algorithm design paradigms: divide and conquer

Outline:

General Idea

Review of Merge Sort

Why does it work?
  - Importance of balance
  - Importance of super-linear growth

Some interesting applications
  - Closest points
  - Integer Multiplication

Finding & Solving Recurrences
Divide & Conquer

Reduce problem to one or more sub-problems of the same type

Typically, each sub-problem is at most a constant fraction of the size of the original problem

Subproblems typically disjoint

Often gives significant, usually polynomial, speedup

Examples:

- Binary Search, Mergesort, Quicksort (roughly),
- Strassen’s Algorithm, integer multiplication, powering,
- FFT, …
Motivating Example: Mergesort
MS(A: array[1..n]) returns array[1..n] {
  If(n=1) return A;
  New U:array[1:n/2] = MS(A[1..n/2]);
  New L:array[1:n/2] = MS(A[n/2+1..n]);
  Return(Merge(U,L));
}

Merge(U,L: array[1..n]) {
  New C: array[1..2n];
  a=1; b=1;
  For i = 1 to 2n
      “C[i] = smaller of U[a], L[b] and correspondingly a++ or b++, while being careful about running past end of either”;
  Return C;
}
Why does it work? Suppose we’ve already invented DumbSort, taking time $n^2$

Try *Just One Level* of divide & conquer:

- DumbSort(first $n/2$ elements)
- DumbSort(last $n/2$ elements)

Merge results

Time: $2 \left(\frac{n}{2}\right)^2 + n = \frac{n^2}{2} + n \ll n^2$

*Almost twice as fast!*
Moral 1: “two halves are better than a whole”
Two problems of half size are better than one full-size problem, even given $O(n)$ overhead of recombining, since the base algorithm has super-linear complexity.

Moral 2: “If a little's good, then more's better”
Two levels of D&C would be almost 4 times faster, 3 levels almost 8, etc., even though overhead is growing. Best is usually full recursion down to some small constant size (balancing "work" vs "overhead"). In the limit: you’ve just rediscovered mergesort!
Moral 3: unbalanced division good, but less so:

$$(.1n)^2 + (.9n)^2 + n = .82n^2 + n$$

The 18% savings compounds significantly if you carry recursion to more levels, actually giving $O(n \log n)$, but with a bigger constant. So worth doing if you can’t get 50-50 split, but balanced is better if you can.

This is intuitively why Quicksort with random splitter is good – badly unbalanced splits are rare, and not instantly fatal.

Moral 4: but consistent, completely unbalanced division doesn’t help much:

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

Little improvement here.
Mergesort: (recursively) sort 2 half-lists, then merge results.

\[ T(n) = 2T(n/2) + cn, \quad n \geq 2 \]

\[ T(1) = 0 \]

Solution: \( \Theta(n \log n) \)

(details later)
A Divide & Conquer Example:
Closest Pair of Points
closest pair of points: non-geometric version

Given \( n \) points and \textit{arbitrary} distances between them, find the closest pair. (E.g., think of distance as airfare – definitely \textit{not} Euclidean distance!)

\[ \binom{n}{2} \]

\[ \text{… and all the rest of the \( \binom{n}{2} \) edges…} \]

\textit{Must} look at all \( n \) choose 2 pairwise distances, else any one you didn’t check might be the shortest.

Also true for Euclidean distance in 1-2 dimensions?
Given \( n \) points on the real line, find the closest pair

Closest pair is \textit{adjacent} in ordered list

Time \( O(n \log n) \) to sort, if needed

Plus \( O(n) \) to scan adjacent pairs

Key point: do \textit{not} need to calc distances between all pairs: exploit geometry + ordering
Closest pair. Given $n$ points in the plane, find a pair with smallest Euclidean distance between them.

**Fundamental geometric primitive.**
- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
- Special case of nearest neighbor, Euclidean MST, Voronoi.

Brute force. Check all pairs of points $p$ and $q$ with $\Theta(n^2)$ comparisons.

1-D version. $O(n \log n)$ easy if points are on a line.

**Assumption.** No two points have same $x$ coordinate.

↑ Just to simplify presentation
closest pair of points. 2d, Euclidean distance: 1st try

Divide. Sub-divide region into 4 quadrants.
Divide. Sub-divide region into 4 quadrants.

Obstacle. Impossible to ensure n/4 points in each piece, so the “balanced subdivision” goal may be elusive/problematic.
Algorithm.
Divide: draw vertical line $L$ with $\approx n/2$ points on each side.
closest pair of points

Algorithm.

Divide: draw vertical line $L$ with $\approx n/2$ points on each side.
Conquer: find closest pair on each side, recursively.
Algorithm.

Divide: draw vertical line \( L \) with \( \approx n/2 \) points on each side.

Conquer: find closest pair on each side, recursively.

Combine: find closest pair with one point in each side.

Return best of 3 solutions.

\( \Theta(n^2) ? \)
Find closest pair with one point in each side, *assuming* distance $< \delta$.
Find closest pair with one point in each side, *assuming* distance < $\delta$.

Observation: suffices to consider points within $\delta$ of line $L$. 

$\delta = \min(12, 21)$
Find closest pair with one point in each side, \textit{assuming} distance $< \delta$.

Observation: suffices to consider points within $\delta$ of line $L$.

Almost the one-D problem again: Sort points in $2\delta$-strip by their $y$ coordinate.
Find closest pair with one point in each side, \textit{assuming} distance < \( \delta \).

Observation: suffices to consider points within \( \delta \) of line \( L \).

Almost the one-D problem again: Sort points in \( 2\delta \)-strip by their \( y \) coordinate. Only check pts within 8 in sorted list!
Def. Let $s_i$ have the $i^{th}$ smallest $y$-coordinate among points in the $2\delta$-width-strip.

Claim. If $|i - j| > 8$, then the distance between $s_i$ and $s_j$ is $> \delta$.

Pf: No two points lie in the same $\delta/2$-by-$\delta/2$ square:

\[
\sqrt{\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2} = \frac{\sqrt{2}}{2} \delta \approx 0.7\delta < \delta
\]

so $\leq 8$ points within $+\delta$ of $y(s_i)$. 

closest pair of points
Closest-Pair(p₁, ..., pₙ) {
    if(n <= ??) return ??

    Compute separation line L such that half the points are on one side and half on the other side.

    δ₁ = Closest-Pair(left half)
    δ₂ = Closest-Pair(right half)
    δ = min(δ₁, δ₂)

    Delete all points further than δ from separation line L

    Sort remaining points p[1]...p[m] by y-coordinate.

    for i = 1..m
        k = 1
        while i+k <= m && p[i+k].y < p[i].y + δ
            δ = min(δ, distance between p[i] and p[i+k]);
            k++;

        return δ.
    }

Analysis, I: Let $D(n)$ be the number of pairwise distance calculations in the Closest-Pair Algorithm when run on $n \geq 1$ points

$$D(n) \leq \begin{cases} 0 & n = 1 \\ 2D(n/2) + 7n & n > 1 \end{cases} \quad \Rightarrow \quad D(n) = O(n \log n)$$

BUT – that’s only the number of distance calculations

What if we counted comparisons?
Analysis, II: Let $C(n)$ be the number of comparisons between coordinates/distances in the Closest-Pair Algorithm when run on $n \geq 1$ points

$$C(n) \leq \begin{cases} 
0 & n = 1 \\
2C(n/2) + kn \log n & n > 1 
\end{cases} \Rightarrow C(n) = O(n \log^2 n)$$

for some constant $k$

Q. Can we achieve $O(n \log n)$?

A. Yes. Don't sort points from scratch each time.
   Sort by $x$ at top level only.
   Each recursive call returns $\delta$ and list of all points sorted by $y$
   Sort by merging two pre-sorted lists.

$$T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$$
is it worth the effort?

Code is longer & more complex

$O(n \log n)$ vs $O(n^2)$ may hide $10x$ in constant?

How many points?

<table>
<thead>
<tr>
<th>$n$</th>
<th>Speedup: $\frac{n^2}{10n \log_2 n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.3</td>
</tr>
<tr>
<td>100</td>
<td>1.5</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>75</td>
</tr>
<tr>
<td>100,000</td>
<td>602</td>
</tr>
<tr>
<td>1,000,000</td>
<td>5,017</td>
</tr>
<tr>
<td>10,000,000</td>
<td>43,004</td>
</tr>
</tbody>
</table>
Going From Code to Recurrence
Carefully define what you’re counting, and write it down!

“Let $C(n)$ be the number of comparisons between sort keys used by MergeSort when sorting a list of length $n \geq 1$.”

In code, clearly separate base case from recursive case, highlight recursive calls, and operations being counted.

Write Recurrence(s)
merge sort

**Base Case**

**Recursive calls**

**One Recursive Level**

**Operations being counted**

**MS(A: array[1..n]) returns array[1..n] {**

*If*(n=1) *return* A;

New L: array[1:n/2] = MS(A[1..n/2]);

New R: array[1:n/2] = MS(A[n/2+1..n]);

Return(Merge(L,R));

**Merge(A,B: array[1..n]) {**

New C: array[1..2n];

a=1; b=1;

For i = 1 to 2n {

C[i] = “smaller of A[a], B[b] and a++ or b++”;

Return C;
}

Total time: proportional to $C(n)$
(loops, copying data, parameter passing, etc.)
Carefully define what you’re counting, and *write it down*!

“Let $D(n)$ be the number of pairwise distance calculations in the Closest-Pair Algorithm when run on $n \geq 1$ points”

In code, clearly separate *base case* from *recursive case*, highlight *recursive calls*, and *operations being counted*. Write Recurrence(s)
Algorithm:

```plaintext
Closest-Pair(p_1, ..., p_n) {
    if(n <= 1) return \infty

    Compute separation line L such that half the points are on one side and half on the other side.

    δ_1 = Closest-Pair(left half)
    δ_2 = Closest-Pair(right half)
    δ = min(δ_1, δ_2)

    Delete all points further than δ from separation line L

    Sort remaining points p[1]...p[m] by y-coordinate.
    for i = 1..m
        k = 1
        while i+k <= m && p[i+k].y < p[i].y + δ
            δ = min(δ, distance between p[i] and p[i+k]);
            k++;
    return δ.
}
```

Basic operations:
- distance calcs

Recursive calls (2)

Base Case

One recursive level

2D(n / 2)

7n
Analysis, I: Let $D(n)$ be the number of pairwise distance calculations in the Closest-Pair Algorithm when run on $n \geq 1$ points.

$$D(n) \leq \begin{cases} 0 & n = 1 \\ 2D(n/2) + 7n & n > 1 \end{cases} \Rightarrow D(n) = O(n \log n)$$

BUT – that’s only the number of distance calculations.

What if we counted comparisons?
Carefully define what you’re counting, and write it down!

“Let $D(n)$ be the number of comparisons between coordinates/distances in the Closest-Pair Algorithm when run on $n \geq 1$ points”

In code, clearly separate base case from recursive case, highlight recursive calls, and operations being counted.

Write Recurrence(s)
Closest Pair \( (p_1, \ldots, p_n) \) { 
  if \( n \leq 1 \) return \( \infty \)

  Compute separation line \( L \) such that half the points are on one side and half on the other side.

  \[ \delta_1 = \text{Closest Pair(left half)} \]
  \[ \delta_2 = \text{Closest Pair(right half)} \]
  \[ \delta = \min(\delta_1, \delta_2) \]

  Delete all points further than \( \delta \) from separation line \( L \)

  Sort remaining points \( p[1] \ldots p[m] \) by y-coordinate.

  for \( i = 1 \ldots m \)
    \( k = 1 \)
    while \( i+k \leq m \) && \( p[i+k].y < p[i].y + \delta \)
      \( \delta = \min(\delta, \text{distance between } p[i] \text{ and } p[i+k]) \)
      \( k++ \)

  return \( \delta \).
}
closest pair of points: analysis

Analysis, II: Let \( C(n) \) be the number of comparisons of coordinates/distances in the Closest-Pair Algorithm when run on \( n \geq 1 \) points

\[
C(n) \leq \begin{cases} 
0 & n = 1 \\
2C(n/2) + k_4 n \log n + 1 & n > 1 
\end{cases} \Rightarrow C(n) = O(n \log^2 n)
\]

for some \( k_4 \leq k_1 + k_2 + k_3 + 15 \)

Q. Can we achieve time \( O(n \log n) \)?

A. Yes. Don't sort points from scratch each time.

Sort by \( x \) at top level only.

Each recursive call returns \( \delta \) and list of all points sorted by \( y \)

Sort by merging two pre-sorted lists.

\[
T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)
\]
Integer Multiplication
Add. Given two n-bit integers $a$ and $b$, compute $a + b$. 

$O(n)$ bit operations.
Add. Given two n-bit integers $a$ and $b$, compute $a + b$. \[ O(n) \text{ bit operations.} \]

Multiply. Given two n-bit integers $a$ and $b$, compute $a \times b$. The “grade school” method:

\[ \Theta(n^2) \text{ bit operations.} \]
To multiply two 2-digit integers:

Multiply four 1-digit integers.

Add, shift some 2-digit integers to obtain result.

\[
x = 10 \cdot x_1 + x_0 \\
y = 10 \cdot y_1 + y_0 \\
xy = (10 \cdot x_1 + x_0)(10 \cdot y_1 + y_0) \\
= 100 \cdot x_1y_1 + 10 \cdot (x_1y_0 + x_0y_1) + x_0y_0
\]

Same idea works for long integers – can split them into 4 half-sized ints ("10" becomes "10^k", k = length/2)
**divide & conquer multiplication: warmup**

To multiply two $n$-bit integers:
- Multiply four $\frac{1}{2}n$-bit integers.
- Shift/add four $n$-bit integers to obtain result.

\[
x = 2^{n/2} \cdot x_1 + x_0
\]
\[
y = 2^{n/2} \cdot y_1 + y_0
\]
\[
xy = \left(2^{n/2} \cdot x_1 + x_0\right)\left(2^{n/2} \cdot y_1 + y_0\right)
\]
\[
= 2^n \cdot x_1y_1 + 2^{n/2} \cdot (x_1y_0 + x_0y_1) + x_0y_0
\]

\[
T(n) = 4T(n/2) + \Theta(n) \implies T(n) = \Theta(n^2)
\]

\[\uparrow\]

assumes $n$ is a power of 2
key trick: 2 multiplies for the price of 1:

\[
x = 2^{n/2} \cdot x_1 + x_0
\]
\[
y = 2^{n/2} \cdot y_1 + y_0
\]
\[
xy = (2^{n/2} \cdot x_1 + x_0)(2^{n/2} \cdot y_1 + y_0)
\]
\[
= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0
\]

\[
\alpha = x_1 + x_0
\]
\[
\beta = y_1 + y_0
\]
\[
\alpha \beta = (x_1 + x_0)(y_1 + y_0)
\]
\[
= x_1 y_1 + (x_1 y_0 + x_0 y_1) + x_0 y_0
\]
\[
(x_1 y_0 + x_0 y_1) = \alpha \beta - x_1 y_1 - x_0 y_0
\]

Well, ok, 4 for 3 is more accurate…
Karatsuba multiplication

To multiply two n-bit integers:

Add two pairs of \( \frac{1}{2}n \) bit integers.

Multiply three pairs of \( \frac{1}{2}n \)-bit integers.

Add, subtract, and shift n-bit integers to obtain result.

\[
\begin{align*}
x &= 2^{n/2} \cdot x_1 + x_0 \\
y &= 2^{n/2} \cdot y_1 + y_0 \\
xy &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0 \\
&= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot ((x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0) + x_0 y_0
\end{align*}
\]

Theorem. [Karatsuba-Ofman, 1962] Can multiply two n-digit integers in \( O(n^{1.585}) \) bit operations.

\[
T(n) \leq T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + T\left(1 + \left\lfloor \frac{n}{2} \right\rfloor \right) + \Theta(n)
\]

Sloppy version: \( T(n) \leq 3T(n/2) + O(n) \)

\( \Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585}) \)
Karatsuba multiplication

Theorem. [Karatsuba-Ofman, 1962] Can multiply two n-digit integers in $O(n^{1.585})$ bit operations.

$$T(n) \leq T\left(\left\lfloor n/2 \right\rfloor \right) + T\left(\left\lfloor n/2 \right\rfloor \right) + T\left(1 + \left\lfloor n/2 \right\rfloor \right) + \Theta(n)$$

Best to solve it directly (but messy). Instead, it nearly always suffices to solve a simpler recurrence:

Sloppy version: $T(n) \leq 3T(n/2) + O(n)$

Intuition: If $T(n) = n^k$, then $T(n+1) = n^k + kn^{k-1} + \ldots = O(n^k)$

$\Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

(Proof later.)
Naïve: $\Theta(n^2)$

Karatsuba: $\Theta(n^{1.59...})$

Amusing exercise: generalize Karatsuba to do 5 size $n/3$ subproblems $\rightarrow \Theta(n^{1.46...})$

Best known: $\Theta(n \log n \log \log n)$

"Fast Fourier Transform"

but mostly unused in practice (unless you need really big numbers - a billion digits of $\pi$, say)

High precision arithmetic IS important for crypto
Recurrences

Above: Where they come from, how to find them

Next: how to solve them
Mergesort: (recursively) sort 2 half-lists, then merge results.

\[ T(n) = 2T(n/2) + cn, \quad n \geq 2 \]
\[ T(1) = 0 \]

Solution: \( \Theta(n \log n) \) (details later)

\textcolor{red}{\textbf{now!}}
Solve:

\[ T(1) = c \]

\[ T(n) = 2 \cdot T(n/2) + cn \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Num</th>
<th>Size</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 = 2^0</td>
<td>n</td>
<td>cn</td>
</tr>
<tr>
<td>1</td>
<td>2 = 2^1</td>
<td>n/2</td>
<td>2cn/2</td>
</tr>
<tr>
<td>2</td>
<td>4 = 2^2</td>
<td>n/4</td>
<td>4cn/4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>i</td>
<td>2^i</td>
<td>n/2^i</td>
<td>2^i \cdot cn/2^i</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>k-1</td>
<td>2^{k-1}</td>
<td>n/2^{k-1}</td>
<td>2^{k-1} \cdot cn/2^{k-1}</td>
</tr>
<tr>
<td>k</td>
<td>2^k</td>
<td>n/2^k = 1</td>
<td>2^k \cdot T(1)</td>
</tr>
</tbody>
</table>

\[ n = 2^k ; k = \log_2 n \]

Total Work: \[ c \cdot n (1 + \log_2 n) \]

(add last col)
Solve:

\[ T(1) = c \]
\[ T(n) = 4 \cdot T(n/2) + cn \]

\[ n = 2^k \; ; \; k = \log_2 n \]

Total Work: \[ T(n) = \sum_{i=0}^{k} 4^i \cdot cn / 2^i = O(n^2) \]

\[ 4^k = (2^2)^k = (2^k)^2 = n^2 \]
Solve: 
\[ T(1) = c \]
\[ T(n) = 3 \, T(n/2) + cn \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Num</th>
<th>Size</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 = 3^0</td>
<td>n</td>
<td>cn</td>
</tr>
<tr>
<td>1</td>
<td>3 = 3^1</td>
<td>n/2</td>
<td>3cn/2</td>
</tr>
<tr>
<td>2</td>
<td>9 = 3^2</td>
<td>n/4</td>
<td>9cn/4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>i</td>
<td>3^i</td>
<td>n/2^i</td>
<td>3^i \cdot cn/2^i</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>k-1</td>
<td>3^{k-1}</td>
<td>n/2^{k-1}</td>
<td>3^{k-1} \cdot cn/2^{k-1}</td>
</tr>
<tr>
<td>k</td>
<td>3^k</td>
<td>n/2^k = 1</td>
<td>3^k \cdot T(1)</td>
</tr>
</tbody>
</table>

Total Work: 
\[ T(n) = \sum_{i=0}^{k} 3^i \frac{cn}{2^i} \]
Theorem: for $x \neq 1$,

$$1 + x + x^2 + x^3 + \ldots + x^k = \frac{x^{k+1}-1}{x-1}$$

proof:

$$y = 1 + x + x^2 + x^3 + \ldots + x^k$$

$$xy = x + x^2 + x^3 + \ldots + x^k + x^{k+1}$$

$$xy-y = x^{k+1} - 1$$

$$y(x-1) = x^{k+1} - 1$$

$$y = \frac{x^{k+1}-1}{x-1}$$
Solve:

\[ T(1) = c \]
\[ T(n) = 3 \ T(n/2) + cn \]  (cont.)

\[
T(n) = \sum_{i=0}^{k} 3^i \frac{cn}{2^i}
\]

\[
= cn \sum_{i=0}^{k} 3^i / 2^i
\]

\[
= cn \sum_{i=0}^{k} \left(\frac{3}{2}\right)^i
\]

\[
= cn \left(\frac{3}{2}\right)^{k+1} - 1
\]

\[
= \frac{cn}{\left(\frac{3}{2}\right)^{k+1} - 1}
\]
Solve:

\[ T(1) = c \]
\[ T(n) = 3 \ T(\frac{n}{2}) + cn \quad (\text{cont.}) \]

\[
cn \frac{\left(\frac{3}{2}\right)^{k+1} - 1}{\left(\frac{3}{2}\right) - 1} = 2cn \left(\left(\frac{3}{2}\right)^{k+1} - 1\right)
\]

\[
< 2cn \left(\frac{3}{2}\right)^{k+1}
\]

\[
= 3cn \left(\frac{3}{2}\right)^{k}
\]

\[
= 3cn \frac{3^k}{2^k}
\]
Solve:

\[ T(1) = c \]

\[ T(n) = 3 \ T(n/2) + cn \quad \text{(cont.)} \]

\[
3cn \frac{3^k}{2^k} = 3cn \frac{3^{\log_2 n}}{2^{\log_2 n}}
\]

\[
= 3cn \frac{3^{\log_2 n}}{n}
\]

\[
= 3c3^{\log_2 n}
\]

\[
= 3c \left( n^{\log_2 3} \right)
\]

\[
= O\left( n^{1.585\ldots} \right)
\]

\[
a^{\log_b n}
\]

\[
= \left( b^{\log_b a} \right)^{\log_b n}
\]

\[
= \left( b^{\log_b n} \right)^{\log_b a}
\]

\[
= n^{\log_b a}
\]
divide and conquer – master recurrence

\[ T(n) = aT(n/b) + cn^k \] for \( n > b \) then

\[ a > b^k \implies T(n) = \Theta(n^{\log_b a}) \] [many subprobs → leaves dominate]

\[ a < b^k \implies T(n) = \Theta(n^k) \] [few subprobs → top level dominates]

\[ a = b^k \implies T(n) = \Theta(n^k \log n) \] [balanced → all \( \log n \) levels contribute]

Fine print:
\[ T(1) = d; \ a \geq 1; \ b > 1; \ c, \ d, \ k \geq 0; \ n = b^t \] for some \( t > 0; \)
\[ a, \ b, \ k, \ t \] integers. True even if it is \([n/b]\) instead of \(n/b\).
master recurrence: proof sketch

Expand recurrence as in earlier examples, to get

\[ T(n) = n^h \ (d + c \ S) \]

where \( h = \log_b(a) \) (and \( n^h \) = number of tree leaves) and \( S = \sum_{j=1}^{\log_b n} x^j \),

where \( x = b^k/a \).

If \( c = 0 \) the sum \( S \) is irrelevant, and \( T(n) = O(n^h) \): all work happens in the base cases, of which there are \( n^h \), one for each leaf in the recursion tree.

If \( c > 0 \), then the sum matters, and splits into 3 cases (like previous slide):

- if \( x < 1 \), then \( S < x/(1-x) = O(1) \). [\( S \) is the first \( \log n \) terms of the infinite series with that sum.]

- if \( x = 1 \), then \( S = \log_b(n) = O(\log n) \). [All terms in the sum are 1 and there are that many terms.]

- if \( x > 1 \), then \( S = x \cdot (x^{1+\log_b(n)}-1)/(x-1) \). [And after some algebra, \( n^h \cdot S = O(n^k) \).]
Another Example: Exponentiation
another d&c example: fast exponentiation

Power(a,n)

Input: integer n and number a
Output: $a^n$

Obvious algorithm

$n-1$ multiplications

Observation:

if $n$ is even, $n = 2m$, then $a^n = a^m \cdot a^m$
divide & conquer algorithm

Power(a,n)
   if n = 0 then return(1)
   if n = 1 then return(a)
   x ← Power(a,[n/2])
   x ← x•x
   if n is odd then
      x ← a•x
   return(x)
Let $M(n)$ be number of multiplies

Worst-case recurrence:

$$M(n) = \begin{cases} 
0 & n \leq 1 \\
M(\lfloor n/2 \rfloor) + 2 & n > 1
\end{cases}$$

By master theorem

$$M(n) = O(\log n) \quad (a=1, b=2, k=0)$$

More precise analysis:

$$M(n) = \lfloor \log_2 n \rfloor + (\# \text{ of 1's in } n\text{'s binary representation}) - 1$$

Time is $O(M(n))$ if numbers $< \text{word size}$, else also depends on length, multiply algorithm
Instead of $a^n$ want $a^n \mod N$

$$a^{i+j} \mod N = ((a^i \mod N) \cdot (a^j \mod N)) \mod N$$

same algorithm applies with each $x \cdot y$ replaced by

$$((x \mod N) \cdot (y \mod N)) \mod N$$

In RSA cryptosystem (widely used for security)

need $a^n \mod N$ where $a, n, N$ each typically have 1024 bits

Power: at most 2048 multiplies of 1024 bit numbers

relatively easy for modern machines

Naive algorithm: $2^{1024}$ multiplies
Idea:

“Two halves are better than a whole”
if the base algorithm has super-linear complexity.

“If a little's good, then more's better”
repeat above, recursively

Analysis: recursion tree or Master Recurrence

Applications: Many.

Binary Search, Merge Sort, (Quicksort), Closest Points, Integer Multiply, Exponentiation,…