CSE 589
Applied Algorithms
Spring 1999

Subset Sum
Algorithm Evaluation
Mergesort

An Easy NP-Complete Problem

• Subset Sum
  – Input: Integers $a_1, a_2, ..., a_n, b$
  – Output: Determine if there is subset $X \subseteq \{1, 2, ..., n\}$ with the property $\sum_{i \in X} a_i = b$

• Algorithm:
  – Let $A[0..b]$ be a Boolean array of size $b + 1$ initialized as follows $A[0] = 1$ and $A[i] = 0$ for $1 < i < b$.
  – After scanning the input $a_1, a_2, ..., a_n$ maintaining the invariant that $A[i] = 1$ if and only if some subset of $a_1, a_2, ..., a_n$ adds up to $i$.

Example of the Algorithm

<table>
<thead>
<tr>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$a_3$</th>
<th>$a_4$</th>
<th>$a_5$</th>
<th>$a_6$</th>
<th>$a_7$</th>
<th>$a_8$</th>
<th>$a_9$</th>
<th>$a_{10}$</th>
<th>$a_{11}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>2</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Polynomial or Exponential?

• $O((b+1)n)$
• $b$ is represented in binary
  – $a_1, a_2, ..., a_n, b \leq 2^b$ where problem size $s \leq (n+1)k$
  – array $A[0..b]$ has size at most $2^b + 1 = 2^{b(n+1)} + 1$.
• $b$ is represented in unary
  – $a_1, a_2, ..., a_n, b \leq 2^b$ where problem size $s \leq kn + b$
  – array $A[0..b]$ has size $b+1 \leq s$.

Strong NP-Completeness

• A decision problem is strong NP-complete if it remains NP-complete even if the numerical inputs are presented in unary.
• Subset Sum and similar problems are polynomial time solvable if the problem is presented in unary.
• 3-Partition and Bin Packing are strong NP-complete.
Some “Hard” Problems are Easy

- Example: Given a set of fields of a structure of length $f_1, f_2, \ldots, f_n$ in bytes. Can they be fit into two cache lines of length $b$ bytes each.
- Critical observation: $b$ is small, often 32 or 64.
- Algorithm: Use the subset sum algorithm to find the largest $c \leq b$ such that some subset of the fields fits exactly into $c$ bytes. You will need the method of reporting a solution from the decision problem to report a subset that adds up to $c$. The remaining field lengths must sum to be $\leq b$.

Evaluating Algorithms - Correctness

- Correctness or quality of the answer
  - Does it give the right answer.
  - Does it give an answer that is close to the right answer (for an approximate algorithm).
  - This can be extremely difficult to determine.
  - Does it give a good answer on real data or on what I foresee as real data.
    - Must implement and test on real data.
    - Use of benchmarks
      - Good because common to all.
      - Bad because algorithms can be tuned to a benchmark.

Examples of Quality Criteria

- Lossless Data Compression
  - compression ratio
- VLSI Layout
  - area used
- Compiler Optimization
  - percentage reduction in execution time
- Encryption
  - Security of the method from attacks
- Traveling Salesman’s Tour
  - closeness to optimal

Theoretical Analysis

- Time complexity
  - worst case
  - average case
  - amortized time complexity
- Storage complexity
  - worst case
  - average case
- Important operation counts
- Memory performance
  - cache misses or page faults

Empirical Evaluation

- Must implement to test
- Data
  - real data set
  - synthetic - generated by a program
- Profiling
  - wall clock execution time
  - performance monitoring using processor counters
  - instrument program with internal counters
  - binary instrumentation tools - Atom, Etch, ...

Atom

- Alan Eustace and Amitabh Srivastava (1994)
- Examples of use
  - Simulate a cache with specific parameters (size, block size, associativity). Output total memory accesses and cache misses.
  - Generate a histogram of heap data sizes allocated
  - Simulate a branch prediction scheme. Output successes.
- How done
  - Atom inserts code into a binary to do specific tasks.
Atom Flow

Executable binary

Instrumentation code

Atom

Instrumented executable

Analysis code

Analysis data

Sorting

- Output: A[1..n] in sorted order, that is for 1 ≤ i < n, A[i] < A[i+1].

Classic Mergesort

- Two sorted arrays can be merged into one sorted array very quickly in time O(n + m) where n and m are the sizes of the arrays.

Merging (1)

Merging (2)

Merging (3)
Recursive Mergesort

A[1..n] is to be sorted;
B[1..n] is an auxiliary array;
Mergesort(i,j) {sorts the subarray A[i..j]}
if i < j then
k := (i+j)/2;
Mergesort(i,k);
Mergesort(k+1,j);
Merge A[i..k] with A[k+1..j] into B[i..j];
Copy B[i..j] into A[i..j];

Mergesort Analysis

• Storage complexity is 2n plus O(log n) for the call stack.
  – This is not an “in-place” sorting algorithm.
• Time complexity is O(n log n).
  – Recurrence describes the running time

\[
T(0), T(1) \leq a \\
T(n) \leq 2T(n/2) + bn \\
\]

Solving the Recurrence

\[
T(0), T(1) \leq a \\
T(n) \leq 2T(n/2) + bn \\
\leq 2(2T(n/4) + bn/2) + bn \\
= 4T(n/4) + 2bn \\
\vdots \\
\leq 2^k T(n/2^k) + kbn \\
\leq nT(1) + bn \log_2 n \\
\leq an + bn \log_2 n \\
= O(n \log n)
\]

Merging Pattern of Recursive Mergesort