Divide and conquer

Philip II of Macedon
Divide and conquer

1) **Divide** your problem into subproblems

2) **Solve** the subproblems recursively, that is, run the same algorithm on the subproblems (when the subproblems are very small, solve them from scratch)

3) **Combine** the solutions to the subproblems into a solution of the original problem
Divide and conquer

Recursion is “top-down” start from big problem, and make it smaller

Every divide and conquer algorithm can be written without recursion, in an iterative “bottom-up” fashion:
solve smallest subproblems, combine them, and continue

Sometimes recursion is a bit more elegant
Merge sort
Mergesort (low, high) {
    if (high-low <= 1) return; // Smallest subproblems

    // Divide into subproblems low..split and split..high
    split = (low+high) / 2;

    MergeSort(low, split); // Solve subproblem recursively
    MergeSort(split+1, high); // Solve subproblem recursively

    // Combine solutions
    merge sorted sequences low..split and split+1..high into
    the single sorted sequence low..high
}

Merge example

Merge sorted sequences A1 and A2 into B

A1 = [ 3  8  10  21  57 ]
  . . . . .

A2 = [ 7  13  14  17 ]
  . . . .

B =   [ 3  7  8  10  13  14  17  21  57 ]
  . . . . . . .
Mergesort (low, high) {
    if (high-low <= 1) return;
    split = (low+high) / 2;
    MergeSort(low, split);
    MergeSort(split+1, high);
    Merge
    }

Merge A1[1..s1], A2[1..s2] into B[1..(s1+s2)]

    i1=i2=j=1;
    while i1 < s1 and i2 < s2
        if (A1[i1] < A2[i2])
            B[j++] = A1[i1++]
        else
            B[j++] = A2[i2++]
    end while;

    Put what left in A1 or A2 in B
Analysis of running time

Merging $A_1[1..s_1]$, $A_2[1..s_2]$ into $B[1..(s_1+s_2)]$ takes time $\Omega$.

```
MergeSort(low, high) {
    if (high-low <= 1) return;
    split = (low+high) / 2;
    MergeSort(low, split);
    MergeSort(split+1, high);
    Merge low..split and split+1 ..high
}
```
Analysis of running time

Merging A1[1..s1], A2[1..s2] into B[1..(s1+s2)] takes time $c \cdot (s_1 + s_2)$ for some constant $c$

Let $T(n)$ be time for merge sort on $A[1..n]$

Recurrence relation $T(n) = ?$

```java
public void MergeSort(int low, int high) {
    if (high-low <= 1) return;
    int split = (low+high) / 2;
    MergeSort(low, split);
    MergeSort(split+1, high);
    Merge low..split and split+1 ..high
}
```
Analysis of running time

Merging A1[1..s1], A2[1..s2] into B[1..(s1+s2)] takes time $c \cdot (s1+s2)$ for some constant $c$

Let $T(n)$ be time for merge sort on $A[1..n]$

Recurrence relation $T(n) = 2 \cdot T(n/2) + c \cdot n$

```c
MergeSort(low, high) {
    if (high-low <= 1) return;
    split = (low+high) / 2;
    MergeSort(low, split);
    MergeSort(split+1, high);
    Merge low..split and split+1 ..high
}
```
Solving recurrence $T(n) = 2T(n/2) + cn$

Expand recurrence to obtain recursion tree

Sum of costs at level $i$ is ?
Solving recurrence $T(n) = 2T(n/2) + cn$

Expand recurrence to obtain recursion tree

Sum of costs at level $i$ is $2^i \cdot \frac{cn}{2^i} = cn$

Numbers of levels is?
Analysis of space

How many extra array elements we need?

At least $n$ to merge

It can be implemented to use $O(n)$ space.
Quick sort
QuickSort(lo, hi) { // Sorts array A
    if (hi-lo ≤ 1) return;
    partition(lo, hi) and return split;
    QuickSort(lo, split-1);
    QuickSort(split+1, hi);
}

Partition permutes A[lo..hi] so that
    each element in A[lo.. split] is ≤ A[split],
    each element in A[split+1.. hi] is > A[split].
Partition(A[lo.. hi]) For simplicity, assume distinct elements

Pick pivot index \( p \). // We will explain later how

Swap \( A[p] \) and \( A[hi] \); \( i = lo-1; j = hi; \)
  Do \( i++ \) while \( A[i] < A[hi] \);
  If \( i < j \) then swap \( A[i] \) and \( A[j] \)
  Else {
    swap \( A[i] \) and \( A[hi] \); return \( i \)
  }
}

Running time: \( O(hi - lo) \)
Analysis of running time

$T(n) = \text{number of comparisons on an array of length } n.$

$T(n)$ depends on the choice of the pivot index $p$

- Choosing pivot deterministically
- Choosing pivot randomly

```c
QuickSort(lo, hi) {
    if (hi-lo <= 1) return;
    partition(lo, hi) and return split,
    QuickSort(lo, split-1);
    QuickSort(split+1, hi);
}
```
Analysis of running time

$T(n) = \text{number of comparisons on an array of length } n.$

- Choosing pivot deterministically:
  
  the worst case happens when one sub-array is empty and the other is of size $n-1$, in this case:
  
  $T(n) = T(n-1) + T(0) + cn$

  
  $= ?$

  
  $cn + c(n-1) + c(n-2) + ...$
Analysis of running time

\[ T(n) = \text{number of comparisons on an array of length } n. \]

- Choosing pivot deterministically:
  
  the worst case happens when one sub-array is empty and the other is of size n-1, in this case:
  
  \[ T(n) = T(n-1) + T(0) + c \cdot n \]
  
  \[ = \Theta(n^2). \]

- Choosing pivot randomly we can guarantee

  \[ T(n) = O(n \log n) \text{ with high probability} \]

  \[ \text*{Expected number of comparisons is } O(n \log n). \]

  \[ \text{Space: } O(1). \]
Randomized-Quick sort:
R-QuickSort(low, high) {
    if (high-low ≤ 1) return;
    R-partition(low, high) and return split,
    R-QuickSort(low, split-1);
    R-QuickSort(split+1, high);
}

R-partition(low, high)
    Pick pivot index \( p \) uniformly in \{low, low+1, \ldots \) high\}
    Then partition as before

We bound the total time spent by Partition
• **Definition**: $X$ is the number of comparisons

• Next we bound the **expectation** of $X$, $E[X]$
Rename array A as $z_1, z_2, \ldots, z_n$, with $z_i$ being the $i$-th smallest.

Note: each pair of elements $z_i, z_j$ is compared at most once. Why?
• Rename array $A$ as $z_1, z_2, \ldots, z_n$, with $z_i$ being the $i$-th smallest

• Note: each pair of elements $z_i, z_j$ is compared at most once.
  Elements are compared with the **pivot**.
  An element is a pivot at most once.

• Define indicator random variables
  $X_{ij} := 1 \text{ if } \{ z_i \text{ is compared to } z_j \}$
  $X_{ij} := 0 \text{ otherwise}$

• Note: $X = ?$
- Rename array $A$ as $z_1, z_2, \ldots, z_n$, with $z_i$ being the $i$-th smallest.

- Note: each pair of elements $z_i, z_j$ is compared at most once. Elements are compared with the pivot. An element is a pivot at most once.

- Define indicator random variables $X_{ij} := 1$ if $\{z_i \text{ is compared to } z_j\}$
  $X_{ij} := 0$ otherwise

- Note: $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$. 
\[ X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}. \]

Taking expectation, and using linearity:

\[
E[X] = E \left( \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \right) \\
= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\
= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr \{ z_i \text{ is compared to } z_j \} 
\]
• Pr \{z_i \text{ is compared to } z_j\} = ?

• If some element \(y, z_i < y < z_j\) chosen as pivot, \(z_i\) and \(z_j\) cannot be compared.

  Why?
• Pr \{z_i \text{ is compared to } z_j\} = ?

• If some element \( y, z_i < y < z_j \) chosen as pivot,
  
  \( z_i \) and \( z_j \) can not be compared.
  
  Because after partition \( z_i \) and \( z_j \) will be in two different parts.

• Definition: \( Z_{ij} \) is \( = \{ z_i, z_{i+1}, \ldots, z_j \} \)

• \( z_i \) and \( z_j \) are compared if
  
  first element chosen as pivot from \( Z_{ij} \) is either \( z_i \) or \( z_j \).
\[ \Pr \{ z_i \text{ is compared to } z_j \} = \Pr [ z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij} ] \]
Pr \{z_i \text{ is compared to } z_j\} = Pr [z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}] \\
= Pr [z_j \text{ is first pivot chosen from } Z_{ij}] \\
+ \ Pr [z_i \text{ is first pivot chosen from } Z_{ij}]
Pr \{z_i \text{ is compared to } z_j \}\} = \Pr [z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}] \\
= \Pr [z_i \text{ is first pivot chosen from } Z_{ij}] \\
\quad + \Pr [z_j \text{ is first pivot chosen from } Z_{ij}] \\
= 1/(j-i+1) + 1/(j-i+1) = 2/(j-i+1).
\[
\Pr \{z_i \text{ is compared to } z_j\} = \Pr \{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\}
\]
\[
= \Pr \{z_i \text{ is first pivot chosen from } Z_{ij}\}
+ \Pr \{z_j \text{ is first pivot chosen from } Z_{ij}\}
\]
\[
= \frac{1}{(j-i+1)} + \frac{1}{(j-i+1)} = \frac{2}{(j-i+1)}.
\]

\[
E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr \{z_i \text{ is compared to } z_j\}
\]
\[
= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{(j-i+1)}.
\]
Pr \{z_i \text{ is compared to } z_j\} = Pr [z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}] \\
= Pr [z_i \text{ is first pivot chosen from } Z_{ij}] \\
\quad + Pr [z_j \text{ is first pivot chosen from } Z_{ij}] \\
= \frac{1}{(j-i+1)} + \frac{1}{(j-i+1)} = \frac{2}{(j-i+1)}. \\
E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr \{z_i \text{ is compared to } z_j\} \\
= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{(j-i+1)} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{(k+1)} \\
< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}
Pr \{z_i \text{ is compared to } z_j\} = Pr \{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\}

= Pr \{z_i \text{ is first pivot chosen from } Z_{ij}\}

+ Pr \{z_j \text{ is first pivot chosen from } Z_{ij}\}

= \frac{1}{(j-i+1)} + \frac{1}{(j-i+1)} = \frac{2}{(j-i+1)}.

\[E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr \{z_i \text{ is compared to } z_j\}\]

= \sum_{i=1}^{n-1} \left( \sum_{j=i+1}^{n} \frac{2}{(j-i+1)} \right)

= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{(k+1)}

= \sum_{i=1}^{n-1} \sum_{k=1}^{n-1} \frac{2}{k} = \sum_{i=1}^{O(\log n)} = O(n \log n).

Expected running time of Randomized-QuickSort is \(O(n \log n)\).
An application of Markov's inequality

Let $T$ be the running time of Randomized Quick sort.

We just proved $E[T] \leq c \cdot n \log n$, for some constant $c$.

Hence, $\Pr[T > 100 \cdot c \cdot n \log n] < ?$
An application of Markov's inequality

Let $T$ be the running time of Randomized Quick sort.

We just proved $\mathbb{E}[T] \leq c n \log n$, for some constant $c$.

Hence, $\Pr[T > 100 c n \log n] < 1/100$

Markov's inequality useful to translate bounds on the expectation in bounds of the form: “It is unlikely the algorithm will take too long.”
Oblivious Sorting

Want an algorithm that only accesses the input via

\textbf{Compare-exchange}(x,y)

Compares \( A[x] \) and \( A[y] \) and swaps them if necessary

We call such algorithms \textit{oblivious}. Useful if you want to sort with a (non-programmable) piece of hardware

Did we see any oblivious algorithms?
Oblivious Mergesort

This is just like Merge sort except that the merge subroutine is replaced with a subroutine whose comparisons do not depend on the input.

Assumption:

Size of the input sequence, $n$, is a power of 2.

Convenient to index from 0 to $n-1$
Oblivious-Mergesort (A[0..n-1])
{
    if n > 1 then
        Oblivious-Mergesort(A[0.. n/2-1]);
        Oblivious-Mergesort(A [n/2 .. n-1]);
        odd-even-Merge(A[0..n-1]);
    }

Same structure as Mergesort

But Odd-even-merge is more complicated, recursive
odd-even-merge(A[0..n-1]); {
  if n = 2 then compare-exchange(0,1);
  else {
    odd-even-merge(A[0,2 .. n-2]); //even subsequence
    odd-even-merge(A[1,3,5 .. n-1]); //odd subsequence
    for i ∈ {1,3,5, ... n-1} do
      compare-exchange(i, i +1);
  }
}

Compare-exchange(x,y) compares A[x] and A[y] and swaps them if necessary

- Merges correctly if A[0.. n/2-1] and A[n/2 .. n-1] are sorted
odd-even-merge(A[0..n-1]);
if n = 2 then compare-exchange(0,1);
else
  odd-even-merge(A[0,2 .. n-2]);
  odd-even-merge(A[1,3,5 .. n-1]);
  for i ∈ {1,3,5, ... n-1} do
    compare-exchange(i, i+1);

0-1 principle: If algorithm works correctly on sequences of 0 and 1, then it works correctly on all sequences

True when input only accessed through compare-exchange
odd-even-merge(A[0..n-1]);
if $n = 2$ then compare-exchange(0,1);
else
    odd-even-merge(A[0,2 .. n-2]);
    odd-even-merge(A[1,3,5 .. n-1]);
for $i \in \{1,3,5, \ldots, n-1\}$ do
    compare-exchange(i, i+1);

Critical Observation: Number of 1's in columns $C_1$ & $C_2$ is within 2.
Analysis of running time

\[ T(n) = \text{number of comparisons.} \]

\[ T(n) = 2T(n/2) + T'(n) = 2T(n/2) + T(n/2) + c n = 2T'(n/2) + c n \]

T'(n) = number of operations in odd-even-merge

Oblivious-Mergesort(A[0..n-1])
if n > 1 then
Oblivious-Mergesort(A[0.. n/2-1]);
Oblivious-Mergesort(A [n/2 .. n-1]);
Odd-even-merge(A[0..n-1]);
if n = 2 then
compare-exchange(0,1);
else
odd-even-merge(A[0,2 .. n-2]);
odd-even-merge(A[1,3,5 .. n-1]);
for i \in \{1,3,5, \ldots n-1\} do
compare-exchange(i, i +1);
Analysis of running time

\[ T(n) = \text{number of comparisons}. \]
\[ = 2T(n/2) + T'(n) \]
\[ = 2T(n/2) + O(n \log n). \]
\[ = ? \]

\[ T'(n) = \text{number of operations in odd-even-merge} \]
\[ = 2T'(n/2) + c n = O(n \log n). \]

Oblivious-Mergesort(A[0..n-1])
if n > 1 then
Oblivious-Mergesort(A[0.. n/2-1]);
Oblivious-Mergesort(A [n/2 .. n-1]);
Odd-even-merge(A[0..n-1]);

odd-even-merge(A[0..n-1]);
if n = 2 then
  compare-exchange(0,1);
else
  odd-even-merge(A[0,2 .. n-2]);
  odd-even-merge(A[1,3,5 .. n-1]);
  for i \in \{1,3,5, \ldots n-1\} do
    compare-exchange(i, i +1);
Analysis of running time

T(n) = number of comparisons.

= 2T(n/2) + T'(n)
= 2T(n/2) + O(n \log n)
= O(n \log^2 n).

Oblivious-Mergesort(A[0..n-1])
if n > 1 then
   Oblivious-Mergesort(A[0.. n/2-1]);
   Oblivious-Mergesort(A [n/2 .. n-1]);
   Odd-even-merge(A[0..n-1]);
else
   odd-even-merge(A[0,2 .. n-2]);
   odd-even-merge(A[1,3,5 .. n-1]);
   for i ∈ {1,3,5, … n-1} do
      compare-exchange(i, i +1);
<table>
<thead>
<tr>
<th>Sorting algorithm</th>
<th>Time</th>
<th>Space</th>
<th>Assumption/Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble sort</td>
<td>$\Theta(n^2)$</td>
<td>$O(1)$</td>
<td>Easy to code</td>
</tr>
<tr>
<td>Counting sort</td>
<td>$\Theta(n+k)$</td>
<td>$O(n+k)$</td>
<td>Input range is [0..k]</td>
</tr>
<tr>
<td>Radix sort</td>
<td>$\Theta(d(n+k))$</td>
<td>$O(n+k)$</td>
<td>Inputs are d-digit integers in base k</td>
</tr>
<tr>
<td>Quick sort (deterministic)</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>Quick sort (Randomized)</td>
<td>$O(n \log n)$</td>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>Merge sort</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>Oblivious merge sort</td>
<td>$O(n \log^2 n)$</td>
<td>$O(1)$</td>
<td>Comparisons are independent of input</td>
</tr>
</tbody>
</table>
Sorting is still open!

- Input: n integers in \( \{0, 1, \ldots, 2^w - 1\} \)

- Model: Usual operations (+, *, AND, ...) on w-bit integers in constant time

- Open question: Can you sort in time \( O(n) \)?

- Best known time: \( O(n \log \log n) \)
Next

- View other divide-and-conquer algorithms
- Some related to sorting
Selecting h-th smallest element

- **Definition**: For array \( A[1..n] \) and index \( h \),
  \[ S(A, h) := \text{h-th smallest element in } A, \]
  \[ = B[h] \text{ for } B = \text{sorted version of } A \]

- \( S(A, (n+1)/2) \) is the **median** of \( A \), when \( n \) is odd

- We show how to compute \( S(A, h) \) with \( O(n) \) comparisons
Computing $S(A,h)$

- Divide array in consecutive blocks of 5:
  $A[1..5], A[6..10], A[11..15], ...$

- Find median of each
  $m_1 = S(A[1..5],3), m_2 = S(A[6..10],3), m_3 = S(A[11..15],3)$

- Find median of medians, $x = S([m_1, m_2, ..., m_{n/5}], (n/5+1)/2)$

- Partition $A$ according to $x$. Let $x$ be in position $k$

- If $h = k$ return $x$, if $h < k$ return $S(A[1..k-1],h)$,
  if $h > k$ return $S(A[k+1..n],h-k-1)$
• Divide array in consecutive blocks of 5
• Find median of each
  \[ m_1 = S(A[1..5],3), \ m_2 = S(A[6..10],3), \ m_3 = S(A[11..15],3) \]
• Find median of medians, \[ x = S([m_1, m_2, ..., m_{n/5}], (n/5+1)/2) \]
• Partition \( A \) according to \( x \). Let \( x \) be in position \( k \)
  • If \( h = k \) return \( x \), if \( h < k \) return \( S(A[1..k-1],h) \),
    if \( h > k \) return \( S(A[k+1..n],h-k-1) \)

• Running time:
  When partition, half the medians \( m_i \) will be \( \geq x \).
  Each contributes \( \geq ? \) elements from their 5.
Divide array in consecutive blocks of 5

Find median of each

\[ m_1 = S(A[1..5],3), \quad m_2 = S(A[6..10],3), \quad m_3 = S(A[11..15],3) \]

Find median of medians, \( x = S([m_1, m_2, \ldots, m_{n/5}], (n/5+1)/2) \)

Partition \( A \) according to \( x \). Let \( x \) be in position \( k \)

If \( h = k \) return \( x \), if \( h < k \) return \( S(A[1..k-1],h) \),

\[ \text{if } h > k \text{ return } S(A[k+1..n],h-k-1) \]

Running time:

When partition, half the medians \( m_i \) will be \( \geq x \).
Each contributes \( \geq 3 \) elements from their 5.
So we recurse on \( \leq \) ???
- Divide array in consecutive blocks of 5
- Find median of each
  \[ m_1 = S(A[1..5], 3), \ m_2 = S(A[6..10], 3), \ m_3 = S(A[11..15], 3) \]
- Find median of medians, \( x = S([m_1, m_2, ..., m_{n/5}], (n/5+1)/2) \)
- Partition \( A \) according to \( x \). Let \( x \) be in position \( k \)
  - If \( h = k \) return \( x \), if \( h < k \) return \( S(A[1..k-1], h) \),
    - if \( h > k \) return \( S(A[k+1..n], h-k-1) \)
- Running time:
  When partition, half the medians \( m_i \) will be \( \geq x \).
  Each contributes \( \geq 3 \) elements from their 5.
  So we recurse on \( \leq 7n/10 \) elements
  \[ T(n) \leq T(n/5) + T(7n/10) + O(n) \]
  This implies \( T(n) = O(n) \)
How to solve recurrence $T(n) \leq T(n/5) + T(7n/10) + cn$

Guess $T(n) \leq an$, for some constant $a$

Does recurrence hold for guess?

$an \leq an/5 + a7n/10 + cn$

$\iff$ \hspace{1em} (divide by $an$)

$1 \leq 1/5 + 7/10 + c/a$

Because $1/5 + 7/10 = 9/10 < 1$, this is true for large enough $a$. \hspace{1em} $\square$
Closest pair of points

Input:
Set $P$ of $n$ points in the plane

Output:
Two points $x_1$ and $x_2$ with the shortest (Euclidean) distance from each other.

**Trivial algorithm:** Compute every distance: $\Omega(n^2)$ time

**Next:** Clever algorithm with $O(n \log(n))$ time
Closest pair of points

Input:
Set $P$ of $n$ points in the plane

Output:
Two points $x_1$ and $x_2$ with the shortest (Euclidean) distance from each other.

- For the following algorithm we assume that we have two arrays $X$ and $Y$, each containing all the points of $P$.
- $X$ is sorted so that the $x$-coordinates are increasing
- $Y$ is sorted so that $y$-coordinates are increasing.
Closest pair of points

Divide:
Closest pair of points

Divide: find a vertical line \( L \) that bisects \( P \) into two sets

\[ P_L := \{ \text{points in } P \text{ that are on } L \text{ or to the left of } L \} \]

\[ P_R := \{ \text{points in } P \text{ that are to the right of } L \} \]

Such that \( |P_L| = n/2 \) and \( |P_R| = n/2 \) (plus or minus 1)

Conquer:
Closest pair of points

**Divide:** find a vertical line $L$ that bisects $P$ into two sets

$P_L := \{ \text{points in } P \text{ that are on } L \text{ or to the left of } L \}.$

$P_R := \{ \text{points in } P \text{ that are to the right of } L \}.$

Such that $|P_L| = n/2$ and $|P_R| = n/2$ (plus or minus 1)

**Conquer:** Make two recursive calls to find the closest pair of points in $P_L$ and $P_R$.

Let the closest distances in $P_L$ and $P_R$ be $\delta_L$ and $\delta_R$, and let $\delta = \min(\delta_L, \delta_R)$.

**Combine:**
Closest pair of points

Divide: find a vertical line \( L \) that bisects \( P \) into two sets

\[
\begin{align*}
P_L &:= \{ \text{points in } P \text{ that are on } L \text{ or to the left of } L \}. \\
P_R &:= \{ \text{points in } P \text{ that are to the right of } L \}.
\end{align*}
\]

Such that \(|P_L| = n/2\) and \(|P_R| = n/2\) (plus or minus 1)

Conquer: Make two recursive calls to find the closest pair of points in \( P_L \) and \( P_R \).

Let the closest distances in \( P_L \) and \( P_R \) be \( \delta_L \) and \( \delta_R \), and let \( \delta = \min(\delta_L, \delta_R) \).

Combine: The closest pair is either the one with distance \( \delta \) or it is a pair with one point in \( P_L \) and the other in \( P_R \) with distance less than \( \delta \), NO SAVING?
Closest pair of points

**Divide:** find a vertical line $L$ that bisects $P$ into two sets

$P_L := \{ \text{points in } P \text{ that are on } L \text{ or to the left of } L \}$. 

$P_R := \{ \text{points in } P \text{ that are to the right of } L \}$. 

Such that $|P_L| = n/2$ and $|P_R| = n/2$ (plus or minus 1)

**Conquer:** Make two recursive calls to find the closest pair of points in $P_L$ and $P_R$.

Let the closest distances in $P_L$ and $P_R$ be $\delta_L$ and $\delta_R$, and let $\delta = \min(\delta_L, \delta_R)$.

**Combine:** The closest pair is either the one with distance $\delta$ or it is a pair with one point in $P_L$ and the other in $P_R$ with distance less than $\delta$, in a $\delta \times 2\delta$ box straddling $L$. 
How to find points in the box

- Create $Y'$ by removing from $Y$ points that are not in $2\delta$-wide vertical strip.
How to find points in the box

- Create $Y'$ by removing from $Y$ points that are not in $2\delta$-wide vertical strip.
How to find points in the box

- Create $Y'$ by removing from $Y$ points that are not in $2\delta$-wide vertical strip.
How to find points in the box

- Create $Y'$ by removing from $Y$ points that are not in $2\delta$-wide vertical strip.
- For each consecutive 8 points in $Y'$
  $p_1, p_2, \ldots, p_8$
  compute all their distances.
- If any of them are closer than $\delta$,
  update the closest pair
  and the shortest distance $\delta$.
- Return $\delta$ and the closest pair.
Why 8?

Fact: If there are 9 points in a $\delta \times 2\delta$ box straddling $L$.

$\Rightarrow$ there are 5 points in a $\delta \times \delta$ box on one side of $L$.

$\Rightarrow$ there are 2 points on one side of $L$ with distance less than $\delta$.

This violates the definition of $\delta$. 
Analysis of running time

Same as Merge sort:

T(n) = number of operations
T(n) = 2 T(n/2) + c n
= O(n \log n).
Is multiplication harder than addition?

Alan Cobham, < 1964
Is multiplication harder than addition?

Alan Cobham, < 1964

We still do not know!
Addition

Input: two n-digit integers a, b in base w

Output: One integer c = a + b.

Operations allowed: only on digits

The simple way to add takes ?
Addition

Input: two n-digit integers a, b in base w

(think w = 2, 10)

Output: One integer c = a + b.

Operations allowed: only on digits

The simple way to add takes O(n)

optimal?
Addition

Input: two n-digit integers a, b in base w

\[ \text{think } w = 2, 10 \]

Output: One integer \( c = a + b \).

Operations allowed: only on digits

The simple way to add takes \( O(n) \)

This is optimal, since we need at least to write \( c \)
Multiplication

Input: two n-digit integers a, b in base w

(Think w = 2, 10)

Output: One integer c = a \cdot b.

Operations allowed: only on digits

Simple way takes?

\[
\begin{array}{c}
23958233 \\
5830 \times \\
\hline
00000000 ( = 23,958,233 \times 0) \\
71874699 ( = 23,958,233 \times 30) \\
191665864 ( = 23,958,233 \times 800) \\
119791165 ( = 23,958,233 \times 5,000) \\
\hline
139676498390 ( = 139,676,498,390 )
\end{array}
\]
Multiplication

Input: two n-digit integers \(a, b\) in base \(w\)

\[(\text{think } w = 2, 10)\]

Output: One integer \(c = a \cdot b\).

Operations allowed: only on digits

The simple way to multiply takes \(\Omega(n^2)\)

Can we do this any faster?
Can we multiply faster than \( n^2 \) ?

Feeling: “As regards number systems and calculation techniques, it seems that the final and best solutions were found in science long ago”

In 1950’s, Kolmogorov conjectured \( \Omega(n^2) \)

One week later, \( O(n^{1.59}) \) time by Karatsuba

See “The complexity of Computations”
Can we multiply faster than $n^2$?

Feeling: “As regards number systems and calculation techniques, it seems that the final and best solutions were found in science long ago”

In 1950’s, Kolmogorov conjectured $Ω(n^2)$.

One week later, $O(n^{1.59})$ time by Karatsuba.

See “The complexity of Computations”
Multiplication

Example:
2-digit numbers $N_1$ and $N_2$ in base $w$.

$N_1 = a_0 + a_1 w$.

$N_2 = b_0 + b_1 w$.

For this example, think $w$ very large, like $w = 2^{32}$.
Multiplication

Example:

2-digit numbers $N_1$ and $N_2$ in base $w$.

$N_1 = a_0 + a_1 w$

$N_2 = b_0 + b_1 w$

$P = N_1 N_2$

$= a_0 b_0 + (a_0 b_1 + a_1 b_0)w + a_1 b_1 w^2$

$= p_0 + p_1 w + p_2 w^2$.

This can be done with 3 multiplications.
Multiplication

Example:
2-digit numbers $N_1$ and $N_2$ in base $w$.

$N_1 = a_0 + a_1w$.

$N_2 = b_0 + b_1w$.

$P = N_1 N_2$

$= a_0 b_0 + (a_0 b_1 + a_1 b_0)w + a_1 b_1 w^2$

$= p_0 + p_1w + p_2w^2$.

This can be done with 4 multiplications.

Can we save multiplications, possibly increasing additions?
Compute

\[ P = a_0 b_0 + (a_0 b_1 + a_1 b_0)w + a_1 b_1 w^2 \]

\[ = p_0 + p_1 w + p_2 w^2. \]

\[ q_0 = a_0 b_0. \]
\[ q_1 = (a_0 + a_1)(b_1 + b_0). \]
\[ q_2 = a_1 b_1. \]

Note:

\[ q_0 = p_0. \]
\[ \uparrow \quad \Rightarrow \]
\[ p_0 = q_0. \]

\[ q_1 = p_1 + p_0 + p_2. \]
\[ \Rightarrow \quad p_1 = q_1 - q_0 - q_2. \]

\[ q_2 = p_2. \]
\[ \Rightarrow \quad p_2 = q_2. \]

So the three digits of \( P \) are evaluated using 3 multiplications rather than 4.

What to do for larger numbers?
The Karatsuba algorithm
Input: two n-digit integers a, b in base w.
Output: One integer c = a \cdot b.

Divide:
   How?
The Karatsuba algorithm

Input: two n-digit integers $a$, $b$ in base $w$.

Output: One integer $c = a \cdot b$.

Divide:

$m = n/2.$

$a = a_0 + a_1 w^m.$

$b = b_0 + b_1 w^m.$

$$a \cdot b = a_0 b_0 + (a_0 b_1 + a_1 b_0) w^m + a_1 b_1 w^{2m}$$

$$= p_0 + p_1 w^m + p_2 w^{2m}$$
The Karatsuba algorithm

Input: two n-digit integers $a$, $b$ in base $w$.

Output: One integer $c = a \cdot b$.

Divide:

$$m = \frac{n}{2}.$$ $a = a_0 + a_1 w^m$.

$$b = b_0 + b_1 w^m.$$ $a \cdot b = a_0 b_0 + (a_0 b_1 + a_1 b_0) w^m + a_1 b_1 w^{2m} = p_0 + p_1 w^m + p_2 w^{2m}$

Conquer:

$q_0 = a_0 \times b_0$. Each $\times$ is a recursive call

$q_1 = (a_0 + a_1) \times (b_1 + b_0)$. $q_2 = a_1 \times b_1$
The Karatsuba algorithm

Input: two n-digit integers a, b in base w.

Output: One integer c = a⋅b.

Divide:

\[ m = \frac{n}{2}. \]
\[ a = a_0 + a_1 w^m. \]
\[ b = b_0 + b_1 w^m. \]

Conquer:

\[ q_0 = a_0 \times b_0. \]
\[ q_1 = (a_0 + a_1) \times (b_1 + b_0). \]
\[ q_2 = a_1 \times b_1. \]

Each \( \times \) is a recursive call

Combine:

\[ p_0 = q_0. \]
\[ p_1 = q_1 - q_0 - q_2. \]
\[ p_2 = q_2. \]

\[ a \cdot b = a_0 b_0 + (a_0 b_1 + a_1 b_0) w^m + a_1 b_1 w^{2m} = p_0 + p_1 w^m + p_2 w^{2m} \]
Analysis of running time

$T(n) = \text{number of operations.}$

$T(n) = 3 \cdot T(n/2) + O(n)$

$\quad = ?$
Analysis of running time

\[ T(n) = \text{number of operations.} \]

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

\[ = ? \]

Recursion tree

Cost at level \( i = cn \left( \frac{3}{2} \right)^i \)

Number of levels = \( \log_2(n) \)

Total cost = \( \sum_{i=0}^{\log_2 n} cn \left( \frac{3}{2} \right)^i = O \left( n \left( \frac{3}{2} \right)^{\log_2 n} \right) = O(n^{\log_2 3}) \)
Analysis of running time

\[ T(n) = \text{number of operations.} \]

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

\[ = \Theta(n \log 3) \quad \text{(log in base 2)} \]

\[ = O(n^{1.59}). \]

Karatsuba may be used in your computers to reduce, say, multiplication of 128-bit integers to 64-bit integers.

Are there faster algorithms for multiplication?
Algorithms taking essentially $O(n \log n)$ are known.

1971: Schöning-Strassen $O(n \log n \log \log n)$

2007: Furer $O(n \log n \exp(\log^* n))$

$log^*n = \text{times you need to apply log to n to make it 1}$

They are all based on Fast Fourier Transform
Matrix Multiplication

\(n \times n\) matrices. Note input length is \(n^2\)

\[
A = \begin{pmatrix}
0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1
\end{pmatrix}
\quad \cdot \quad
B = \begin{pmatrix}
1 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 \\
0 & 1 & 1 & 1 \\
0 & 1 & 0 & 0
\end{pmatrix}
= \begin{pmatrix}
\text{\textcolor{red}{1}}
\end{pmatrix}
\]

Just to write down output need time \(\Omega(n^2)\)

The simple way to do matrix multiplication takes ?
Matrix Multiplication

$n \times n$ matrixes. Note input length is $n^2$

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

Just to write down output need time $\Omega(n^2)$

The simple way to do matrix multiplication takes $O(n^3)$. 
Strassen's Matrix Multiplication

Input: two $n \times n$ matrices $A$, $B$.
Output: One $n \times n$ matrix $C = A \cdot B$. 
**Strassen's Matrix Multiplication**

**Divide:**

Divide each of the input matrices $A$ and $B$ into 4 matrices of size $n/2 \times n/2$, as follows:

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}
\]

\[
A.B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}
\]
Strassen's Matrix Multiplication

Conquer:

Compute the following 7 products:

\[ M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) \cdot \]
\[ M_2 = (A_{21} + A_{22})B_{11} \cdot \]
\[ M_3 = A_{11}(B_{12} - B_{22}) \cdot \]
\[ M_4 = A_{22}(B_{21} - B_{11}) \cdot \]
\[ M_5 = (A_{11} + A_{12})B_{22} \cdot \]
\[ M_6 = (A_{21} - A_{11})(B_{11} - B_{12}) \cdot \]
\[ M_7 = (A_{12} - A_{22})(B_{21} - B_{22}) \cdot \]
Strassen's Matrix Multiplication

Combine:

\[ C_{11} = M_1 + M_4 - M_5 + M_7. \]
\[ C_{12} = M_3 + M_5. \]
\[ C_{21} = M_2 + M_4. \]
\[ C_{22} = M_1 - M_2 + M_3 + M_6. \]

\[
C = \begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}
\]
Analysis of running time

$T(n) = \text{number of operations}$

$T(n) = 7 \, T(n/2) + 18 \{\text{Time to do matrix addition}\}$

$= 7 \, T(n/2) + \Theta(n^2)$

$= ?$
Analysis of running time

\[ T(n) = \text{number of operations} \]

\[ T(n) = 7 \ T(n/2) + 18 \ \{\text{Time to do matrix addition}\} \]

\[ = 7 \ T(n/2) + \Theta(n^2) \]

\[ = \Theta(n \log 7) \]

\[ = O(n^{2.81}). \]
**Definition:** $\omega$ is the smallest number such that multiplication of $n \times n$ matrices can be computed in time $n^{\omega+\varepsilon}$ for every $\varepsilon > 0$

**Meaning:** time $n^{\omega}$ up to lower-order factors

$\omega \geq 2$ because you need to write the output

$\omega < 2.81$ Strassen, just seen

$\omega < 2.38$ state of the art

Determining $\omega$ is a prominent problem
Fast Fourier Transform (FFT)

We start with the most basic case
Walsh-Hadamard transform

Hadamard $2^i \times 2^i$ matrix $H_i$:

$$H_0 = [1]$$

$$H_{i+1} = \begin{pmatrix} H_i & H_i \\ H_i & -H_i \end{pmatrix}$$

Problem: Given vector $x$ of length $n = 2^k$, compute $H_k x$

Trivial: $O(n^2)$

Next: $O(n \log n)$
Walsh-Hadamard transform

Write $x = [y \ z]^T$, and note that $H_{k+1} x =$

$$\begin{bmatrix} H_k y + H_k z \\ H_k y - H_k z \end{bmatrix}$$

This gives $T(n) = ?$
Walsh-Hadamard transform

Write $x = [y \ z]^T$, and note that $H_{k+1} x =$

$$
\begin{pmatrix}
H_k y + H_k z \\
H_k y - H_k z
\end{pmatrix}
$$

This gives $T(n) = 2 \ T(n/2) + O(n) = O(n \log n)$
Polynomials and Fast Fourier Transform (FFT)
Polynomials

\[ A(x) = \sum_{i=0}^{n-1} a_i x^i \quad \text{a polynomial of degree n-1} \]

Evaluate at a point \( x = b \) with how many multiplications?

2n trivial
Polynomials

\[ A(x) = \sum_{i=0}^{n-1} a_i x^i \quad \text{a polynomial of degree } n-1 \]

Evaluate at a point \( x = b \) with Horner's rule:
Compute \( a_{n-1} \),
\[ a_{n-2} + a_{n-1}x, \]
\[ a_{n-3} + a_{n-2}x + a_{n-1}x^2 \]
\[ \vdots \]

Each step: multiply by \( x \), and add a coefficient

There are \( \leq n \) steps \( \exists \quad n \) multiplications
Summing Polynomials

\[ \sum_{i=0}^{n-1} a_i x^i \]  
\[ \text{a polynomial of degree n-1} \]

\[ \sum_{i=0}^{n-1} b_i x^i \]  
\[ \text{a polynomial of degree n-1} \]

\[ \sum_{i=0}^{n-1} c_i x^i \]  
\[ \text{the sum polynomial of degree n-1} \]

\[ c_i = a_i + b_i \]

Time $O(n)$
How to multiply polynomials?

$$\sum_{i=0}^{n-1} a_i x^i$$ a polynomial of degree n-1

$$\sum_{i=0}^{n-1} b_i x^i$$ a polynomial of degree n-1

$$\sum_{i=0}^{2n-2} c_i x^i$$ the product polynomial of degree n-1

$$c_i = \sum_{j \leq i} a_j b_{i-j}$$

Trivial algorithm: time $O(n^2)$
FFT gives time $O(n \log n)$
Polynomial representations

Coefficient: \((a_0, a_1, a_2, ... a_{n-1})\)

Point-value: have points \(x_0, x_1, ... x_{n-1}\) in mind

Represent polynomials \(A(X)\) by pairs
\[
\{(x_0, y_0), (x_1, y_1), ...\}
\]
\[A(x_i) = y_i\]

To multiply in point-value, just need \(O(n)\) operations.
Approach to polynomial multiplication:

A, B given as coefficient representation

1) Convert A, B to point-value representation

2) Multiply C = AB in point-value representation

3) Convert C back to coefficient representation

2) done easily in time $O(n)$

FFT allows to do 1) and 3) in time $O(n \log n)$.

Note: For C we need $2n-1$ points; we'll just think “n”
From coefficient to point-value:

\[
\begin{align*}
\begin{bmatrix}
y_0 \\
y_1 \\
\vdots \\
\vdots \\
y_{n-1}
\end{bmatrix}
&= 
\begin{bmatrix}
1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\
1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1}
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
\vdots \\
\vdots \\
a_{n-1}
\end{bmatrix}
\end{align*}
\]

From point-value representation, note above matrix is invertible (if points distinct)

Alternatively, Lagrange's formula
We need to evaluate $A$ at points $x_1 \ldots x_n$ in time $O(n \log n)$

Idea: divide and conquer:

$$A(x) = A^0(x^2) + x A^1(x^2)$$

where $A^0$ has the even-degree terms, $A^1$ the odd

Example:  

$$A = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5$$

$$A^0(x^2) = a_0 + a_2 x^2 + a_4 x^4$$

$$A^1(x^2) = a_1 + a_3 x^2 + a_5 x^4$$

How is this useful?
We need to evaluate $A$ at points $x_1 \ldots x_n$ in time $O(n \log n)$

Idea: divide and conquer:

$$A(x) = A^0(x^2) + x A^1(x^2)$$

where $A^0$ has the even-degree terms, $A^1$ the odd

If my points are $x_1, x_2, x_{n/2}, -x_1, -x_2, -x_{n/2}$

I just need the evaluations of $A^0, A^1$ at $x_1^2, x_2^2, \ldots, x_{n/2}^2$

$$T(n) \leq 2 T(n/2) + O(n), \text{ with solution } O(n \log n). \text{ Are we done?}$$
We need to evaluate $A$ at points $x_1 \ldots x_n$ in time $O(n \log n)$

Idea: divide and conquer:

$$A(x) = A^0(x^2) + xA^1(x^2)$$

where $A^0$ has the even-degree terms, $A^1$ the odd

If my points are $x_1, x_2, x_{n/2}, -x_1, -x_2, -x_{n/2}$

I just need the evaluations of $A^0, A^1$ at $x_1^2, x_2^2, \ldots x_{n/2}^2$

$T(n) \leq 2T(n/2) + O(n)$, with solution $O(n \log n)$. Are we done?

Need points which can be iteratively decomposed in $+$ and $-$.
Complex numbers:
Real numbers "with a twist"

\[ z = x + iy \]

\[ r = \sqrt{x^2 + y^2} \]

\[ x = r \cos \theta \]
\[ y = r \sin \theta \]
\( \omega_n = n\)-th primitive root of unity

\[ \omega_n^0, \ldots, \omega_n^{n-1} \]

\( n\)-th roots of unity

We evaluate polynomial \( A \)
of degree \( n-1 \)
at roots of unity

\[ \omega_n^0, \ldots, \omega_n^{n-1} \]

Fact: The \( n \) squares of the \( n\)-th roots of unity are:

- first the \( n/2 \) \( n/2\)-th roots of unity,
- then again the \( n/2 \) \( n/2\)-th roots of unity.

\( \mathbb{R} \) from coefficient to point-value in \( O(n \log n) \) (complex) steps
Summary: Evaluate $A$ at $n$-th roots of unity $\omega_n^0, \ldots, \omega_n^{n-1}$

Divide: $A(x) = A^0(x^2) + x A^1(x^2)$
where $A^0$ has the even-degree terms, $A^1$ the odd

Conquer: Evaluate $A^0, A^1$ at $n/2$-th roots $\omega_{n/2}^0, \ldots, \omega_{n/2}^{n/2-1}$
This yields evaluation vectors $y^0, y^1$

Combine: $z := 1 = \omega_n^0$
for $(k = 0, k < n, k++)$

$$y[k] = y^0[k \text{ modulo } n/2] + z y^1[k \text{ modulo } n/2]; \quad z = z \cdot \omega_n$$

$T(n) \leq 2T(n/2) + O(n)$, with solution $O(n \log n)$. 
It only remains to go from point-value to coefficient represent.

\[
\begin{pmatrix}
y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1}
\end{pmatrix} = 
\begin{pmatrix}
1 & 1 & 1 & \ldots & 1 \\
1 & \omega_n & \omega_n^2 & \omega_n^3 & \ldots & \omega_n^{n-1} \\
1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \ldots & \omega_n^{2(n-1)} \\
1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \ldots & \omega_n^{3(n-1)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \ldots & \omega_n^{(n-1)(n-1)}
\end{pmatrix}
\begin{pmatrix}
a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1}
\end{pmatrix}
\]

\( F \)

We need to invert \( F \).
It only remains to go from point-value to coefficient representation.

\[
\begin{pmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3 \\
\vdots \\
y_{n-1}
\end{pmatrix} =
\begin{pmatrix}
1 & 1 & 1 & \cdots & 1 \\
1 & \omega_n & \omega_n^2 & \cdots & \omega_n^{n-1} \\
1 & \omega_n^2 & \omega_n^4 & \cdots & \omega_n^{2(n-1)} \\
1 & \omega_n^3 & \omega_n^6 & \cdots & \omega_n^{3(n-1)} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \cdots & \omega_n^{(n-1)(n-1)}
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
\vdots \\
a_{n-1}
\end{pmatrix}
\]

Fact: \((F^{-1})_{j,k} = \frac{\omega_n^{-jk}}{n}\) \quad \text{Note } j,k \in \{0,1,\ldots,n-1\}

To compute inverse, use FFT with \(\omega^{-1}\) instead of \(\omega\),
then divide by \(n\).