Introduction to Algorithms

Alexandre David

Outline

- Notion of algorithms, GCD example.
- Algorithmic problem solving.
- Problem types.
- Sorting example.
- Numerical example.
- Analyzing algorithms.

Notion of Algorithms

- Why study algorithms?
- What is an algorithm?
- Example: GCD
  - Known example
  - Nonambiguity requirement
  - Define range of inputs
  - Different representations of the algorithm
  - Several algorithm for the same problem
  - Different ideas, different running speeds

GCD: The Problem

- Greatest common divisor of **2 nonnegative, not both zero integers**, denoted gcd(m,n), defined as the largest integer that **divides both m and n with a remainder of zero**.
- Algorithms:
  - Consecutive integer checking
  - Euclid's algorithm
  - Prime decomposition

Consecutive Integer Checking

- Idea: solution cannot be greater than min(m,n). Let *t=min{m,n}.* Check *t* and try again by decreasing *t*.
- Correctness: greatest? Termination?
- Efficiency: running time?

Step 1: assign *min{m,n}* to *t.*

Step 2: divide *m* by *t*. If remainder == 0 then step 3, otherwise step 4.

Step 3: divide *n* by *t*. If remainder == 0 return *t*, otherwise step 4.

Step 4: decrease *t* by 1, go to step 2.

- Idea: apply repeadly *gcd(m,n) = gcd(n, m mod n)* until *m mod n* is equal to zero (stop when reach *gcd(m,0)=m*). Ex: gcd(60,24)=gcd(24,12)=gcd(12,0)=12.

```
Algorithm Euclid(m,n)
// Computes gcd(m,n) by Euclid's algorithm
// Input: two nonnegative, non both zero integers m and n
// Output: GCD of m and n
while n != 0 do
    r := m mod n
    m := n
    n := r
return m
```

Prime Decomposition

- Idea: decomposition into primes and pick the common factors.

Step 1: find the prime factors of m.

Step 2: find the prime factors of n.

Step 3: identify all the common factors (if *p* is a common factor occuring p_m and p_n times in *m* and *n*, respectively, it should be repeated $min\{p_m, p_n\}$ times).

Step 4: Compute the product of all the common factors and return it as the result.

- Problem: Step 1&2 are sub-problems to be solved.

Algorithmic Problem Solving

- Understand the problem
- Choose exact/approximate problem solving
- Decide on appropriate data structures
- Apply an algorithm design technique
- Specify the algorithm
- Prove the correctness of the algorithm
- Analyze the algorithm – time and space efficiency – simplicity – generality
- Code the algorithm

Problem Types

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

Sorting Example

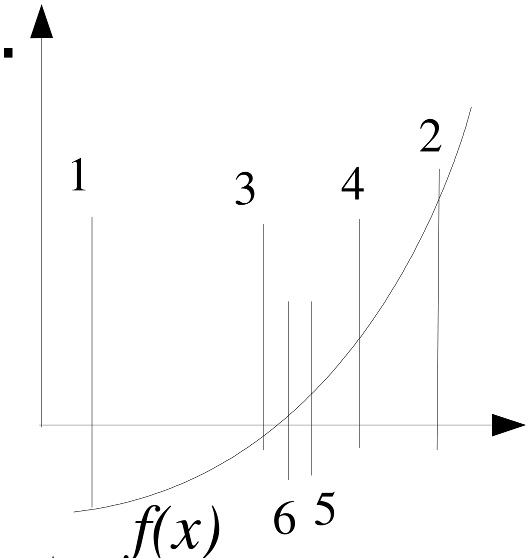
- The sorting problem:

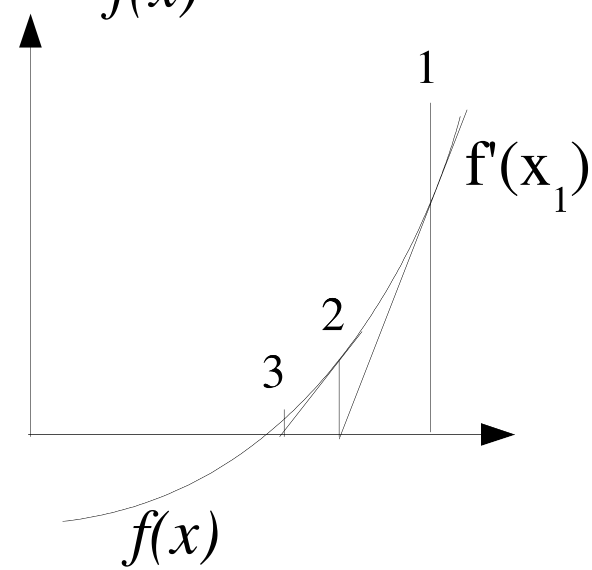
  Input: A sequence of *n* numbers <a1,a2,...,an>

  Output: A permutation (reordering) $<a_1',a_2',...,a_n'>$ of the input sequence such that $a_1'\leq a_2'\leq...\leq a_n'$.
- Algorithms to solve it: insertion sort, merge sort, quicksort. Insertion sort takes c_1n^2 in time, merge sort takes $c_2n\lg(n)$. Let's sort 10^6 elements.
  - Good insertion code $2n^2$: $2(10^6)^2/10^9=2000s$
  - Average merge sort $50n\lg(n)$: $50*10^6\lg(10^6)/10^7=100s$ on another CPU 100x slower.

Numerical Example

- Find *x* s.t. $f(x)=0$ for a continuous monotonic function.
  - Bisection algorithm:
    iterate on $[x,y]^0,[x,y]^1$.. s.t. $f(x)<0$ and $f(y)>0$ (or opposite), reduce interval by 2 everytime.
  - Newton-Raphson algorithm:
    use the derivative $x_{i+1}=x_i-f(x_i)/f'(x_i)$
    converge much faster.
- Numerical problems with flat or exponential functions.





Analyzing Algorithms

- Criteria:
  - Correctness
  - Amount of work done
  - Amount of space used
  - Simplicity, clarity
  - Optimality
- Asymptotic behaviour
- Different analysis techniques