COMP251: Divide-and-Conquer
(1)

Jérôme Waldispühl
School of Computer Science
McGill University

Based on (Kleinberg & Tardos, 2005) & slides from (Snoeyink, 2004)
Divide and Conquer

• Recursive in structure
  – *Divide* the problem into sub-problems that are similar to the original but smaller in size
  – *Conquer* the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  – *Combine* the solutions to create a solution to the original problem
An Example: Merge Sort

**Sorting Problem:** Sort a sequence of $n$ elements into non-decreasing order.

- **Divide:** Divide the $n$-element sequence to be sorted into two subsequences of $n/2$ elements each.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.
Sorting problem

**Problem.** Given a list of $n$ elements from a totally-ordered universe, rearrange them in ascending order.
Sorting applications

Obvious applications.
- Organize an MP3 library.
- Display Google PageRank results.
- List RSS news items in reverse chronological order.

Some problems become easier once elements are sorted.
- Identify statistical outliers.
- Binary search in a database.
- Remove duplicates in a mailing list.

Non-obvious applications.
- Convex hull.
- Closest pair of points.
- Interval scheduling / interval partitioning.
- Minimum spanning trees (Kruskal's algorithm).
- Scheduling to minimize maximum lateness or average completion time.
- ...

...
Merge Sort – Example

Original Sequence

Sorted Sequence
Merge-Sort (A, p, r)

**INPUT:** a sequence of $n$ numbers stored in array A

**OUTPUT:** an ordered sequence of $n$ numbers

```
MergeSort (A, p, r)  // sort A[p..r] by divide & conquer
1  if p < r
2    then q ← ⌊(p+r)/2⌋
3        MergeSort (A, p, q)
4        MergeSort (A, q+1, r)
5        Merge (A, p, q, r)  // merges A[p..q] with A[q+1..r]
```

**Initial Call:** MergeSort(A, 1, n)
Procedure Merge

```
Merge(A, p, q, r)
1  n_1 ← q – p + 1
2  n_2 ← r – q
3  for i ← 1 to n_1
4      do L[i] ← A[p + i – 1]
5  for j ← 1 to n_2
6      do R[j] ← A[q + j]
7  L[n_1+1] ← ∞
8  R[n_2+1] ← ∞
9  i ← 1
10  j ← 1
11  for k ← p to r
12      do if L[i] ≤ R[j]
13          then A[k] ← L[i]
14              i ← i + 1
15          else A[k] ← R[j]
16              j ← j + 1
```

Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.


**Sentinels**, to avoid having to check if either subarray is fully copied at each step.
Merge – Example

A

... 1 6 8 9 26 32 42 43 ...

k

L

6 8 26 32 ∞

i

R

1 9 42 43 ∞

j
Correctness of Merge

\[ \text{Merge}(A, p, q, r) \]

1. \( n_1 \leftarrow q - p + 1 \)
2. \( n_2 \leftarrow r - q \)
3. \( \text{for } i \leftarrow 1 \text{ to } n_1 \) do \( L[i] \leftarrow A[p + i - 1] \)
4. \( \text{for } j \leftarrow 1 \text{ to } n_2 \) do \( R[j] \leftarrow A[q + j] \)
5. \( L[n_1 + 1] \leftarrow \infty \)
6. \( R[n_2 + 1] \leftarrow \infty \)
7. \( i \leftarrow 1 \)
8. \( j \leftarrow 1 \)
9. \( \text{for } k \leftarrow p \text{ to } r \) do if \( L[i] \leq R[j] \) then \( A[k] \leftarrow L[i] \)
10. \( i \leftarrow i + 1 \)
11. Else \( A[k] \leftarrow R[j] \)
12. \( j \leftarrow j + 1 \)

**Loop Invariant for the for loop**

At the start of each iteration of the for loop:

- Subarray \( A[p..k - 1] \) contains the \( k - p \) smallest elements of \( L \) and \( R \) in sorted order.
- \( L[i] \) and \( R[j] \) are the smallest elements of \( L \) and \( R \) that have not been copied back into \( A \).

**Initialization:**

Before the first iteration:
- \( A[p..k - 1] \) is empty.
- \( i = j = 1 \).
- \( L[1] \) and \( R[1] \) are the smallest elements of \( L \) and \( R \) not copied to \( A \).
Correctness of Merge

Maintenance:

Case 1: $L[i] \leq R[j]$

- By LI, $A$ contains $p - k$ smallest elements of $L$ and $R$ in sorted order.
- By LI, $L[i]$ and $R[j]$ are the smallest elements of $L$ and $R$ not yet copied into $A$.
- Line 13 results in $A$ containing $p - k + 1$ smallest elements (again in sorted order).
- Incrementing $i$ and $k$ reestablishes the LI for the next iteration.


Termination:

- On termination, $k = r + 1$.
- By LI, $A$ contains $r - p + 1$ smallest elements of $L$ and $R$ in sorted order.
- $L$ and $R$ together contain $r - p + 3$ elements. All but the two sentinels have been copied back into $A$. 

```
Merge(A, p, q, r)
1  n_1 ← q - p + 1
2  n_2 ← r - q
3  for i ← 1 to n_1
4    do L[i] ← A[p + i - 1]
5  for j ← 1 to n_2
6    do R[j] ← A[q + j]
7  L[n_1+1] ← ∞
8  R[n_2+1] ← ∞
9  i ← 1
10  j ← 1
11  for k ← p to r
12    do if L[i] ≤ R[j]
13      then A[k] ← L[i]
14        i ← i + 1
15      else A[k] ← R[j]
16        j ← j + 1
```
Analysis of Merge Sort

- Running time $T(n)$ of Merge Sort:
  - Divide: computing the middle takes $\Theta(1)$
  - Conquer: solving 2 subproblems takes $2T(n/2)$
  - Combine: merging $n$ elements takes $\Theta(n)$

- Total:

  \[
  T(n) = \Theta(1) \quad \text{if } n = 1
  \]
  \[
  T(n) = 2T(n/2) + \Theta(n) \quad \text{if } n > 1
  \]

  $\Rightarrow T(n) = \Theta(n \log n)$
A useful recurrence relation

**Def.** $T(n) = \text{max number of compares to mergesort a list of size } \leq n.$

**Note.** $T(n)$ is monotone nondecreasing.

**Mergesort recurrence.**

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ T([n/2]) + T([n/2]) + n & \text{otherwise} \end{cases}$$

**Solution.** $T(n)$ is $O(n \log_2 n)$.

**Assorted proofs.** We describe several ways to prove this recurrence. Initially we assume $n$ is a power of 2 and replace $\leq$ with $=.$
Proposition. If \( T(n) \) satisfies the following recurrence, then \( T(n) = n \log_2 n \).

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

Pf 1.

Assuming \( n \) is a power of 2.

\[
T(n) = n \log_2 n
\]
Proof by induction

**Proposition.** If \( T(n) \) satisfies the following recurrence, then \( T(n) = n \log_2 n \).

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

**Pf 2.** [by induction on \( n \)]

- **Base case:** when \( n = 1 \), \( T(1) = 0 \).
- **Inductive hypothesis:** assume \( T(n) = n \log_2 n \).
- **Goal:** show that \( T(2n) = 2n \log_2 (2n) \).

\[
T(2n) = 2 \ T(n) + 2n \\
= 2n \log_2 n + 2n \\
= 2n (\log_2 (2n) - 1) + 2n \\
= 2n \log_2 (2n). \quad \blacksquare
\]
Analysis of mergesort recurrence

Claim. If $T(n)$ satisfies the following recurrence, then $T(n) \leq n \lceil \log_2 n \rceil$.

\[
T(n) \leq \begin{cases} 
0 & \text{if } n = 1 \\
T([n/2]) + T([n/2]) + n & \text{otherwise}
\end{cases}
\]

Pf. [by strong induction on $n$]

- Base case: $n = 1$.
- Define $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lceil n/2 \rceil$.
- Induction step: assume true for $1, 2, \ldots, n-1$.

\[
T(n) \leq T(n_1) + T(n_2) + n \\
\leq n_1 \lceil \log_2 n_1 \rceil + n_2 \lceil \log_2 n_2 \rceil + n \\
\leq n_1 \lceil \log_2 n_2 \rceil + n_2 \lceil \log_2 n_2 \rceil + n \\
= n \lceil \log_2 n_2 \rceil + n \\
\leq n (\lceil \log_2 n \rceil - 1) + n \\
= n \lceil \log_2 n \rceil.
\]
Arithmetic operations

Given 2 (binary) numbers, we want efficient algorithms to:

• Add 2 numbers
• Multiply 2 numbers (here, we will use a divide-and-conquer method!)
Integer addition

Addition. Given two $n$-bit integers $a$ and $b$, compute $a + b$.
Subtraction. Given two $n$-bit integers $a$ and $b$, compute $a - b$.

Grade-school algorithm. $\Theta(n)$ bit operations.

Remark. Grade-school addition and subtraction algorithms are asymptotically optimal.
**Integer multiplication**

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

**Grade-school algorithm.** \( \Theta(n^2) \) bit operations.

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\times & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
\]

**Conjecture.** [Kolmogorov 1952] Grade-school algorithm is optimal.

**Theorem.** [Karatsuba 1960] Conjecture is wrong.
## Divide-and-conquer multiplication

To multiply two $n$-bit integers $x$ and $y$:

- Divide $x$ and $y$ into low- and high-order bits.
- Multiply four $\frac{n}{2}$-bit integers, recursively.
- Add and shift to obtain result.

\[
m = \lfloor n / 2 \rfloor
\]

\[
a = \lfloor x / 2^m \rfloor \quad b = x \mod 2^m
\]

\[
c = \lfloor y / 2^m \rfloor \quad d = y \mod 2^m
\]

\[
(2^m a + b) (2^m c + d) = 2^{2m} ac + 2^m (bc + ad) + bd
\]

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>

**Ex.** $x = 10001101$ $y = 11100001$

\[
\begin{align*}
a & = 10001101 \\ b & = 10101011 \\ c & = 11100001 \\ d & = 01010101
\end{align*}
\]
**Divide-and-conquer multiplication**

\[ \text{MULTIPLY}(x, y, n) \]

**IF** \((n = 1)\)

\[ \text{RETURN } x \times y. \]

**ELSE**

\[ m \leftarrow \left\lfloor \frac{n}{2} \right\rfloor. \]
\[ a \leftarrow \left\lfloor \frac{x}{2^m} \right\rfloor; \quad b \leftarrow x \mod 2^m. \]
\[ c \leftarrow \left\lfloor \frac{y}{2^m} \right\rfloor; \quad d \leftarrow y \mod 2^m. \]
\[ e \leftarrow \text{MULTIPLY}(a, c, m). \]
\[ f \leftarrow \text{MULTIPLY}(b, d, m). \]
\[ g \leftarrow \text{MULTIPLY}(b, c, m). \]
\[ h \leftarrow \text{MULTIPLY}(a, d, m). \]
\[ \text{RETURN } 2^{2m} e + 2^m (g + h) + f. \]
Divide-and-conquer multiplication analysis

**Proposition.** The divide-and-conquer multiplication algorithm requires \(\Theta(n^2)\) bit operations to multiply two \(n\)-bit integers.

**Pf.** Apply case 1 of the master theorem to the recurrence:

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

To compute middle term $bc + ad$, use identity:

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

$$m = \lfloor n / 2 \rfloor$$

$$a = \lfloor x / 2^m \rfloor \quad b = x \mod 2^m$$

$$c = \lfloor y / 2^m \rfloor \quad d = y \mod 2^m$$

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

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

Bottom line. Only three multiplication of $n/2$-bit integers.
Karatsuba multiplication

\[
\text{KARATSUBA-MULTIPLY}(x, y, n)
\]

IF \((n = 1)\)

RETURN \(x \times y\).

ELSE

\(m \leftarrow \lfloor \frac{n}{2} \rfloor\).
\(a \leftarrow \lfloor \frac{x}{2^m} \rfloor; \quad b \leftarrow x \mod 2^m.\)
\(c \leftarrow \lfloor \frac{y}{2^m} \rfloor; \quad d \leftarrow y \mod 2^m.\)
\(e \leftarrow \text{KARATSUBA-MULTIPLY}(a, c, m).\)
\(f \leftarrow \text{KARATSUBA-MULTIPLY}(b, d, m).\)
\(g \leftarrow \text{KARATSUBA-MULTIPLY}(a - b, c - d, m).\)
RETURN \(2^m e + 2^m (e + f - g) + f.\)
Karatsuba analysis

**Proposition.** Karatsuba's algorithm requires $O(n^{1.585})$ bit operations to multiply two $n$-bit integers.

**Pf.** Apply case 1 of the master theorem to the recurrence:

$$T(n) = 3T(n/2) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n^{\lg 3}) = O(n^{1.585}).$$

**Practice.** Faster than grade-school algorithm for about 320-640 bits.

Next class!
Integer arithmetic reductions

**Integer multiplication.** Given two \( n \)-bit integers, compute their product.

<table>
<thead>
<tr>
<th>problem</th>
<th>arithmetic</th>
<th>running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer multiplication</td>
<td>( a \times b )</td>
<td>( \Theta(M(n)) )</td>
</tr>
<tr>
<td>integer division</td>
<td>( a / b, \ a \mod b )</td>
<td>( \Theta(M(n)) )</td>
</tr>
<tr>
<td>integer square</td>
<td>( a^2 )</td>
<td>( \Theta(M(n)) )</td>
</tr>
<tr>
<td>integer square root</td>
<td>( \sqrt{a} )</td>
<td>( \Theta(M(n)) )</td>
</tr>
</tbody>
</table>

All integer arithmetic problems with the same complexity as integer multiplication.
History of asymptotic complexity of integer multiplication

<table>
<thead>
<tr>
<th>year</th>
<th>algorithm</th>
<th>order of growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>brute force</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td>1962</td>
<td>Karatsuba-Ofman</td>
<td>$\Theta(n^{1.585})$</td>
</tr>
<tr>
<td>1963</td>
<td>Toom-3, Toom-4</td>
<td>$\Theta(n^{1.465}), \Theta(n^{1.404})$</td>
</tr>
<tr>
<td>1966</td>
<td>Toom-Cook</td>
<td>$\Theta(n^{1+\varepsilon})$</td>
</tr>
<tr>
<td>1971</td>
<td>Schönhage–Strassen</td>
<td>$\Theta(n \log n \log \log n)$</td>
</tr>
<tr>
<td>2007</td>
<td>Fürer</td>
<td>$n \log n 2^{O(\log^*n)}$</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

number of bit operations to multiply two $n$–bit integers

used in Maple, Mathematica, gcc, cryptography, ...

Remark. GNU Multiple Precision Library uses one of five different algorithm depending on size of operands.