Schönhage–Strassen algorithm

\[ T(n) \in O(n \log n \log \log n) \]

Only used for really big numbers: \( a > 2^{2^{12}} \)

State of the Art Integer Multiplication (Theory): \( O(n \log n \log \log n) \) for increasing small \( g(n) \equiv \log n \)

Integer Division:
- Input: \( x, y \) (positive n-bit integers)
- Output: positive integers \( q \) (quotient) and remainder \( r \) s.t. \( x = qy + r \) and \( 0 < r < y \)
- Algorithm to compute quotient \( q \) and remainder \( r \) requires \( O(\log n) \) multiplications using Newton’s method (approximates roots of a real-valued polynomial).

Announcement: Homework 3 due February 15th at 11:59PM

Begun, the decimal wars have. \([\text{Pan, Bini et al, Schönhage, ...}]\]

December, 1979.

Integer Multiplication in time \( \Theta(n^{2.805}) \)

Key Trick:
- Divide each n-bit number into two n/2-bit numbers
- \( T(n) = O(n^{2.521801}) \) for large enough \( k \)

Caveat: Hidden constants increase with \( k \)

Fast Matrix Multiplication: Theory

Q. Multiply two 2-by-2 matrices with 1 scalar multiplications?
A. Yes! \([\text{Strassen 1969}]\)

Q. Multiply two 2-by-2 matrices with 4 scalar multiplications?
A. Impossible. \([\text{Hopcroft and Kerr 1971}]\)

Q. Two 3-by-3 matrices with 21 scalar multiplications?
A. Also impossible. \([\text{Pan, Bini et al, Schönhage, ...}]\)

Begin, the decimal wars have. \([\text{Pan, Bini et al, Schönhage, ...}]\)
- Two 20-by-20 matrices with 4,440 scalar multiplications.
- Two 46-by-46 matrices with 47,217 scalar multiplications.
- A year later.
Fast Matrix Multiplication: Theory

Best known. $O(n^{2.376})$ [Coppersmith-Winograd, 1987]

Conjecture. $O(n^{1+\epsilon})$ for any $\epsilon > 0$.

Caveat. Theoretical improvements to Strassen are progressively less practical.

Best known. $O(n^{2.373})$ [Williams, 2014]

Conjecture. $O(n^{2+\epsilon})$ for any $\epsilon > 0$.

Caveat. Theoretical improvements to Strassen are progressively less practical.

Best known. $O(n^{2.3729})$ [Le Gall, 2014]

Conjecture. $O(n^{2+\epsilon})$ for any $\epsilon > 0$.

Caveat. Theoretical improvements to Strassen are progressively less practical.

Chapter 6
Dynamic Programming

Greedy. Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer. Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Algorithmic Paradigms

Greedy: Build up a solution incrementally, myopically optimizing some local criterion.

Divide-and-conquer: Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

Dynamic programming: Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Dynamic Programming History

Bellman, [1950s] Pioneered the systematic study of dynamic programming.

Etymology:
- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.

"It's impossible to use dynamic in a pejorative sense" "something not even a Congressman could object to"

Dynamic Programming Applications

Areas:
- Bioinformatics
- Control theory
- Information theory
- Operations research
- Computer science: theory, graphics, AI, compilers, systems, ...

Some famous dynamic programming algorithms:
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context-free grammars.

Computing Fibonacci numbers

On the board.

6.1 Weighted Interval Scheduling

Weighted interval scheduling problem:
- Job $j$ starts at $s_j$ and finishes at $f_j$ with weight or value $v_j$.
- Two jobs compatible if they don’t overlap.
- Goal: find maximum weight subset of mutually compatible jobs.

Unweighted Interval Scheduling (will cover in Greedy paradigms)

Previously showed: Greedy algorithm works if all weights are 1.
Solution: Sort requests by finish time (ascending order)

Observation: Greedy algorithm can fail spectacularly if arbitrary weights are allowed.

Unweighted Interval Scheduling (will cover in Greedy paradigms)

Notation: Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.
Def: $p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$.

En: $p(8) = 5, p(7) = 3, p(2) = 0$. 

Weighted Interval Scheduling

Weighted Internal scheduling problem:
- Job $j$ starts at $s_j$ and finishes at $f_j$ with weight or value $v_j$.
- Two jobs compatible if they don’t overlap.
- Goal: Find maximum weight subset of mutually compatible jobs.

Computing Fibonacci numbers

On the board.
Dynamic Programming: Binary Choice

Notation. \( \text{OPT}(j) \) = value of optimal solution to the problem consisting of job requests 1, 2, ..., \( j \).

- Case 1: \( \text{OPT} \) selects job \( j \).
  - collect profit \( v_j \)
  - can’t use incompatible jobs \( \{ p(j) + 1, p(j) + 2, \ldots, j - 1 \} \)
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., \( p(j) \)

- Case 2: \( \text{OPT} \) does not select job \( j \).
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., \( j - 1 \)

\[
\text{OPT}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \{ v_j + \text{OPT}(p(j)), \text{OPT}(j-1) \} & \text{otherwise}
\end{cases}
\]

Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems \( \Rightarrow \) exponential algorithms.

Ex. Number of recursive calls for family of “layered” instances grows like Fibonacci sequence \( F_n > 1.6^n \).

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

Key Insight: Do we really need to repeat this computation?

Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

Claim. Memoized version of algorithm takes \( O(n \log n) \) time.

- Sort by finish time: \( O(n \log n) \).
- Computing \( p(\cdot) \) \( : O(n \log n) \) via sorting by start time.
- \( \text{M-Compute-Opt}(\cdot) \) each invocation takes \( O(1) \) time and either
  - (i) returns an existing value \( M[\cdot] \)
  - (ii) fills in one new entry \( M[\cdot] \) and makes two recursive calls
- Progress measure \( 0 = \# \) nonempty entries of \( M[\cdot] \),
  - initially \( 0 < 0 \), throughout \( 0 < \# \)
  - (ii) increases \( 0 \)-by-1 \( \Rightarrow \) at most \( 2n \) recursive calls.
- Overall running time of \( \text{M-Compute-Opt}(\cdot) \) is \( O(n) \).

Remark. \( O(n) \) if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?

A. Do some post-processing.

Run \( \text{M-Compute-Opt}(n) \)

\[
\text{Find-Solution}(n) \{
\text{if } (j = 0) \{ \\
\text{output nothing} \\
\text{else if } (v_j + M[p(j)] > M[j-1]) \\
\text{print } j \\
\text{Find-Solution}(p(j)) \\
\text{else} \\
\text{Find-Solution}(j-1) \\
\}\}
\]

\# of recursive calls \( \leq n \Rightarrow O(n) \).
Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming: Unwind recursion.

Input: \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

Sort jobs by finish times so that \( f_1 \leq f_2 \leq \cdots \leq f_n \).

Compute \( p[1], p[2], \ldots, p[n] \)

Iterative-Compute-Opt

\[
M[0] = 0 \\
\text{for } j = 1 \text{ to } n \\
\quad M[j] = \max(v_j + M[p(j)], M[j-1])
\]

6.3 Segmented Least Squares

Least squares:
- Foundational problem in statistics and numerical analysis.
- Given \( n \) points in the plane: \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\)
- Find a line \( y = ax + b \) that minimizes the sum of the squared error:

\[
\text{SSE} = \sum_{i=1}^{n} (y_i - ax_i - b)^2
\]

Solution: Calculus \( \Rightarrow \) min error is achieved when

\[
a = \frac{\sum_{i=1}^{n} x_i y_i - (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n} y_i)}{\sum_{i=1}^{n} x_i^2 - (\sum_{i=1}^{n} x_i)^2} \quad \text{and} \quad b = \frac{\sum_{i=1}^{n} y_i - ax_i}{n}
\]

Segmented Least Squares

Points lie roughly on a sequence of several line segments.
- Given \( n \) points in the plane: \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\)
- Find a sequence of lines that minimizes:
  - the sum of the sums of the squared errors \( E \) in each segment
  - the number of lines \( L \)
  - Tradeoff function: \( E + cL \) for some constant \( c > 0 \).

\[
\text{Optimal choice for } f(x) \text{ to balance accuracy and parsimony:}
\]

\[
\frac{\text{number of lines}}{\text{goodness of fit}}
\]

Dynamic Programming: Multiway Choice

Notation:
- \( \text{OPT}(j) = \text{minimum cost for points } p_1, p_2, \ldots, p_j \)
- \( e[i, j] = \text{minimum sum of squares for points } p_i, p_{i+1}, \ldots, p_j \)

To compute \( \text{OPT}(j) \):
- Last segment uses points \( p_i, p_{i+1}, \ldots, p_j \) for some \( i \).
- Cost = \( e[i, j] + c \times \text{OPT}(i-1) \)

\[
\text{OPT}(j) = \begin{cases} 
0 & \text{if } 1 \leq j < i \\
\min_{1 \leq k \leq i} \left( e[i, j] + c \times \text{OPT}(i-1) \right) & \text{otherwise}
\end{cases}
\]
Segmented Least Squares: Algorithm

**Input:** \( p_0, \ldots, p_n, a \)

**Segmented-Least-Squares()**
- \( M[0] = 0 \)
- for \( j = 1 \) to \( n \)
  - for \( i = 1 \) to \( j \)
    - compute the least square error \( e_{ij} \) for the segment \( p_i, \ldots, p_j \)
  - for \( j = 1 \) to \( n \)
    - \( M[j] = \min_{1 \leq i \leq j} (e_{ij} + c + M[i-1]) \)
- return \( M[n] \)

**Running time:** \( O(n^3) \)
- Bottleneck: computing \( e_{ij} \) for \( O(n^2) \) pairs, \( O(n) \) per pair using previous formula.

**Knapsack Problem**

Given \( n \) objects and a "knapsack."
- Item \( i \) weighs \( w_i \) kilograms and has value \( v_i > 0 \).
- Knapsack has capacity of \( W \) kilograms.
- Goal: fill knapsack as to maximize total value.

**Ex:** \( \{ 3, 4 \} \) has value 40.

**Greedy:** repeatedly add item with maximum ratio \( v_i / w_i \).
- Ex: \( \{ 5, 2, 1 \} \) achieves only value = 35 \( \Rightarrow \) greedy not optimal.

**Def.** \( \text{OPT}(i) = \max \text{ profit subset of items } 1, \ldots, i \).
- Case 1: \( \text{OPT} \) does not select item \( i \).
  - \( \text{OPT} \) selects best of \( \{ 1, 2, \ldots, i-1 \} \)
- Case 2: \( \text{OPT} \) selects item \( i \).
  - \( \text{OPT} \) selects best of \( \{ 1, 2, \ldots, i-1 \} \) using this new weight limit

**Dynamic Programming: False Start**

**Def.** \( \text{OPT}(i, w) = \max \text{ profit subset of items } 1, \ldots, i \) with weight limit \( w \).
- Case 1: \( \text{OPT} \) does not select item \( i \).
  - \( \text{OPT} \) selects best of \( \{ 1, 2, \ldots, i-1 \} \) using weight limit \( w \)
- Case 2: \( \text{OPT} \) selects item \( i \).
  - new weight limit \( = w - w_i \)
  - \( \text{OPT} \) selects best of \( \{ 1, 2, \ldots, i-1 \} \) using this new weight limit

**Knapsack Problem: Bottom-Up**

**Def.** \( \text{OPT}(i, w) = \max \text{ profit subset of items } 1, \ldots, i \) with weight limit \( w \).
- Case 1: \( \text{OPT} \) does not select item \( i \).
  - \( \text{OPT} \) selects best of \( \{ 1, 2, \ldots, i-1 \} \) using weight limit \( w \)
- Case 2: \( \text{OPT} \) selects item \( i \).
  - new weight limit \( = w - w_i \)
  - \( \text{OPT} \) selects best of \( \{ 1, 2, \ldots, i-1 \} \) using this new weight limit

**Input:** \( n, W, w_1, \ldots, w_n, v_1, \ldots, v_n \)

for \( w = 0 \) to \( W \)
  - \( M[0, w] = 0 \)
for \( i = 1 \) to \( n \)
  - for \( w = 1 \) to \( W \)
    - if \( w_i = w \)
      - \( M[i, w] = M[i-1, w] \)
    - else
      - \( M[i, w] = \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \} \)
return \( M[n, W] \)
### Knapsack Algorithm

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

**W = 11**

OPT: {4, 5}  
value = 22 + 18 = 40

### Knapsack Problem: Running Time

*Running time: \( \Theta(nW) \)*

- Not polynomial in input size!
- Only need \( \log W \) bits to encode each weight
- Problem can be encoded with \( \Theta(n \log W) \) bits
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete.  

Knapsack approximation algorithm: There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum.  

[Section 11.8]