - d) = bc + ad$$

This trick lets use replace the last three lines in the previous algorithm as follows:

```
FASTMULTIPLY($x, y, n$):
  if $n = 1$
      return $x \cdot y$
  else
      $m \leftarrow \lceil n/2 \rceil$
      $a \leftarrow \lfloor x/10^m \rfloor$;  $b \leftarrow x \bmod 10^m$
      $d \leftarrow \lfloor y/10^m \rfloor$;  $c \leftarrow y \bmod 10^m$
      $e \leftarrow$ FASTMULTIPLY($a, c, m$)
      $f \leftarrow$ FASTMULTIPLY($b, d, m$)
      $g \leftarrow$ FASTMULTIPLY($a - b, c - d, m$)
      return $10^{2m}e + 10^m(e + f - g) + f$
```

The running time of Karatsuba's FASTMULTIPLY algorithm is given by the recurrence

$$T(n) \leq 3T(\lceil n/2 \rceil) + O(n), \qquad T(1) = 1.$$

After a domain transformation, we can plug this into the Master Theorem to get the solution $T(n) = O(n^{\lg 3}) = O(n^{1.585})$, a significant improvement over our earlier quadratic-time algorithm.[4]

[3]However, the same basic trick was used non-recursively by Gauss in the 1800s to multiply complex numbers using only three real multiplications.

[4]Karatsuba actually proposed an algorithm based on the formula $(a + c)(b + d) - ac - bd = bc + ad$. This algorithm also runs in $O(n^{\lg 3})$ time, but the actual recurrence is a bit messier: $a - b$ and $c - d$ are still m-digit numbers, but $a + b$ and $c + d$ might have $m + 1$ digits. The simplification presented here is due to Donald Knuth.

Of course, in practice, all this is done in binary instead of decimal.

We can take this idea even further, splitting the numbers into more pieces and combining them in more complicated ways, to get even faster multiplication algorithms. Ultimately, this idea leads to the development of the *Fast Fourier transform*, a more complicated divide-and-conquer algorithm that can be used to multiply two n-digit numbers in $O(n \log n)$ time.[5] We'll talk about Fast Fourier transforms in detail in the next lecture.

2.6 Exponentiation

Given a number a and a positive integer n, suppose we want to compute a^n. The standard naïve method is a simple for-loop that does $n-1$ multiplications by a:

```
SLOWPOWER($a, n$):
  $x \leftarrow a$
  for $i \leftarrow 2$ to $n$
      $x \leftarrow x \cdot a$
  return $x$
```

This iterative algorithm requires n multiplications.

Notice that the input a could be an integer, or a rational, or a floating point number. In fact, it doesn't need to be a number at all, as long as it's something that we know how to multiply. For example, the same algorithm can be used to compute powers modulo some finite number (an operation commonly used in cryptography algorithms) or to compute powers of matrices (an operation used to evaluate recurrences and to compute shortest paths in graphs). All that's required is that a belong to a multiplicative group.[6] Since we don't know what kind of things we're mutliplying, we can't know how long a multiplication takes, so we're forced analyze the running time in terms of the number of multiplications.

There is a much faster divide-and-conquer method, using the simple formula $a^n = a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil}$. What makes this approach more efficient is that once we compute the first factor $a^{\lfloor n/2 \rfloor}$, we can compute the second factor $a^{\lceil n/2 \rceil}$ using at most one more multiplication.

```
FASTPOWER($a, n$):
  if $n = 1$
      return $a$
  else
      $x \leftarrow$ FASTPOWER($a, \lfloor n/2 \rfloor$)
      if $n$ is even
          return $x \cdot x$
      else
          return $x \cdot x \cdot a$
```

The total number of multiplications is given by the recurrence $T(n) \leq T(\lfloor n/2 \rfloor) + 2$, with the base case $T(1) = 0$. After a domain transformation, the Master Theorem gives us the solution $T(n) = O(\log n)$.

[5]This fast algorithm for multiplying integers using FFTs was discovered by Arnold Schönhange and Volker Strassen in 1971.

[6]A *multiplicative group* $(G, \otimes)$ is a set G and a function $\otimes : G \times G \to G$, satisfying three axioms:

1. There is a *unit* element $1 \in G$ such that $1 \otimes g = g \otimes 1$ for any element $g \in G$.
2. Any element $g \in G$ has a *inverse* element $g^{-1} \in G$ such that $g \otimes g^{-1} = g^{-1} \otimes g = 1$
3. The function is *associative*: for any elements $f, g, h \in G$, we have $f \otimes (g \otimes h) = (f \otimes g) \otimes h$.

Incidentally, this algorithm is asymptotically optimal—any algorithm for computing a^n must perform $\Omega(\log n)$ multiplications. In fact, when n is a power of two, this algorithm is *exactly* optimal. However, there are slightly faster methods for other values of n. For example, our divide-and-conquer algorithm computes a^{15} in six multiplications ($a^{15} = a^7 \cdot a^7 \cdot a$; $a^7 = a^3 \cdot a^3 \cdot a$; $a^3 = a \cdot a \cdot a$), but only five multiplications are necessary ($a \to a^2 \to a^3 \to a^5 \to a^{10} \to a^{15}$). Nobody knows of an algorithm that always uses the minimum possible number of multiplications.

2.7 Optimal Binary Search Trees

This last example combines the divide-and-conquer strategy with recursive backtracking.

You all remember that the cost of a successful search in a binary search tree is proportional to the number of ancestors of the target node.[7] As a result, the worst-case search time is proportional to the depth of the tree. To minimize the worst-case search time, the height of the tree should be as small as possible; ideally, the tree is perfectly balanced.

In many applications of binary search trees, it is more important to minimize the total cost of several searches than to minimize the worst-case cost of a single search. If x is a more 'popular' search target than y, we can save time by building a tree where the depth of x is smaller than the depth of y, even if that means increasing the overall depth of the tree. A perfectly balanced tree is *not* the best choice if some items are significantly more popular than others. In fact, a totally unbalanced tree of depth $\Omega(n)$ might actually be the best choice!

This situation suggests the following problem. Suppose we are given a sorted array of n keys $A[1\,..\,n]$ and an array of corresponding *access frequencies* $f[1\,..\,n]$. Over the lifetime of the search tree, we will search for the key $A[i]$ exactly $f[i]$ times. Our task is to build the binary search tree that minimizes the *total* search time.

Before we think about how to solve this problem, we should first come up with a good recursive definition of the function we are trying to optimize! Suppose we are also given a binary search tree T with n nodes; let v_i denote the node that stores $A[i]$. Up to constant factors, the total cost of performing all the binary searches is given by the following expression:

$$\begin{aligned} \mathit{Cost}(T, f[1\,..\,n]) &= \sum_{i=1}^{n} f[i] \cdot \#\text{ancestors of } v_i \\ &= \sum_{i=1}^{n} f[i] \cdot \sum_{j=1}^{n} \big[v_j \text{ is an ancestor of } v_i\big] \end{aligned}$$

Here I am using the extremely useful *Iverson bracket notation* to transform Boolean expressions into integers—for any Boolean expression X, we define $[X] = 1$ if X is true, and $[X] = 0$ if X is false.[8] Since addition is commutative, we can swap the order of the two summations.

$$\mathit{Cost}(T, f[1\,..\,n]) = \sum_{j=1}^{n} \sum_{i=1}^{n} f[i] \cdot \big[v_j \text{ is an ancestor of } v_i\big] \qquad (*)$$

Finally, we can exploit the recursive structure of the binary tree by splitting the outer sum into a sum over the left subtree of T, a sum over the right subtree of T, and a 'sum' over the root of T.

[7] An *ancestor* of a node v is either the node itself or an ancestor of the parent of v. A *proper* ancestor of v is either the parent of v or a proper ancestor of the parent of v.

[8] In other words, $[X]$ has precisely the same semantics as $!!X$ in C.

Suppose v_r is the root of T.

$$\begin{aligned}
\textit{Cost}(T, f[1\,..\,n]) = &\sum_{j=1}^{r-1}\sum_{i=1}^{n} f[i]\cdot\big[v_j \text{ is an ancestor of } v_i\big]\\
&+\sum_{i=1}^{n} f[i]\cdot\big[v_r \text{ is an ancestor of } v_i\big]\\
&+\sum_{j=1}^{r+1}\sum_{i=1}^{n} f[i]\cdot\big[v_j \text{ is an ancestor of } v_i\big]
\end{aligned}$$

We can simplify all three of these partial sums. Since any node in the left subtree is an ancestor only of nodes in the left subtree, we can change the upper limit of the first inner summation from n to $r-1$. Similarly, we can change the lower limit of the last inner summation from 1 to $r+1$. Finally, the root v_r is an ancestor of *every* node in T, so we can remove the bracket expression from the middle summation.

$$\begin{aligned}
\textit{Cost}(T, f[1\,..\,n]) = &\sum_{j=1}^{r-1}\sum_{i=1}^{r} f[i]\cdot\big[v_j \text{ is an ancestor of } v_i\big]\\
&+\sum_{i=1}^{n} f[i]\\
&+\sum_{j=1}^{r+1}\sum_{i=r+1}^{n} f[i]\cdot\big[v_j \text{ is an ancestor of } v_i\big]
\end{aligned}$$

Now the first and third summations look exactly like our earlier expression (*) for $\textit{Cost}(T, f[1\,..\,n])$. We finally have our recursive definition!

$$\boxed{\textit{Cost}(T, f[1\,..\,n]) = \textit{Cost}(\textit{left}(T), f[1\,..\,r-1]) \;+\; \sum_{i=1}^{n} f[i] \;+\; \textit{Cost}(\textit{right}(T), f[r+1\,..\,n])}$$

The base case for this recurrence is, as usual, $n=0$; the cost of the empty tree, over the empty set of frequency counts, is zero.

Now our task is to compute the tree T_{opt} that minimizes this cost function. Suppose we somehow magically knew that the root of T_{opt} is v_r. Then the recursive definition of $\textit{Cost}(T, f)$ immediately implies that the left subtree left(T_{opt}) must be the optimal search tree for the keys $A[1\,..\,r-1]$ and access frequencies $f[1\,..\,r-1]$. Similarly, the right subtree right(T_{opt}) must be the optimal search tree for the keys $A[r+1\,..\,n]$ and access frequencies $f[r+1\,..\,n]$. **Once we choose the correct key to store at the root, the Recursion Fairy will automatically construct the rest of the optimal tree for us.** More formally, let $\textit{OptCost}(f[1\,..\,n])$ denote the total cost of the optimal search tree for the given frequency counts. We immediately have the following recursive definition.

$$\boxed{\textit{OptCost}(f[1\,..\,n]) = \min_{1\le r\le n}\left\{\textit{OptCost}(f[1\,..\,r-1]) \;+\; \sum_{i=1}^{n} f[i] \;+\; \textit{OptCost}(f[r+1\,..\,n])\right\}}$$

Again, the base case is $\textit{OptCost}(f[1\,..\,0]) = 0$; the best way to organize zero keys, given an empty set of frequencies, is by storing them in the empty tree!

This recursive definition can be translated mechanically into a recursive algorithm, whose running time $T(n)$ satisfies the recurrence

$$T(n) = \Theta(n) + \sum_{k=1}^{n} \big(T(k-1) + T(n-k)\big).$$

The $\Theta(n)$ term comes from computing the total number of searches $\sum_{i=1}^{n} f[i]$. The recurrence looks harder than it really is. To transform it into a more familiar form, we regroup and collect identical terms, subtract the recurrence for $T(n-1)$ to get rid of the summation, and then regroup again.

$$\begin{aligned} T(n) &= \Theta(n) + 2\sum_{k=0}^{n-1} T(k) \\ T(n-1) &= \Theta(n-1) + 2\sum_{k=0}^{n-2} T(k) \\ T(n) - T(n-1) &= \Theta(1) + 2T(n-1) \\ T(n) &= 3T(n-1) + \Theta(1) \end{aligned}$$

The solution $\boxed{T(n) = \Theta(3^n)}$ now follows from the annihilator method.

It's worth emphasizing here that our recursive algorithm does *not* examine all possible binary search trees. The number of binary search trees with n nodes satisfies the recurrence

$$N(n) = \sum_{r=1}^{n-1} N(r-1) \cdot N(n-r), \quad N(0) = 1,$$

which has the closed-from solution $N(n) = \Theta(4^n/\sqrt{n})$. Our algorithm saves considerable time by searching *independently* for the optimal left and right subtrees. A full enumeration of binary search trees would consider all possible *pairings* of left and right subtrees; hence the product in the recurrence for $N(n)$.