CSE 417

Algorithms:
Divide and Conquer

Larry Ruzzo

Thanks to Richard Anderson, Paul Beame, Kevin Wayne for some slides
Outline:

- General Idea
- Review of Merge Sort
- Why does it work?
  - Importance of balance
  - Importance of super-linear growth
- Some interesting applications
  - Inversions
  - Closest points
  - Integer Multiplication
- Finding & Solving Recurrences
Divide & Conquer

Reduce a problem to one or more (smaller) 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;
}

Time: Θ(n log n)
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) \( \mathcal{O}((n/2)^2) \)
- DumbSort(last $n/2$ elements) \( \mathcal{O}((n/2)^2) \)
- Merge results \( \mathcal{O}(n) \)

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

*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 merge sort!
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 *arbitrary* distances between them, find the closest pair. (E.g., think of distance as airfare – definitely *not* Euclidean distance!)

*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 *adjacent* in ordered list
Time \( O(n \log n) \) to sort, if needed
Plus \( O(n) \) to scan adjacent pairs
Key point: do *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.

Can we do as well in 2-D?

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.
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.
Find closest pair with one point in each side, *assuming* distance $< \delta$. 

$$\delta = \min(12, 21)$$
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, *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 $j - i \geq 8$, 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 $\leq 7$ points within $+\delta$ of $y(s_i)$. 
Closest-Pair($p_1, \ldots, p_n$) {
    if($n \leq ??$) return ??

    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 $i = 1..m$
        $k = 1$
        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 $\delta$.
}
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?
closest pair of points: analysis

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: $\frac{n^2}{10^n \log_2 n}$</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

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;
}
The recurrence

\[ C(n) = \begin{cases} 
0 & \text{if } n = 1 \\
2C(n/2) + (n - 1) & \text{if } n > 1 
\end{cases} \]

Base case

Recursive calls

Total time: proportional to \( C(n) \)

(loops, copying data, parameter passing, etc.)
going from code to recurrence

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)
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 $i = 1..m$
        $k = 1$
        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 $\delta$.
}
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?
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 \( \delta \).
}
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.

$$C(n) \leq \begin{cases} 0 & n = 1 \\ 2C(n/2) + k_4 n \log_2 n & n > 1 \end{cases} \Rightarrow C(n) = O(n \log^2 n)$$

for $k_4 = k_1 + k_2 + k_3 + 16$

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 8 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 \( a + b \).

\( \mathcal{O}(n) \) bit operations.

Multiply. Given two \( n \)-bit integers \( a \) and \( b \), compute \( a \times b \).

The “grade school” method:

\( \Theta(n^2) \) 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)
To multiply two \( n \)-bit integers:

Multiply four \( \frac{1}{2}n \)-bit integers.
Shift/add four \( n \)-bit integers to obtain result.

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

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

\[
\begin{array}{cccc}
  1 & 1 & 0 & 1 \\
  0 & 1 & 1 & 1 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 0 & 1
\end{array}
\]

\[
\begin{array}{cccc}
  y_1 & y_0 \\
  x_1 & x_0 \\
  x_0 \cdot y_0 \\
  x_1 \cdot y_1
\end{array}
\]

\[
\begin{array}{cccc}
  1 & 0 & 1 & 0 \\
  1 & 0 & 0 & 1 \\
  0 & 0 & 1 & 0 \\
  0 & 0 & 1 & 1
\end{array}
\]

\[
\begin{array}{cccc}
  x_1 \cdot y_1 \\
  x_1 \cdot y_0 \\
  x_0 \cdot y_0 \\
  x_1 \cdot y_1
\end{array}
\]

\[
\begin{array}{cccc}
  0 & 1 & 0 & 1 \\
  1 & 0 & 1 & 1 \\
  0 & 1 & 1 & 0 \\
  1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 0 & 1
\end{array}
\]

\[
\uparrow
\]

assumes \( n \) is a power of 2
key trick: 2 multiplies for the price of 1:

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

Well, ok, 4 for 3 is more accurate...

\[
\begin{align*}
\alpha &= x_1 + x_0 \\
\beta &= y_1 + y_0 \\
\alpha \beta &= (x_1 + x_0) (y_1 + y_0) \\
&= x_1 y_1 + (x_1 y_0 + x_0 y_1) + x_0 y_0 \\
\left(x_1 y_0 + x_0 y_1\right) &= \alpha \beta - x_1 y_1 - x_0 y_0
\end{align*}
\]
To multiply two $n$-bit integers:

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

Multiply \textit{three} pairs of $\frac{1}{2}n$-bit integers.

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

\[
\begin{align*}
x &= 2^{n/2} \cdot x_1 + x_0 \\
y &= 2^{n/2} \cdot y_1 + y_0 \\
xy &= 2^n \cdot x_1y_1 + 2^{n/2} \cdot (x_1y_0 + x_0y_1) + x_0y_0 \\
&= 2^n \cdot x_1y_1 + 2^{n/2} \cdot (x_1 + x_0)(y_1 + y_0) - x_1y_1 - x_0y_0 + x_0y_0
\end{align*}
\]

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

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

\textit{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.

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

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.)
multiplication – the bottom line

Naïve: \( \Theta(n^2) \)

Karatsuba: \( \Theta(n^{1.59\ldots}) \)

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

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, among other uses
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)

now!
Solve: \[ T(1) = c \]
\[ T(n) = 2 \, T(n/2) + cn \]

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

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

(add last col)
Solve:

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

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

Total Work:

\[ T(n) = \sum_{i=0}^{k} 4^i \frac{cn}{2^i} = O(n^2) \]

\[ 4^k = (2^2)^k = (2^k)^2 = n^2 \]
Solve:

\[ T(1) = c \]

\[ T(n) = 3 \cdot T(n/2) + cn \]

<table>
<thead>
<tr>
<th>Level</th>
<th>Num</th>
<th>Size</th>
<th>Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 = 3^0</td>
<td>n</td>
<td>( cn )</td>
</tr>
<tr>
<td>1</td>
<td>3 = 3^1</td>
<td>n/2</td>
<td>3( cn/2 )</td>
</tr>
<tr>
<td>2</td>
<td>9 = 3^2</td>
<td>n/4</td>
<td>9( cn/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 c \cdot n/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 c \cdot n/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>

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

Total Work: \( T(n) = \sum_{i=0}^{k} 3^i cn / 2^i \)
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: \[ T(1) = c \]
[736x53]
\[ T(n) = 3 \, T(n/2) + cn \quad (\text{cont.}) \]

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

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

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

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

\[
\sum_{i=0}^{k} x^i = \frac{x^{k+1} - 1}{x - 1} \quad (x \neq 1)
\]
Solve:  
\[ T(1) = c \]
\[ T(n) = 3 \ T(n/2) + cn \quad (\text{cont.}) \]

\[
\begin{align*}
\frac{cn}{(\frac{3}{2})^{k+1}} - 1 &= 2cn\left((\frac{3}{2})^{k+1} - 1\right) \\
< 2cn\left(\frac{3}{2}\right)^{k+1} \\
= 3cn\left(\frac{3}{2}\right)^{k} \\
= 3cn \frac{3^k}{2^k}
\end{align*}
\]
Solve:

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

\[
3cn \frac{3^k}{2^k} = 3cn \frac{3^{\log_2 n}}{2^{\log_2 n}}
\]
\[ = 3cn \frac{3^{\log_2 n}}{n} \]
\[ = 3c3^{\log_2 n} \]
\[ = 3c \left( n^{\log_2 3} \right) \]
\[ = O \left( n^{1.585...} \right) \]
divide and conquer – master recurrence

\[ T(n) = d \quad \text{for } n < b, \]
\[ T(n) = aT(n/b) + cn^k \quad \text{for } n \geq b \text{ then} \]

\[ a > b^k \quad \Rightarrow \quad T(n) = \Theta(n^{\log_b a}) \quad \text{[many subprobs → leaves dominate]} \]
\[ a < b^k \quad \Rightarrow \quad T(n) = \Theta(n^k) \quad \text{[few subprobs → top level dominates]} \]
\[ a = b^k \quad \Rightarrow \quad T(n) = \Theta(n^k \log n) \quad \text{[balanced → all log } n \text{ levels contribute]} \]

Fine print:
\[ a \geq 1; \quad b > 1; \quad c, \quad d, \quad k \geq 0; \quad n = b^t \text{ for some } t > 0; \]
\[ a, \quad b, \quad k, \quad t \text{ integers. True even if it is } \left\lfloor n/b \right\rfloor \text{ instead of } n/b \text{ when } \]
\[ t \text{ is not an integer.} \]
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 \( S < x/(1-x) = O(1) \). \[ S \text{ is the first log n terms of the infinite series with that sum.} \]

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

if \( x > 1 \), then \( S = x \cdot (x^{1+\log_b(n)}-1)/(x-1) \). \[ \text{And after some algebra, } n^h \cdot S = O(n^k). \]
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:

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

Time is \( O(M(n)) \) if numbers < word size, else also depends on length, multiply algorithm
a practical application - RSA

Instead of $a^n$ want $a^n \text{ mod } N$

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

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

$$((x \text{ mod } N) \cdot (y \text{ mod } N)) \text{ mod } N$$

In RSA cryptosystem (widely used for security)

need $a^n \text{ mod } N$ 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
Utility:
Correctness often easy; often faster

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
among others

Applications: Many.
Binary Search, Merge Sort, (Quicksort), Closest Points, Integer Multiply, Exponentiation,…