Scheduling to minimizing lateness

Minimizing lateness problem.
- Single resource processes one job at a time.
- Job $j$ requires $t_j$ units of processing time and is due at time $d_j$.
- If $j$ starts at time $s_j$, it finishes at time $f_j = s_j + t_j$.
- Lateness: $\ell_j = \max \{ 0, f_j - d_j \}$.
- Goal: schedule all jobs to minimize maximum lateness $L = \max_j \ell_j$.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$d_j$</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$d_3 = 9$</th>
<th>$d_2 = 8$</th>
<th>$d_6 = 15$</th>
<th>$d_4 = 6$</th>
<th>$d_5 = 14$</th>
<th>$d_4 = 9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

lateness = 2  lateness = 0  max lateness = 6
Minimizing lateness: greedy algorithms

Greedy template. Schedule jobs according to some natural order.

- [Shortest processing time first] Schedule jobs in ascending order of processing time $t_j$.

- [Earliest deadline first] Schedule jobs in ascending order of deadline $d_j$.

- [Smallest slack] Schedule jobs in ascending order of slack $d_j - t_j$.

---

Minimizing lateness: greedy algorithms

Greedy template. Schedule jobs according to some natural order.

- [Shortest processing time first] Schedule jobs in ascending order of processing time $t_j$.

  ![Counterexample Table 1]

- [Smallest slack] Schedule jobs in ascending order of slack $d_j - t_j$.

  ![Counterexample Table 2]
Minimizing lateness: earliest deadline first

**Earliest-Deadline-First** \((n, t_1, t_2, ..., t_n, d_1, d_2, ..., d_n)\)

**Sort** \(n\) jobs so that \(d_1 \leq d_2 \leq ... \leq d_n\).

\(t \leftarrow 0\)

**For** \(j = 1\) to \(n\)

Assign job \(j\) to interval \([t, t + t_j]\).

\(s_j \leftarrow t; \quad f_j \leftarrow t + t_j\)

\(t \leftarrow t + t_j\)

**Return** intervals \([s_1, f_1], [s_2, f_2], ..., [s_n, f_n]\).

Max lateness = 1

<table>
<thead>
<tr>
<th>(d_1 = 6)</th>
<th>(d_2 = 8)</th>
<th>(d_3 = 9)</th>
<th>(d_4 = 9)</th>
<th>(d_5 = 14)</th>
<th>(d_6 = 15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

---

Minimizing lateness: no idle time

**Observation 1.** There exists an optimal schedule with no idle time.

**Observation 2.** The earliest-deadline-first schedule has no idle time.
Minimizing lateness: inversions

**Def.** Given a schedule $S$, an inversion is a pair of jobs $i$ and $j$ such that: $i < j$ but $j$ scheduled before $i$.

![Diagram showing an inversion between jobs $i$ and $j$.]

[as before, we assume jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$]

**Observation 3.** The earliest-deadline-first schedule has no inversions.

**Observation 4.** If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.
Minimizing lateness: inversions

**Def.** Given a schedule $S$, an inversion is a pair of jobs $i$ and $j$ such that: $i < j$ but $j$ scheduled before $i$.

**Claim.** Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

**Pf.** Let $\ell$ be the lateness before the swap, and let $\ell'$ be it afterwards.

- $\ell'_k = \ell_k$ for all $k \neq i, j$.
- $\ell'_i \leq \ell_i$.
- If job $j$ is late, $\ell'_j = f'_i - d_i$ (definition)
  - $\leq f_i - d_i$ (since $i$ and $j$ inverted)
  - $\leq \ell_j$ (definition)

Minimizing lateness: analysis of earliest-deadline-first algorithm

**Theorem.** The earliest-deadline-first schedule $S$ is optimal.

**Pf.** [by contradiction]

Define $S^*$ to be an optimal schedule that has the fewest number of inversions, and let's see what happens.

- Can assume $S^*$ has no idle time.
- If $S^*$ has no inversions, then $S = S^*$.
- If $S^*$ has an inversion, let $i$–$j$ be an adjacent inversion.
- Swapping $i$ and $j$
  - does not increase the max lateness
  - strictly decreases the number of inversions
- This contradicts definition of $S^*$.

Exercise: two schedules with no idle time & no inversions, are they actually good?
Divide-and-conquer paradigm

Divide-and-conquer.
- Divide up problem into several subproblems.
- Solve each subproblem recursively.
- Combine solutions to subproblems into overall solution.

Most common usage.
- Divide problem of size $n$ into two subproblems of size $n/2$ in linear time.
- Solve two subproblems recursively.
- Combine two solutions into overall solution in linear time.

Consequence.
- Brute force: $\Theta(n^2)$.
- Divide-and-conquer: $\Theta(n \log n)$.

\[ T(n) = \frac{1}{2} T\left(\frac{n}{2}\right) + O(n) \]
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.
- ...
Mergesort

- Recursively sort left half.
- Recursively sort right half.
- Merge two halves to make sorted whole.

input

```
ALGORITHM
```

sort left half

```
AGLOR
ITHMS
```

sort right half

```
AGLOR
HIMST
```

merge results

```
AGHILMORST
```

Merging

**Goal.** Combine two sorted lists $A$ and $B$ into a sorted whole $C$.

- Scan $A$ and $B$ from left to right.
- Compare $a_i$ and $b_j$.
- If $a_i \leq b_j$, append $a_i$ to $C$ (no larger than any remaining element in $B$).
- If $a_i > b_j$, append $b_j$ to $C$ (smaller than every remaining element in $A$).

<table>
<thead>
<tr>
<th>sorted list A</th>
<th>sorted list B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 7 10 $a_i$ 18</td>
<td>2 11 $b_j$ 17 23</td>
</tr>
</tbody>
</table>

merge to form sorted list $C$

```
2 3 7 10 11
```

$\min\{a_i, b_j\} = a_i$
A useful recurrence relation

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

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

Mergesort recurrence.

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

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 $=$.
Divide-and-conquer recurrence: proof by recursion tree

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

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

**Pf 1.**

![Recursion Tree Diagram]

Proof by induction

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

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

**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 $T(2n) = 2n \log_2 (2n)$.

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

\[ \Box \]
Analysis of mergesort recurrence

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

$$T(n) \leq \begin{cases} 
0 & \text{if } n = 1 \\
T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & \text{otherwise}
\end{cases}$$

Pf. [by strong induction on $n$]

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

\[
\begin{align*}
T(n) & \leq T(n_1) + T(n_2) + n \\
& \leq n_1 \lfloor \log_2 n_1 \rfloor + n_2 \lfloor \log_2 n_2 \rfloor + n \\
& \leq n_1 \lfloor \log_2 n_2 \rfloor + n_2 \lfloor \log_2 n_2 \rfloor + n \\
& = n_1 \lfloor \log_2 n_2 \rfloor + n \quad \text{by induction} \\
& \leq n_1 \lfloor \log_2 n_2 \rfloor - 1 + n \\
& = n \lfloor \log_2 n_2 \rfloor.
\end{align*}
\]
Counting inversions

Music site tries to match your song preferences with others.
• You rank \( n \) songs.
• Music site consults database to find people with similar tastes.

**Similarity metric:** number of inversions between two rankings.
• My rank: \( 1, 2, ..., n \).
• Your rank: \( a_1, a_2, ..., a_n \).
• Songs \( i \) and \( j \) are inverted if \( i < j \), but \( a_i > a_j \).

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>me</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>you</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

2 inversions: 3–2, 4–2

**Brute force:** check all \( \Theta(n^2) \) pairs.

Counting inversions: applications

• Voting theory.
• Collaborative filtering.
• Measuring the "sortedness" of an array.
• Sensitivity analysis of Google's ranking function.
• Rank aggregation for meta-searching on the Web.
• Nonparametric statistics (e.g., Kendall's tau distance).

---

**Rank Aggregation Methods for the Web**

Cynthia Dwork\textsuperscript{1}  Ravi Kumar\textsuperscript{1}  Moni Naor\textsuperscript{1}  D. Sivakumar\textsuperscript{1}

**ABSTRACT**

We consider the problem of combining ranking results from various sources. In the context of the Web, the main applications include building news-watch engines, combining ranking functions, selecting documents based on multiple criteria, and improving search precision through word associations. We develop a set of techniques for the rank-aggregation problem and compare their performance to that of well-known methods. A primary goal of our work is to design rank-aggregation techniques that are effectively robust to "spam," a serious problem in Web search. Experiments show that our methods are simple, efficient, and effective.

Keywords: rank aggregation, ranking functions, meta-search, multi-weather systems, spam
Counting inversions: divide-and-conquer

- Divide: separate list into two halves \( A \) and \( B \).
- Conquer: recursively count inversions in each list.
- Combine: count inversions \((a, b)\) with \(a \in A\) and \(b \in B\).
- Return sum of three counts.

**Input**

| 1 | 5 | 4 | 8 | 10 | 2 | 6 | 9 | 3 | 7 |

**Count inversions in left half A**

| 1 | 5 | 4 | 8 | 10 |

- 5-4

**Count inversions in right half B**

| 2 | 6 | 9 | 3 | 7 |

- 6-3 9-3 9-7

**Count inversions \((a, b)\) with \(a \in A\) and \(b \in B\)**

| 1 | 5 | 4 | 8 | 10 |

- 4-2 4-3 5-2 5-3 8-2 8-3 8-6 8-7 10-2 10-3 10-6 10-7 10-9

**Output** \(1 + 3 + 13 = 17\)
Counting inversions: how to combine two subproblems?

Q. How to count inversions \((a, b)\) with \(a \in A\) and \(b \in B\)?
A. Easy if \(A\) and \(B\) are sorted!

Warmup algorithm.
- Sort \(A\) and \(B\).
- For each element \(b \in B\),
  - binary search in \(A\) to find how elements in \(A\) are greater than \(b\).

```
list A
7   10  18  3   14
sort A
3   7   10  14  18

list B
17  23  2   11  16
sort B
2   11  16  17  23
```

binary search to count inversions \((a, b)\) with \(a \in A\) and \(b \in B\)

```
3   7   10  14  18

2   11  16  17  23
5   2   1   1   0
```

Counting inversions: how to combine two subproblems?

Count inversions \((a, b)\) with \(a \in A\) and \(b \in B\), assuming \(A\) and \(B\) are sorted.
- Scan \(A\) and \(B\) from left to right.
- Compare \(a_i\) and \(b_j\).
  - If \(a_i < b_j\), then \(a_i\) is not inverted with any element left in \(B\).
  - If \(a_i > b_j\), then \(b_j\) is inverted with every element left in \(A\).
- Append smaller element to sorted list \(C\).

```
count inversions \((a, b)\) with \(a \in A\) and \(b \in B\)
3   7   10  \(a_i\)  18
2   11  \(b_j\)  17  23
5   2

merge to form sorted list \(C\)
2   3   7   10  11
```

17
Counting inversions: divide-and-conquer algorithm implementation

**Input.** List \( L \).

**Output.** Number of inversions in \( L \) and sorted list of elements \( L' \).

\[
\text{SORT-AND-COUNT}(L)
\]

- **If** list \( L \) has one element
  - **RETURN** \((0, L)\).

- **DIVIDE** the list into two halves \( A \) and \( B \).
  - \((r_A, A) \leftarrow \text{SORT-AND-COUNT}(A)\).
  - \((r_B, B) \leftarrow \text{SORT-AND-COUNT}(B)\).
  - \((r_{AB}, L') \leftarrow \text{MERGE-AND-COUNT}(A, B)\).

- **RETURN** \((r_A + r_B + r_{AB}, L')\).

\text{time} \ O(1A1+1B1)

Counting inversions: divide-and-conquer algorithm analysis

**Proposition.** The sort-and-count algorithm counts the number of inversions in a permutation of size \( n \) in \( O(n \log n) \) time.

**Pf.** The worst-case running time \( T(n) \) satisfies the recurrence:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & \text{otherwise}
\end{cases}
\]

\[
T(n) = 2 \cdot T \left( \frac{n}{2} \right) + \Theta(n)
\]

\[
T(1n) = \Theta(n \cdot \log n)
\]
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.

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

**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.

![Multiplication Diagram]

**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.

\[
m = \lceil \frac{n}{2} \rceil \\
a = \lfloor x / 2^m \rfloor \quad b = x \mod 2^m \\
c = \lfloor y / 2^m \rfloor \quad d = y \mod 2^m
\]

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

\[
\text{Ex. } x = 10001101 \quad y = 11100001
\]

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c} \\
\text{d}
\end{array}
\]

\[
\begin{array}{c}
\text{x} \\
\text{y}
\end{array}
\]
**Divide-and-conquer multiplication**

\textbf{MULTIPLY}(x, y, n)

\textbf{IF} \ (n = 1)
  \textbf{RETURN} \ x \times y.

\textbf{ELSE}
  \hspace{1em} m \leftarrow \lfloor \frac{n}{2} \rfloor.
  \hspace{1em} a \leftarrow \lfloor \frac{x}{2^m} \rfloor; \ b \leftarrow x \mod 2^m.
  \hspace{1em} c \leftarrow \lfloor \frac{y}{2^m} \rfloor; \ d \leftarrow y \mod 2^m.
  \hspace{1em} e \leftarrow \text{MULTIPLY}(a, c, m).
  \hspace{1em} f \leftarrow \text{MULTIPLY}(b, d, m).
  \hspace{1em} g \leftarrow \text{MULTIPLY}(b, c, m).
  \hspace{1em} h \leftarrow \text{MULTIPLY}(a, d, m).
  \hspace{1em} \text{RETURN} \ 2^{2m}e + 2^m (g + h) + f.

**Divide-and-conquer multiplication analysis**

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

\textbf{Pf.} Apply case 1 of the master theorem to the recurrence:

\[ T(n) = 4T(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(n^2) \]
Karatsuba trick

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

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

$$m = \lfloor n / 2 \rfloor$$
$$a = \lfloor x / 2^m \rfloor \quad b = x \mod 2^m$$
$$c = \lfloor y / 2^m \rfloor \quad d = y \mod 2^m$$

Middle term

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

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

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

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

\textbf{IF} \ (n = 1)
\quad \text{RETURN} \ x \times y.
\textbf{ELSE}
\quad m \leftarrow \lfloor n / 2 \rfloor.
\quad a \leftarrow \lfloor x / 2^m \rfloor; \quad b \leftarrow \mod{x, 2^m}.
\quad c \leftarrow \lfloor y / 2^m \rfloor; \quad d \leftarrow \mod{y, 2^m}.
\quad e \leftarrow \text{KARATSUBA-MULTIPLY}(a, c, m).
\quad f \leftarrow \text{KARATSUBA-MULTIPLY}(b, d, m).
\quad g \leftarrow \text{KARATSUBA-MULTIPLY}(a - b, c - d, m).
\quad \text{RETURN} \ 2^{2m} e + 2^{m} (e + f - g) + f.

Combining in time $\Theta(n)$

Karatsuba analysis

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

\textbf{Pf.} Apply case 1 of the master theorem to the recurrence:

\[ T(n) = 3 T(n/2) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n^{\log_2 3}) = O(n^{1.585}). \]

\textbf{Practice.} Faster than grade-school algorithm for about 320-640 bits.
\[ T(n) = 3 \cdot T \left( \frac{n}{2} \right) + \eta \]

Total time:
\[ n + \frac{3}{2} n + \left( \frac{3}{2} \right) n + \ldots \]

\[ = n \cdot \left( 1 + \frac{3}{2} + \left( \frac{3}{2} \right)^2 + \ldots + \left( \frac{3}{2} \right)^t \right) \]

\[ \sum_{i=0}^{t} \left( \frac{3}{2} \right)^i = \frac{1 - q^t}{1 - q} \]
\[ q = \frac{3}{2} \]

\[ t = \log_2 \left( \frac{n}{1 - \left( \frac{3}{2} \right)^{\frac{3}{2}}} \right) \]

\[ \text{time} = n \cdot \frac{\log_2 n + 1}{1 - \frac{3}{2}} \]

\[ -n \cdot \frac{\left( \frac{3}{2} \right)^{\frac{3}{2}} \log_2 n}{\log_2 n} \]

\[ \approx \frac{n}{2} \cdot \left( \frac{3}{2} \right)^{\frac{3}{2}} \log_2 n \]

\[-\frac{3}{2} \cdot \frac{3}{n} = \frac{3}{2} \cdot \left( \log_2 \frac{3}{2} \right) \log_2 n \]

\[ \left( \log_2 \frac{3}{2} \right)^{\log_2 3} \leq \frac{\log_2 3}{n} \]
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>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^{1.408}) )</td>
</tr>
<tr>
<td>1966</td>
<td>Toom-Cook</td>
<td>( \Theta(n^{1.8}) )</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>
</tbody>
</table>

number of bit operations to multiply two \( n \)-bit integers

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