Divide and Conquer
Richard Anderson
With Special Cameo Appearance by Larry Ruzzo

Divide and Conquer Algorithms

Split into sub problems
Recursively solve the problem
Combine solutions

Make progress in the split and combine stages
- Quicksort – progress made at the split step
- Mergesort – progress made at the combine step

D&C Algorithms
- Strassen’s Algorithm – Matrix Multiplication
- Inversions
- Median
- Closest Pair
- Integer Multiplication
- FFT
- ...

divide & conquer – the key idea

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!

d&c approach, cont.

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!

mergesort (review)

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

$T(n) = 2T(n/2) + cn, \ n \geq 2$

$T(1) = 0$

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

What you really need to know about recurrences

Work per level changes geometrically with the level
- Geometrically increasing ($x > 1$)
  - The bottom level wins – count leaves
- Geometrically decreasing ($x < 1$)
  - The top level wins – count top level work
- Balanced ($x = 1$)
  - Equal contribution – top • levels (e.g. “n log n”)
Recurrences

Next: how to solve them

Solve: \( T(1) = c \)
\( T(n) = 2 T(n/2) + cn \)

\[
\begin{array}{|c|c|c|}
\hline
\text{Level} & \text{Num} & \text{Size} \\
\hline
0 & 1 = 2^0 & n \\
1 & 2 = 2^1 & n/2 \\
2 & 4 = 2^2 & n/4 \\
\vdots & \vdots & \vdots \\
\text{k-1} & 2^{k-1} & n/2^{k-1} \\
\text{k} & 2^k & n/2^{k} = 1 \\
\hline
\end{array}
\]

Total Work: \( c n (1 + \log_2 n) \)

Solve: \( T(1) = c \)
\( T(n) = 3 T(n/2) + cn \)

\[
\begin{array}{|c|c|c|}
\hline
\text{Level} & \text{Num} & \text{Size} \\
\hline
0 & 1 = 3^0 & n \\
1 & 3 = 3^1 & n/2 \\
2 & 9 = 3^2 & n/4 \\
\vdots & \vdots & \vdots \\
\text{k-1} & 3^{k-1} & n/2^{k-1} \\
\text{k} & 3^k & n/2^{k} = 1 \\
\hline
\end{array}
\]

Total Work: \( \sum_{i=0}^{k} 3^i cn / 2^i \)

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

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

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

Solve: \( T(1) = c \)
\( T(n) = 4 T(n/2) + cn \)

\[
\begin{array}{|c|c|c|}
\hline
\text{Level} & \text{Num} & \text{Size} \\
\hline
0 & 1 = 4^0 & n \\
1 & 4 = 4^1 & n/2 \\
2 & 16 = 4^2 & n/4 \\
\vdots & \vdots & \vdots \\
\text{k-1} & 4^{k-1} & n/2^{k-1} \\
\text{k} & 4^k & n/2^{k} = 1 \\
\hline
\end{array}
\]

Total Work: \( 4^k (2^{2^k} - 1) / (2^2 - 1) \)
Theorem:

\[ 1 + x + x^2 + x^3 + \ldots + x^k = \frac{(x^{k+1}-1)}{(x-1)} \]

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:

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

For \( n > b \) then:

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

Fine print:

\[ a \geq 1; b > 1; c, d, k \geq 0; T(1) = d; n = b^t \text{ for some } t > 0; \]
\[ a, b, k, t \text{ integers. True even if it is } [n/b] \text{ instead of } n/b \]

Solve:

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

\[ T(n) = \sum_{i=0}^{k} \frac{3^i cn / 2^i}{2^i} \]
\[ = 3cn \sum_{i=0}^{k} \frac{3^i}{2^i} \]
\[ = 3cn \left( \frac{\frac{3}{2}^{k+1} - 1}{\frac{3}{2} - 1} \right) \]
\[ = 3cn \cdot \frac{3^k}{2^k} \]

\[ \sum_{i=0}^{k} x^i = \frac{x^{k+1} - 1}{x - 1} \quad \text{[for } x = 1] \]

master recurrence: proof sketch

Expanding recurrence as in earlier examples, to get

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

where \( h = \log_b a \) (tree height) and \( S = \sum_{i=0}^{h-1} x^i \) where \( x = \frac{b}{a} \).

If \( c = 0 \) the sum \( S \) is irrelevant, and \( T(n) = O(n^h) \); all the 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 \( S = x^h / (1-x) = O(1) \). It is just the first \( h \) terms of the infinite series with that sum.

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

if \( x > 1 \), then \( S = x - (x^{h+1} - 1) / (x-1) \). After some algebra,

\[ n^h + O(n) \]
Example:

Matrix Multiplication –

Strassen’s Method

```
for i = 1 to n
  for j = 1 to n
    C[i,j] = 0
    for k = 1 to n
```

n^3 multiplications, n^3 - n^2 additions
Multiplying Matrices

Counting arithmetic operations:
\[ T(n) = 8T(n/2) + 4(n/2)^2 = 8T(n/2) + n^2 \]

Multiplying Matrices

Strassen’s algorithm
Multiply 2x2 matrices using 7 instead of 8 multiplications (and lots more than 4 additions)
\[ T(n) = \Theta(n^\log_27) = \Theta(n^{2.81}) \]

Asymptotically fastest known algorithm uses \( O(n^{2.376}) \) time not practical but Strassen’s may be practical provided calculations are exact and we stop recursion when matrix has size about 100 (maybe 10)

The algorithm
\[
\begin{align*}
P_1 &= A_{12}(B_{11} + B_{21}) \\
P_2 &= A_{11}B_{12} + A_{12}B_{22} \\
P_3 &= A_{21}B_{11} + A_{22}B_{21} + A_{11}B_{12} + A_{12}B_{22} \\
P_4 &= (A_{21} - A_{12})(B_{12} + B_{22}) \\
P_5 &= (A_{22} - A_{12})(B_{21} - B_{22}) \\
P_6 &= (A_{11} - A_{21})(B_{12} - B_{11}) \\
P_7 &= (A_{11} - A_{12})(B_{11} + B_{22}) \\
C_{11} &= P_1 + P_3 \\
C_{12} &= P_2 + P_3 + P_6 - P_7 \\
C_{21} &= P_1 + P_4 + P_5 + P_7 \\
C_{22} &= P_2 + P_4
\end{align*}
\]

Inversion Problem
Let \( a_1, \ldots, a_n \) be a permutation of 1 . . . n
(a, a) is an inversion if i < j and \( a_i > a_j \)

4, 6, 1, 7, 3, 2, 5

Problem: given a permutation, count the number of inversions
This can be done easily in \( O(n^2) \) time
Can we do better?
Application

Counting inversions can be used to measure closeness of ranked preferences. People rank 20 movies, based on their rankings you cluster people who like the same types of movies. Can also be used to measure nonlinear correlation.

Inversion Problem

Let $a_1, \ldots, a_n$ be a permutation of $1 \ldots n$. $(a_i, a_j)$ is an inversion if $i < j$ and $a_i > a_j$.

For example, consider the permutation $4, 6, 1, 7, 3, 2, 5$.

Problem: given a permutation, count the number of inversions. This can be done easily in $O(n^2)$ time. Can we do better?

Counting Inversions

Count inversions on lower half
Count inversions on upper half
Count the inversions between the halves

Count the Inversions

Indicate the number of inversions for each element detected when merging.

Problem – how do we count inversions between sub problems in $O(n)$ time?

Solution – Count inversions while merging

Counting inversions while merging

Standard merge algorithm – add to inversion count when an element is moved from the upper array to the solution.
Inversions

Counting inversions between two sorted lists

\( O(1) \) per element to count inversions

- \[ X \ x \ x \ x \ x \ x \ x \ y \ y \ y \ y \ y \ y \ y \]
- \[ z \ z \ z \ z \ z \ z \ z \ z \ z \ z \ z \]

Algorithm summary
Satisfies the "Standard recurrence"
\( T(n) = 2T(n/2) + cn \)

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?

Closest pair of points: 1 dimensional version

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 calculate distances between all pairs: exploit geometry + ordering

Closest pair of points: 2 dimensional version

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.

**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.

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 $< \delta$.

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 < δ.

Observation: suffices to consider points within δ of line L.

Almost the one-D problem again: Sort points in 2δ-strip by their y coordinate.

Def. Let s_i have the i-th smallest y-coordinate among points in the 2δ-width-strip.

Claim. If |i − j| > 8, then the distance between s_i and s_j is > δ.

Pf: No two points lie in the same ½δ-by-½δ box:

\[
\sqrt{\frac{1}{2}, \frac{1}{2}} = \sqrt{\frac{1}{2} - \frac{1}{2} < 1}
\]

so ≤ 8 boxes within +δ of y(s_i).

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

\[
D(n) = \begin{cases} 0 & n = 1 \frac{n+1}{2} \times 7n & n > 1 \end{cases} \Rightarrow D(n) = O(n \log n)
\]

BUT – that's only the number of distance calculations

What if we counted comparisons?

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

\[
C(n) = \begin{cases} 0 & n = 1 \frac{2C(n/2) + 7n}{n+1} \times n = 1 \end{cases} \Rightarrow C(n) = O(n \log^2 n)
\]

for some constant C

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 δ and list of all points sorted by y.

Sort by merging two pre-sorted lists.

\[
T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)
\]

Closest Pair Algorithm

Closest-Pair(p_1, ..., p_n) {
  if(n <= ??) return ??
  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.
}
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: $n^2 / (18 \times \log n)$</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)

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

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, …, p_n) {
  if(n <= 1) return ∞
  Compute separation line L such that half the points are on one side and half on the other side.
  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:
comparisons

Base Case
One recursive level

Recursive calls (2)
Basic operations at this recursive level

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

D(n) = \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?

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

C(n) = \begin{cases} 2C(n/2) + 4n \log n & n ≥ 1 \\ 0 & n = 1 \end{cases} \Rightarrow C(n) = O(n \log^2 n)

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 an list of all points sorted by y
   Sort by merging two pre-sorted lists.

\begin{align*}
T(x) &= 2T(n/2) + O(n) \\
T(n) &= O(n \log n)
\end{align*}
The "grade school" method:

\[ O(n^2) \text{ bit operations.} \]

To multiply two \( n \)-bit integers:

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:

\[ \Theta(n^2) \text{ bit operations.} \]

divide & conquer multiplication: warmup

To multiply two 2-digit integers:

Multiply four 1-digit integers. Add, shift some 2-digit integers to obtain result.

\[
\begin{align*}
x &= 10x_1 + x_0 \\
y &= 10y_1 + y_0 \\
xy &= (10x_1 + x_0)(10y_1 + y_0) \\
&= 100x_1y_1 + 10(x_1y_0 + x_0y_1) + x_0y_0
\end{align*}
\]

Same idea works for long integers – can split them into 4 half-sized ints

Karatsuba multiplication

To multiply two \( n \)-bit integers:

Add two \( \frac{n}{2} \) bit integers, and shift to obtain result.

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

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

\[
\begin{align*}
\alpha &= x_1 \times y_1 \\
\beta &= y_1 + y_0 \\
\alpha \beta &= (x_1 + x_0)(y_1 + y_0) \\
&= x_1y_1 + (x_1 + x_0)y_0 + x_0y_1 + x_0y_0 \\
(x_0y_0 + x_1y_1) &= \alpha \beta - x_1y_0 - x_0y_1
\end{align*}
\]

Theorem. [Karatsuba-Ofman, 1962] Can multiply two \( n \)-digit integers in \( O(n^{\log_23}) \) bit operations.
Karatsuba multiplication

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

Naïve: $\Theta(n^2)$
Karatsuba: $\Theta(n^{1.585})$

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

Another D&C Example: Multiplying Polynomials

Similar ideas apply to polynomial multiplication

We’ll describe the basic ideas by multiplying polynomials rather than integers
In fact, it’s somewhat simpler: no carries!

Notes on Polynomials

These are just formal sequences of coefficients so when we show something multiplied by $x^k$ it just means shifted $k$ places to the left – basically no work

Usual Polynomial Multiplication:

<table>
<thead>
<tr>
<th>$3x^2 + 2x + 2$</th>
<th>$x^2 - 3x + 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3x^4 + 2x^2 + 2x$</td>
<td></td>
</tr>
<tr>
<td>$-9x^3 - 6x^2 - 6x$</td>
<td></td>
</tr>
<tr>
<td>$3x^4 + 7x^3 - x^2 - 4x + 2$</td>
<td></td>
</tr>
</tbody>
</table>

Obvious Algorithm:

Compute all $a_i b_j$ and collect terms $\Theta(m^2)$ time

4. Polynomial Multiplication

| Given: Degree $m-1$ polynomials $P$ and $Q$ |
| $P = a_0 + a_1 x + a_2 x^2 + \ldots + a_{m-2} x^{m-2} + a_{m-1} x^{m-1}$ |
| $Q = b_0 + b_1 x + b_2 x^2 + \ldots + b_{m-2} x^{m-2} + b_{m-1} x^{m-1}$ |

Compute:

Degree $2m-2$ Polynomial $PQ$

| $PQ = a_0 b_0 + (a_0 b_1 + a_1 b_0) x + (a_0 b_2 + a_1 b_1 + a_2 b_0) x^2 + \ldots + (a_{m-2} b_{m-2} + a_{m-1} b_{m-2}) x^{2m-3} + a_{m-1} b_{m-1} x^{2m-2}$ |

Obvious Algorithm:

Compute all $a_i b_j$ and collect terms $\Theta(m^2)$ time
Naive Divide and Conquer

Assume \( m = 2k \)

\[
P = \sum_{i=0}^{k-1} a_i x^i + \sum_{i=k}^{m-1} a_i x^i
\]

\[
Q = \sum_{i=0}^{k-1} b_i x^i + \sum_{i=k}^{m-1} b_i x^i
\]

4 sub-problems of size \( k = m/2 \) plus linear combining

\[
T(m) = 4T(m/2) + cm
\]

Solution \( T(m) = O(m^2) \)

Karatsuba’s Algorithm

A better way to compute terms

Compute

\[
P_0 = P_0 Q_0
\]

\[
P_1 = P_1 Q_1
\]

\[
(Q_0 + Q_1) (P_0 + P_1)
\]

Then

\[
P_1 Q_0 + P_0 Q_1
\]

3 sub-problems of size \( m/2 \) plus \( O(m) \) work

\[
T(m) = 3T(m/2) + cm
\]

\[
T(m) = O(m^{\alpha})
\]

where

\[
\alpha = \log_2 3 = 1.585...
\]

Karatsuba: Details

\[
\text{PolyMul}(P, Q):
\]

// P, Q are length \( m = 2k \) vectors, with \( P[i], Q[i] \) being
// the coefficient of \( x^i \) in polynomials \( P, Q \) respectively.
if (\( m = 1 \)) return \( P[0] \times Q[0] \);

Let \( Pzero \) be elements \( 0.k-1 \) of \( P \); \( Pone \) be elements \( k.m-1 \)

\[
Qzero, Qone : \text{ similar}
\]

\[
\text{Prod1} = \text{PolyMul}(Pzero, Qzero);
\]

// result is a \( (2k-1) \)-vector

\[
\text{Prod2} = \text{PolyMul}(Pone, Qone);
\]

// ditto

\[
Pzero \times Pone;
\]

// add corresponding elements

\[
Qzero \times Qone;
\]

// ditto

\[
\text{Prod3} = \text{PolyMul}(Pzero, Qzero);
\]

// another \( (2k-1) \)-vector

\[
\text{Mid} = \text{Prod3} - \text{Prod1} - \text{Prod2};
\]

// subtract corr. elements

\[
R = \text{Prod1} + \text{Shift} (\text{Mid}, m/2) + \text{Shift} (\text{Prod2}, m)
\]

// a \( (2m-1) \)-vector

Return \( R \).

Multiplication – The Bottom Line

Polynomials

Naive: \( \Theta(n^2) \)

Karatsuba: \( \Theta(n^{1.585...}) \)

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

“Fast Fourier Transform”

Integers

Similar, but some ugly details re: carries, etc.

gives \( \Theta(n \log n \log \log n) \),

but mostly unused in practice

Computing the Median

Median: Given \( n \) numbers, find the number of rank \( n/2 \) (to be precise, say \( \lceil n/2 \rceil \))

Selection: given \( n \) numbers and an integer \( k \), find the \( k \)-th largest

E.g., Median is \( \lceil n/2 \rceil \)-nd largest

Median and Selection
Can find max with n-1 comparisons
Can find 2nd largest with another n-2
3rd largest with another n-3
eetc.: kth largest in O(kn)

What about k > log n?

Can we do better?

```
Select(A, k){
    Choose x from A
    S1 = {y in A | y < x}
    S2 = {y in A | y = x}
    S3 = {y in A | y > x}
    if (|S1| >= k)
        return Select(S1, k)
    else if (|S1| + |S2| >= k)
        return x
    else
        return Select(S3, k - |S1| - |S2|)
}
```

Randomized Selection

Choose the element at random
Analysis (not here) can show that the algorithm has expected run time O(n)
Sketch: a random element eliminates, on average, ~½ of the data
Although worst case is Θ(n²), albeit improbable (like Quicksort), for most purposes this is the method of choice
Worst case matters? Read on...

Deterministic Selection

What is the run time of select if we can guarantee that “choose” finds an x such that |S1| < 3n/4 and |S3| < 3n/4

BFPRT Algorithm

A very clever “choose” algorithm . . .
Split into n/5 sets of size 5
M be the set of medians of these sets
Return x = the median of M

BFPRT runtime

Split into n/5 sets of size 5
Let M be the set of medians of these sets
Choose x to be the median of M
Construct S1, S2 and S3 as above
Recursive call in S1 or S3

To show: |S1| < 3n/4, |S3| < 3n/4

n/5 + 3n/4 = 0.95n ⇒ O(n), worst case
Median of Medians

\[ x = \text{median of medians} \]

NB: conceptual; algorithm finds median(s), but does not sort

Points \( \leq x \), \( \therefore \text{NOT in S} \)
\[ \approx \frac{3n}{10} \] of them

Points \( \geq x \), \( \therefore \text{NOT in S} \)
\[ \approx \frac{3n}{10} \] of them

Bottom Line:
recursive call on \( S_1 \) or \( S_3 \) includes only about 70% of points

BFPRT Recurrence

\[ = \frac{7n}{10} \text{ points in subproblem} \]

More precisely, various fussiness:

\[ \lceil \frac{n}{5} \rceil \text{ groups, all but (possibly last) of size 5} \]

Upper/lower half of \( \geq \lceil \frac{n}{5} \rceil \) groups excluded

With some algebra, \( 3a, b, c \) such that:

\[ T(n) \leq T(7n/10 + a) + T(n/5 + b) + c \cdot n \]

Prove that \( T(n) \leq 20 \cdot c \cdot n \) for \( n > 20(a+b) \)

\[ d & c \ summary \]

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

Applications: Many.

Binary Search, Merge Sort, (Quicksort), Closest points, Integer multiply, …
**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**

\[
\text{Power}(a,n) = \begin{cases} 
1 & \text{if } n = 0 \\
 a & \text{if } n = 1 \\
 x & \text{if } n \text{ is odd} \\
 x \cdot x & \text{if } n \text{ is even} \\
\end{cases}
\]

where

\[
x = \text{Power}(a, \lfloor n/2 \rfloor) \]

**analysis**

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

\(M(n) = O(\log n)\) \hspace{1cm} (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 < word size, else also depends on length, multiply algorithm

**d & c summary**

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), counting inversions, closest points, median, integer/polynomial/matrix multiplication, FFT/convolution, exponentiation,…

**a practical application - RSA**

Instead of \(a^n\) want \(a^n \mod N\)

\(a^n \mod N = ((a^i \mod N) \cdot (a^j \mod N)) \mod N\)

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