COMP251: Divide-and-Conquer

(1)

Jérôme Waldispühl
School of Computer Science
McGill University

Based on (Kleinberg & Tardos, 2005) and slides by K. Wayne & Snoeyink
Resources

Additional resources on the course website to prepare the final:
• Practice problems
• Exam from previous year
• No solution posted yet. Try to solve it yourself first. We will post the solutions later

Additional online reference textbook with exercises:
• Algorithms by Jeff Erikson (UIUC)
• URL: http://algorithms.wtf
## Algorithm design techniques

<table>
<thead>
<tr>
<th></th>
<th>Greedy choice</th>
<th>Optimal Substructure</th>
<th>Recursion</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Greedy</strong></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Kruskall, Huffman, Dijkstra...</td>
</tr>
<tr>
<td><strong>Dynamic Programming</strong></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>Weighted interval scheduling, Bellman-Ford...</td>
</tr>
<tr>
<td><strong>Divide-and-conquer</strong></td>
<td></td>
<td></td>
<td>✓</td>
<td>MergeSort, Karatsuba...</td>
</tr>
</tbody>
</table>
Outline

• MergeSort
  - Definition
  - Correctness
  - Complexity analysis

• Integer multiplication
  - “Naïve” recursive algorithm
  - Karatsuba

Objective: Designing a divide-and-conquer algorithm and characterizing its running time.
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.
Merge Sort - Example

Divide

Merge
Merge-Sort (A, p, r)

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

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

\[
\text{MergeSort} \ (A, p, r) \quad \text{// sort } A[p..r] \text{ by divide & conquer}
\]

1. if $p < r$
2. then $q \leftarrow \lfloor (p+r)/2 \rfloor$
3. \text{MergeSort} (A, p, q)
4. \text{MergeSort} (A, q+1, r)
5. \text{Merge} (A, p, q, r) \quad \text{// merges } A[p..q] \text{ with } A[q+1..r]

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

- **Input:** Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.
- **Output:** Merged sorted subarray in $A[p..r]$.

**Sentinels**, to avoid having to check if either subarray is fully copied at each step.
Execution of a call to Merge(A,p,q,r):

The first and second half of A[p,r] are already sorted.

Merge creates an array for each first and second halves.
Execution of a call to Merge(A,p,q,r):

Pick the lowest value in L and R and place it at the lowest index not used yet (i.e., index k)

Note: We sort in-place
Execution of a call to Merge(A,p,q,r):

Iterate the same process
Execution of a call to Merge(A,p,q,r):

```
L  6  7  12  24  ∞
  i

R  1  4  8  13  ∞
  j
```

A
```
...  1  4  6  24  1  4  8  13  ...
```

Merge (example)
Execution of a call to Merge(A, p, q, r):

$\text{L} = \begin{array}{cccccc}
6 & 7 & 12 & 24 & \infty \\
\end{array}$

$\text{R} = \begin{array}{cccccc}
1 & 4 & 8 & 13 & \infty \\
\end{array}$
Merge (example)

Execution of a call to Merge(A,p,q,r):

A

\[ \begin{array}{cccccc}
  \text{p} & \text{q} & \text{q+1} & \text{r} \\
 1 & 4 & 6 & 7 & 8 & 4 & 8 & 13 & \ldots
\end{array} \]

L

\[ \begin{array}{cccccc}
  6 & 7 & 12 & 24 & \infty
\end{array} \]

R

\[ \begin{array}{cccccc}
  1 & 4 & 8 & 13 & \infty
\end{array} \]
**Merge (example)**

Execution of a call to Merge(A,p,q,r):

![Diagram](image)

- **A**
  - ... [1, 4, 6, 7, 8, 12, 8, 13, ...]
  - p: 1, q: 4, q+1: 6, r: 13

- **L**
  - [6, 7, 12, 24, ∞]
  - i: 4

- **R**
  - [1, 4, 8, 13, ∞]
  - j: 8

L and R are sorted subarrays, and k is the index in A where the merge operation occurs.
Merge (example)

Execution of a call to Merge(A,p,q,r):

A

\[ \begin{array}{ccccccc}
    & p & q & q+1 & \cdots & r \\
\vdots & \cdots & 1 & 4 & 6 & 7 & 8 & 12 & 13 & 13 & \cdots \\
    A & \text{...} & 1 & 4 & 6 & 7 & 8 & 12 & 13 & 13 & \cdots \\
\end{array} \]

L

| 6 | 7 | 12 | 24 | \(\infty\) |

R

| 1 | 4 | 8 | 13 | \(\infty\) |
Merge (example)

Execution of a call to Merge(A,p,q,r):

We stop when A[p,r] has been filled
Correctness of Merge

**Loop Invariant property (main 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 $A$. 

**Merge($A$, $p$, $q$, $r$)**
1. $n_1 \leftarrow q - p + 1$
2. $n_2 \leftarrow r - q$
3. **for** $i \leftarrow 1$ **to** $n_1$
4. **do** $L[i] \leftarrow A[p + i - 1]$
5. **for** $j \leftarrow 1$ **to** $n_2$
6. **do** $R[j] \leftarrow A[q + j]$
7. $L[n_1+1] \leftarrow \infty$
8. $R[n_2+1] \leftarrow \infty$
9. $i \leftarrow 1$
10. $j \leftarrow 1$
11. **for** $k \leftarrow p$ **to** $r$
12. **do if** $L[i] \leq R[j]$
13. **then** $A[k] \leftarrow L[i]$
14. $i \leftarrow i + 1$
15. **else** $A[k] \leftarrow R[j]$
16. $j \leftarrow j + 1$
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.

**Case 2: Similar arguments with \( L[i] > R[j] \)**

**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 including the two sentinels. So, all elements are sorted!
Analysis of Merge Sort

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

$$T(n) = \Theta(1) \quad \text{if } n = 1$$

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

Solution: $T(n) = \Theta(n \lg n)$

We will describe two ways to prove it. Though, to make our task easier, we will assume that $n$ is a power of 2.
Divide-and-conquer recurrence: proof by recursion tree

**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.**

![Recursion Tree Diagram]

- $n = n$
- $2(n/2) = n$
- $4(n/4) = n$
- $8(n/8) = n$
- $\vdots$

\[
T(n) = n \log n
\]
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 \( T(2n) = 2n \log_2 (2n) \).

\[
T(2n) = 2 \ T(n) + 2n \\
= 2 \ n \log_2 n + 2n \\
= 2n (\log_2 (2n) - 1) + 2n \\
= 2n \log_2 (2n).
\]

This is an example where the inductive hypothesis is used to prove \( T(2n) \) instead of \( T(n+1) \).
Arithmetic operations

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

• Add 2 numbers

• Multiply 2 numbers (using divide-and-conquer!)
**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.

```
  1 1 1 1 1 1 0 1
+ 1 1 0 1 0 1 0 1
  + 0 1 1 1 1 1 0 1
  1 0 1 0 1 0 0 1 0
```

**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.

**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 $\frac{1}{2}n$-bit integers, recursively.
- Add and shift to obtain result.

$$m = \left\lfloor \frac{n}{2} \right\rfloor$$

$$a = \left\lfloor \frac{x}{2^m} \right\rfloor \quad b = x \mod 2^m$$

$$c = \left\lfloor \frac{y}{2^m} \right\rfloor \quad d = y \mod 2^m$$

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

\[\begin{array}{c}
\framebox{1} \\
\framebox{2} \\
\framebox{3} \\
\framebox{4}
\end{array}\]

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

\[\begin{array}{c}
a \\
b \\
c \\
d
\end{array}\]
**Divide-and-conquer multiplication**

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

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

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

**ELSE**

\[ m \leftarrow \lfloor n / 2 \rfloor. \]
\[ a \leftarrow \lfloor x / 2^m \rfloor; \quad b \leftarrow x \mod 2^m. \]
\[ c \leftarrow \lfloor y / 2^m \rfloor; \quad d \leftarrow y \mod 2^m. \]
\[ e \leftarrow \text{MULTIPLY}(a, c, m). \]
\[ f \leftarrow \text{MULTIPLY}(b, d, m). \]
\[ g \leftarrow \text{MULTIPLY}(b, c, m). \]
\[ h \leftarrow \text{MULTIPLY}(a, d, m). \]

\[ \text{RETURN } 2^m e + 2^m (g + h) + f. \]
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:

\[
T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n^2)
\]
Karatsuba trick

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

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

$$m = \left\lfloor \frac{n}{2} \right\rfloor$$

$$a = \left\lfloor \frac{x}{2^m} \right\rfloor \quad b = x \mod 2^m$$

$$c = \left\lfloor \frac{y}{2^m} \right\rfloor \quad d = y \mod 2^m$$

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

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

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

**KARATSUBA-MULTIPLY**(x, y, n)

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

**RETURN**  \( x \times y \).

**ELSE**

\[
m \leftarrow \lfloor n / 2 \rfloor.
\]

\[
a \leftarrow \lfloor x / 2^m \rfloor; \quad b \leftarrow x \mod 2^m.
\]

\[
c \leftarrow \lfloor y / 2^m \rfloor; \quad d \leftarrow y \mod 2^m.
\]

\[
e \leftarrow \text{KARATSUBA-MULTIPLY}(a, c, m).
\]

\[
f \leftarrow \text{KARATSUBA-MULTIPLY}(b, d, m).
\]

\[
g \leftarrow \text{KARATSUBA-MULTIPLY}(a - b, c - d, m).
\]

**RETURN**  \( 2^{2m} e + 2^m (e + f - g) + f \).
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:

$$T(n) = 3T(n/2) + \Theta(n) \implies T(n) = \Theta(n^{\log_2 3}) = O(n^{1.585}).$$

**Practice.** Faster than grade-school algorithm for about 320-640 bits.
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>$\lfloor \sqrt{a} \rfloor$</td>
<td>$\Theta(M(n))$</td>
</tr>
</tbody>
</table>

**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>$\Theta(n^{1+\epsilon})$</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>

**Remarks.**

- These results have even been improved since 2007...
- **Remark.** GNU Multiple Precision Library uses one of five different algorithm depending on size of operands.