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 $a_1, a_2, \ldots, a_k$ maintaining the invariant that A[i] = 1 if and only if some subset of $a_1, a_2, \ldots, a_k$ adds up to i.

Subset Sum Algorithm

```
for k = 1 to n do
   for i = b to 0
      if A[i] = 1 and a_k + i ≤ b then
         A[a_k + i] := 1
if A[b] = 1 then some subset adds up to b
```

Time Complexity is O((b+1)n)

Polynomial time?

Example of the Algorithm

3, 5, 2, 7, 4, 2, b = 11

	0	1	2	3	4	5	6	7	8	9	10	11
	1	0	0	0	0	0	0	0	0	0	0	0
3	1	0	0	1	0	0	0	0	0	0	0	0
5	1	0	0	1	0	1	0	0	1	0	0	0
2	1	0	1	1	0	1	0	1	1	0	1	0
7	1	0	1	1	0	1	0	1	1	1	1	0
4	1	0	1	1	1	1	1	1	1	1	1	1
2	1	0	1	1	1	1	1	1	1	1	1	1

Polynomial or Exponential?

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

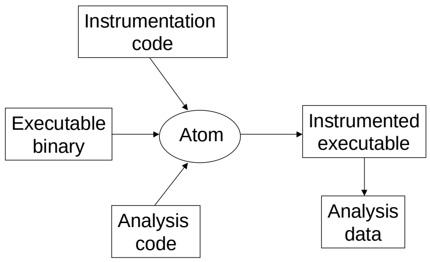
Empirical Evaluation

- 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

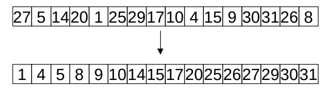
- Alan Eustace and Amitabh Srivastava (1994)
- Examples of use
  - Simulate a cache with specific parameters (size, block size, associativity). Output total memory accesses and cache misses.
  - Generate a histogram of heap data sizes allocated
  - Simulate a branch prediction scheme. Output successes.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

Atom Flow



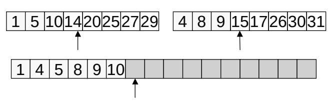
Sorting

- Input: Array A[1..n] of keys.
- Output: A[1..n] in sorted order, that is for $1 \leq i < n$, $A[i] < A[i+1]$.

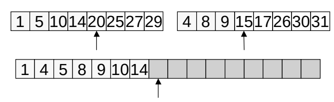


Classic Mergesort

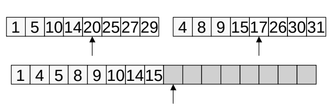
- Two sorted arrays can be merged into one sorted array very quickly in time O(n + m) where n and m are the sizes of the arrays.



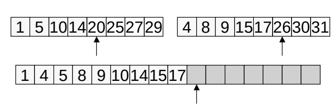
Merging (1)



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

$$T(0), T(1) \le a$$
$$T(n) \le 2T(n/2) + bn$$

2 recursive calls ($2T(n/2)$)

Time to merge and copy. (bn)

Solving the Recurrence

$$T(0), T(1) \le a$$
$$T(n) \le 2T(n/2) + bn$$
$$\le 2(2T(n/4) + bn/2) + bn \quad \text{substitution}$$
$$= 4T(n/4) + 2bn \quad \text{algebra}$$
$$\vdots$$
$$\le 2^k T(n/2^k) + kbn \quad \text{generalization}$$
$$\le nT(1) + bn\log_2 n \quad n = 2^k$$
$$\le an + bn\log_2 n$$
$$= O(n\log n)$$

Merging Pattern of Recursive Mergesort