7. Math

Fibonacci

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$

Fibonacci

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$

```
int fib(int n) {
   if (n <= 1) return n;
   return fib(n - 1) + fib(n - 2);
}
```

Fibonacci

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Easy Right?

Fibonacci

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$

```
int fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

Easy Right?

Don’t do this!
EVER!

Fibonacci

```
int fib(int n)
    if (n <= 1) return n;
    int f2 = 0, f1 = 1;
    for (int i = 2; i <= n; i++) {
        int prev_f1 = f1;
        f1 += f2; f2 = prev_f1;
    }
    return f1;
```

Use memoization to get linear time

Fibonacci

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$

$$(F_{n-2}, F_{n-1}) \times \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (F_{n-1}, F_n)$$

Fibonacci

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$

$$(F_{n-2}, F_{n-1}) \times \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (F_{n-1}, F_n)$$

Use fast exponentiation to make this O(logn)!

Fibonacci

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$

This is also cool to know (see here)

```
Fib(n) = ((1.6180339..)^n - (-0.6180339..)^n) / 2.236067977..
```

But it will give wrong results for n > 100 (rounding errors)

Primes

Useful in

1. cryptography
2. modulo arithmetic
3. as part of other algorithms (e.g. find divisors)

Primes

To check if a number is prime:

```
boolean isPrime(int n):
 for (int i = 2; i < n; i++) {
   if (n % i == 0) return false;
 return true
```

Primes

To check if a number is prime:

```
boolean isPrime(int n):
 for (int i = 2; i < n / 2; i++) {
   if (n % i == 0) return false;
 return true
```

Primes

To check if a number is prime:

```
boolean isPrime(int n):
 for (int i = 2; i <= sqrt(n); i++) {
   if (n % i == 0) return false;
 return true
```

Primes

To check if a number is prime:

```
boolean isPrime(int n):
 if (n == 2) return true;
 for (int i = 3; i <= sqrt(n); i+=2) {
   if (n % i == 0) return false;
 return true;
```

Primes

To check if a number is prime:

```
boolean isPrime(int n):
 for (int i : smallerPrimes(n)) {
   if (n % i == 0) return false;
 return true;
```

Primes

To find all primes smaller than n:

```
void sieve(int n):
    int primes[] = new int[n];
    for (int i = 2; i < n; i++)
        if (primes[i] == 0)
            mark all multiples
```

Primes

To find all primes smaller than n:

```
void sieve(int n):
   char primes[] = new char[n];
   for (int i = 2; i < n; i++)
      if (primes[i] == 0)
         mark all multiples
```

Primes

To find all primes smaller than n:

```
void sieve(int n):
    char primes[] = new char[n];
    for (int i = 2; i < n; i++)
        if (primes[i] == 0)
            mark all multiples
```

Could use bit optimizations for 8x less memory (a bit tricky to implement).

Primes

To find all primes smaller than n:

1. naive approach
   a. O(sqrt(N) * N), O(logn) memory
   b. O(sqrt(N) / log(N) * N), O(logn) memory
2. sieve
   a. O(log(log(N)) * N), O(n) memory

Gaussian Elimination

“Systems of linear equations arise in 75% of scientific computing problems”

S. Skienna

Gaussian Elimination

1. Start with row i = 0, col j = 0
2. Find first non-zero on col j = A[k][j]
3. Swap k and i
4. Divide row i by the A[i][j]
5. Update all rows below row i and to the right of j
    a. subtract (row i) * (first nnz element of row) to update

Gaussian Elimination

```
1.  Start with row i = 0, col j = 0
2.  Find first non-zero on col j = A[k][j]
3.  Swap k and i
4.  Divide row i by the A[i][j]
5.  Update all rows below row i and to the right of j
    a.  subtract (row i) * (first nnz element of row) to update
```

Row echelon form

Gaussian Elimination

```
1.  Start with row i = 0, col j = 0
2.  Find first non-zero on col j = A[k][j]
3.  Swap k and i
4.  Divide row i by the A[i][j]
5.  Update all rows below row i and to the right of j
    a.  subtract (row i) * (first nnz element of row) to update


6.  Start with row i = 0
    a.  find the first non-zero element (j)
    b.  compute the corresponding variable by
        i.  x[j] = A[][] - sum(k = j + 1 .. m, x[k] * A[k][j])
```

Gaussian Elimination

1. Gaussian elimination is OK for small systems

2. But there are some pretty big systems
   a. The Web
   b. Large physics simulation

Iterative Algorithms

Iterative solvers:

1. Guess a solution
2. Check if it's good
3. If not, guess another solution, based on
    a. previous guesses
    b. how far we are from the solution (residual)

Conjugate Gradient

```
for i in xrange(10 ** 3):
    Ap = a * p
    alpha = rsold / (p.T * Ap)
    x = x + multiply(alpha, p)
    r = r - multiply(alpha, Ap)
    rsnew = r.T * r
    if rsnew < EPS ** 2:
        return x
    p = r + multiply(rsnew / rsold, p)
    rsold = rsnew
```

Sparse Algebra

1. Don’t store the matrix as value, row, col
2. Store only nonzero values
3. Sparse systems arise in many situations
    a. power grids
    b. ocean modelling

Sparse Algebra

Pretty important research topic

- trade-offs between:
    - storage
    - computation, communication overhead
- dedicated architectures
- various formats
    - Block/Bit Compressed Row/Column,

Linear Algebra

It's usually best **NOT** to write your own:

1. Intel MKL, CUSP, LAPACK, ACML
2. These (can) take into account:
   a. various cache optimisations
   b. sparsity
   c. preconditioning

Recap

Overview

1. Complete Search
    a. First approach
    b. Bad complexity
    c. Can sometimes use pruning effectively

2. Divide and Conquer
    a. Useful Algo Design paradigm
    b. Search, Sort etc.

Overview

3. Greedy
   a. Pick “locally” best option at each step
   b. Usually really fast
   c. May be wrong (unless greedy property holds)

4. Dynamic Programming
   a. Can speedup algorithms by effective caching

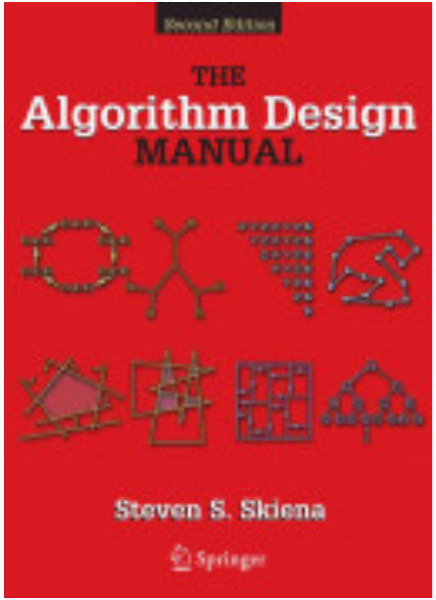
Overview

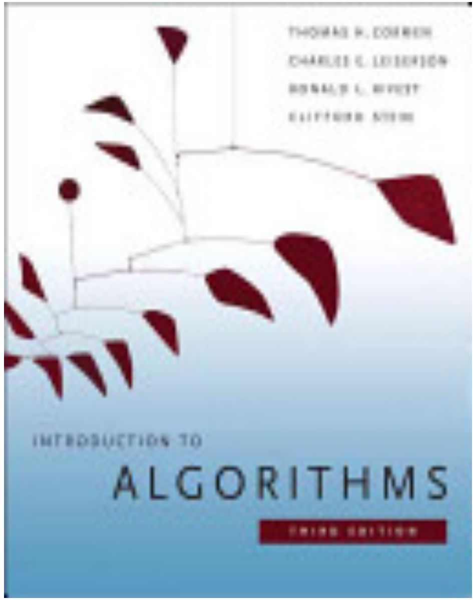
5. Graphs
   a. Natural database for relations
   b. Used in many areas of CS
      i. scheduling, logic problems, assignment etc.
   c. Classic algorithms
      i. DFS, BFS
      ii. Topological Sort
      iii. Minimum Spanning Tree
      iv. Dijkstra etc.

Overview

6. Geometry and Maths
    a. Heavily used in scientific computing
    b. Very nice algorithms (e.g. Graham Scan)
    c. Often tricky to implement
        i. floating point errors
        ii. edge cases etc.

Where To?


THE
Algorithm Design
MANUAL
Steven S. Skiena
Springer


INTRODUCTION TO
ALGORITHMS

Where To?

Practice:

1. https://projecteuler.net/problems
2. http://uhunt.felix-halim.net/
3. http://www.topcoder.com/tc?d1=tutorials&d2=alg_index&module=Static
4. http://codeforces.com/
5. http://code.google.com/codejam/

Questions ?

Feedback?